HIGH AVAILABILITY AND SCALABILITY SCHEMES FOR SOFTWARE-DEFINED NETWORKS (SDN)

A DISSERTATION
IN
Computer Science
and
Telecommunications and Computer Networking

Presented to the faculty of
the University of Missouri-Kansas City
in partial fulfillment of
the requirements for the degree

DOCTOR OF PHILOSOPHY

by
HYUNGBAE PARK

M.S., South Dakota State University, Brookings, SD, USA, 2007
B.E., Kwangwoon University, Seoul, South Korea, 2005

Kansas City, Missouri
2015

HIGH AVAILABILITY AND SCALABILITY SCHEMES FOR SOFTWARE-DEFINED NETWORKS (SDN)

Hyungbae Park, Candidate for the Doctor of Philosophy Degree

University of Missouri–Kansas City, 2015

ABSTRACT

A proliferation of network-enabled devices and network-intensive applications require the underlying networks not only to be agile despite of complex and heterogeneous environments, but also to be highly available and scalable in order to guarantee service integrity and continuity. The Software-Defined Network (SDN) has recently emerged to address the problem of the ossified Internet protocol architecture and to enable agile and flexible network evolvement. SDN, however, heavily relies on control messages between a controller and the forwarding devices for the network operation. Thus, it becomes even more critical to guarantee network high availability (HA) and scalability between a controller and its forwarding devices in the SDN architecture.

In this dissertation, we address HA and scalability issues that are inherent in the current OpenFlow specification and SDN architecture; and solve the problems using practical techniques. With extensive experiments using real systems, we have identified that

the significant issues of HA and scalability in operations of a SDN such as single point of failure of multiple logical connections, multiple redundant configuration, unrecoverable interconnection failure, interface flapping, new flow attack, and event storm. We have designed and implemented the management frameworks that deal with SDN HA and scalability issues that we have observed from a real system. The proposed frameworks include various SDN HA and scalability strategies. For SDN HA, we have developed several SDN *control path* HA algorithms such as *ensuring logical control path redundancy*, *transparency of a controller cluster*, and *fast and accurate failure detection*. We validate the functionalities of the proposed SDN HA schemes with real network experiments. The proposed SDN *control path* HA algorithms overcome the limitations of the current OpenFlow specification and enhance performance as well as simplify management of SDN *control path* HA. For SDN scalability, we have proposed and developed our management framework in two different platforms; an embedded approach in the OpenFlow switch and an agent-based approach with the SUMA platform that is located near the OpenFlow switch. These platforms include various algorithms that enhance scalability of SDN such as *Detect and Mitigate Abnormality* (*DMA*), *Modify and Annotate Control* (*MAC*), and *Message Prioritization and Classification* (*MPC*). We have shown that the proposed framework effectively detects and filters malicious and abnormal network behaviors such as new flow attack, interface flapping, and event storm.

APPROVAL PAGE

The faculty listed below, appointed by the Dean of the School of Graduate Studies, have examined a dissertation titled "High Availability and Scalability Schemes for Software-Defined Networks (SDN)," presented by Hyungbae Park, candidate for the Doctor of Philosophy degree, and hereby certify that in their opinion it is worthy of acceptance.

Supervisory Committee

Baek-Young Choi, Ph.D., Committee Chair
Department of Computer Science Electrical Engineering

Cory Beard, Ph.D.
Department of Computer Science Electrical Engineering

Yugyung Lee, Ph.D.
Department of Computer Science Electrical Engineering

E.K. Park, Ph.D.
Department of Computer Science Electrical Engineering

Xiaojun Shen, Ph.D.
Department of Computer Science Electrical Engineering

Sejun Song, Ph.D.
Department of Computer Science Electrical Engineering

CONTENTS

ILLUSTRATIONS

TABLES

ACKNOWLEDGEMENTS

First of all, I would like to thank my advisors Dr. Baek-Young Choi and Dr. Sejun Song for all their guidance, advice, and support throughout my doctoral research. Their great advice and guidance during my Ph.D. studies contributed to my growth in research skills of reading papers critically, discovering ideas, building up projects, and ultimately leading and managing projects throughout all phases of research. In addition, they have graciously guided me so I can balance my research and my hectic life with two kids.

I sincerely thank all of my committee members, Dr. Cory Beard, Dr. Yugyung Lee, Dr. E.K. Park, and Dr. Xiaojun Shen for all their help and sincere advices when I approached them with questions. Their comments have helped to clarify and improve this work a lot. I also would like to thank all the lab mates, faculty, staff, colleagues, friends, and previous students in my classes because even a short chat and a small exchange of smiles enlightened my day. I also thank Dr. Eun-Kyoung Paik at KT Advanced Institute of Technology for her insightful comments in the early stage of the work.

Lastly, but most importantly, I'm grateful to my family for making this possible. My parents Youngwoo Park and Kyungsook Jang and my wife's parents Daehyun Shin and Youngboon Song have always encouraged me when I was frustrated and taught me how to be patient and not to be anxious. My wife Sunae Shin and my two adorable daughters, Katie Subin Park and Claire Yebin Park, have always been my happiness and driving force during my doctoral research. Their love has been and will always be my momentum to move forward.

CHAPTER 1

INTRODUCTION

Modern computer networking is more complicated now than it ever has been. A proliferation of network-enabled devices and bandwidth-intensive applications lead to the massive growth of customer's demands for higher bandwidth and better quality of networks. As the networks progress, it is getting more difficult to efficiently manage them. Network volume and complexity come to the fore as the main reasons that hinder efficient network management. As the number of network devices in the network gets higher, operating expense (OPEX) of the network accordingly increases. In addition, as more network layers, various protocols, and multiple vendors are introduced in a given network, network operation and management becomes even more inefficient and difficult.

In order to grapple with closed, inflexible, complex, error-prone, and hard-to-manage production network problems, Software-Defined Networking (SDN) [4, 57, 78] has been proposed by many network companies and researchers. Particularly, fueled by increasing data center networking and cloud computing, SDN has been building up significant momentum towards production network deployment. This recently emerging concept of a network architecture supports the programmable control interfaces by separating and abstracting the control plane from the data plane. This centralized control plane, called an SDN controller, becomes the brain of the underlying network. SDN enables many features, such as traffic engineering and network virtualization, in the form of

an application on top of the controller of the network. In addition, SDN allows for rapid and simplified network exploration that improves network reliability, manageability, and security. Therefore, the centralized controller plays a very important role in controlling and operating for example, imposing policies and making decisions for routing, underlying network switches. Logically, it lies between northbound applications and southbound network devices. The OpenFlow protocol [63], which is managed by Open Networking Foundation (ONF) [61] is the de facto protocol that is being used for communication between the controller and the underlying network devices (e.g., OpenFlow switches). Northbound APIs are used for communication between the controller and the applications. These applications are developed to fulfill a specific purpose. The controller is an agent that connects applications and network devices and manages flow control to enable intelligent networking based on the applications' needs. As mentioned, this centralized architecture takes all the intelligence from the underlying network switches and leaves a flow table in the network switches. When the network switches receive new packets, they will forward these packets to the controller to decide where to send them.

High Availability (HA) of a network control system is important in real network operation. Thus, provisioning redundancies a priori, then detecting failures and invoking mitigation mechanisms are the necessary steps in action. In the traditional network, HA is solely limited to data paths so that the system maintains a certain level of availability such as Service-Level Agreement (SLA). In the SDN architecture, the issue of HA becomes more critical, especially for the controllers of SDNs, as they are responsible for the functions of the network switches. Furthermore, the SDN architecture poses more complexity

on HA issues by creating a couple of separate HA network domains such as a controller cluster network as well as control and data plane connection networks in addition to the data plane network. Although there have been a few recent studies that focus on the failures of switches or links connecting them in data plane, little work is found to consider the failures of the controller cluster network itself or to orchestrate the failure detection and recoveries of all the separate HA network domains. Another important aspect with regard to HA is fast and accurate failure detection. Detecting a failure quickly and accurately is also very critical to maintain HA of a system. This is because slow or incorrect failure detection delays the root cause analysis of the problem and delays recovery of the system. As a result, overall performance of the system's HA would be degraded. Therefore, we additionally focus on how fast we can detect failures in order to reduce the downtime of the network and improve HA of the network in the SDN environment.

Along with HA, scalability of a network system is also important in a real network operation. There has been some research conducted on the scalability issue of an SDN controller. That research can be divided into three types such as improving the capacity of the controller itself by using parallel processing and multi-threads, devolving some control functions to the OpenFlow switches, and clustering multiple controllers. Most of the existing approaches try to increase the system capacity to accommodate the increasing amount of network traffic between a controller and underlying switches. However, if we look at the network messages between a controller and underlying switches more closely, we can find each message has different importance according to the activities and status of the current network.

In this research, we study various aspects of controller's HA impacting the overall performance of SDN. We identify several critical HA issues and propose a solution for each problem. In addition, we also study and propose solutions for SDN scalability using prioritization and classification techniques.

1.1 Software-Defined Networks

SDN is an emerging computer networking paradigm that supports programmable interfaces which provide an agile and convenient way to customize the network traffic control. The main contribution of the SDN architecture is the disaggregation of the vertically integrated networking planes in order to improve network flexibility and manageability. The control plane of the vertical network stack is decoupled and abstracted to interact and handle all of the underlying network devices within its domain. It is logically centralized and is called an SDN controller. An SDN controller can run on a commodity server. With this centralized controller in the network, it gets a global view on the underlying network with ease. In addition, new services can be loaded up to the controller and each differentiated service can utilize the global view of the controller to achieve its optimized performance. Therefore, SDN brings many benefits such as easier traffic optimization, agile new feature deployment, and reduced management cost.

As illustrated in Figure 1, this centralized architecture takes all the intelligence from the underlying network switches and leaves a flow table in the network switches. Therefore, when the network switches receive packets, they will search the matching rules from the flow tables. Each flow table consists of flow entries and there are six

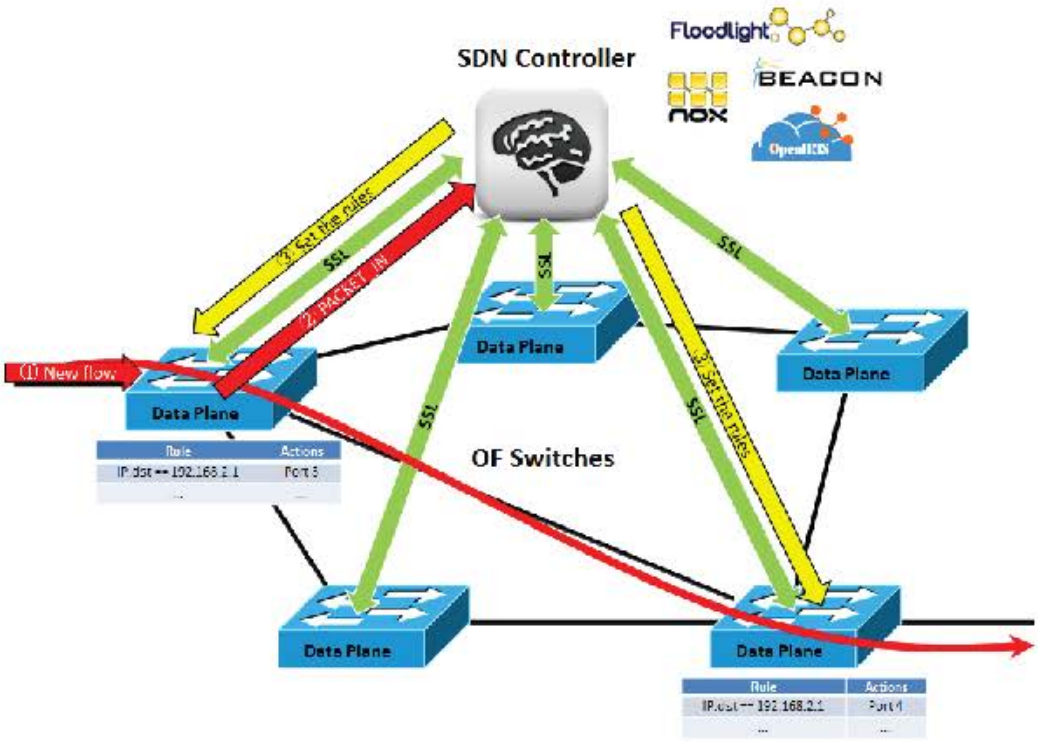


Figure 1: Software-Defined Networks (SDN) architecture

main components in each flow entry such as match fields, priority, counters, instructions, timeouts, and cookie. The match fields category, which consists of the ingress port and packet headers, is the most important factor to examine incoming packets. If there is a matching flow entry, the switch will handle the packets according to the associated action of its instructions category. If there is no matching flow entry (e.g., new packets), then they will forward these new packets to the controller in order to ask how to handle them.

Many companies, universities, and research institutes have been involved in developing and improving SDN for practical use. Major components of SDN are SDN controllers, OpenFlow protocol, and OpenFlow switches. Table 1 and 2 compares the

Table 1: Open source SDN controllers

Controller	Languages	OpenFlow Protocol	Copyright
Beacon [11]	Java	1.0	Apache 2.0 license
Floodlight [13]	Java	1.0	Apache 2.0 license
IRIS [14]	Java	1.0 $\sim$ 1.3	Apache 2.0 license
Maestro [3]	Java	1.0	GNU LGPL v2.1
Mul [15]	C	1.0 $\sim$ 1.4	GNU GPL v2.0
Nox [16]	C++ & Python	1.0	Apache 2.0 license
OpenDaylight [17]	Java	1.0 $\sim$ 1.4	Eclipse public license v1.0
Pox [18]	Pyhon	1.0	Apache 2.0 license
Ryu [19]	Python	1.0 $\sim$ 1.4	Apache 2.0 license
Trema [20]	Ruby & C	1.0	GNU GPL v2.0

specifications of the current SDN controllers. Various SDN controllers have been developed and are currently running commercially and academically such as Beacon [11], Floodlight [13], IRIS [14], Maestro [3], Mul [15], Nox [29], OpenDaylight [17], Pox [18], Ryu [19], Trema [20] and etc.

Being a detailed embodiment of SDN, OpenFlow [63] is a configuration language and protocol that abstracts the data plane of a networking device into a flow based architecture composed of a cascading set of classifiers and policy enforcement. The OpenFlow protocol is currently the de facto standard being used for the communication between an SDN controller and an OpenFlow switch. It is an open source project managed by Open Networking Foundation (ONF) [61]. It enables an SDN controller to control the forwarding plane of switches/routers. It also helps an SDN controller collect statistical information from the network in order to have a global view on the underlying network. The OpenFlow protocol is currently being implemented by major switch/router vendors

Table 2: Commercial SDN controllers

Controller	Languages	OpenFlow Protocol	Company
Big Network Controller	Java	1.0 $\sim$ 1.3	Big Switch Networks
ONE	Java	1.0 $\sim$ 1.4	Cisco
Contrail	Java & Python	1.0 $\sim$ 1.3	Juniper Networks
ProgammableFlow	Ruby & C	1.0 $\sim$ 1.3	NEC
SDN VE	Java & Python	1.0 $\sim$ 1.3	IBM
ViSION	Java	1.0 $\sim$ 1.3	HP
Vyatta	Java	1.0 $\sim$ 1.4	Brocade

to support and deliver OpenFlow-enabled products. Various OpenFlow switches are commercially available in the market.

1.2 Network Availability

Achieving network availability is one of the most important operational objectives of network service providers (NSPs). Availability is the fraction of a time that a system or component is continuously operational. Figure 2 describes terminologies related to network availability. HA can be measured by three main metrics such as *Mean Time Between Failures* (*MTBF*), *Mean Time To Repair* (*MTTRr*), and *Mean Time To Failure* (*MTTF*). *MTBF* is an expected average time between failures of a network component. *MTTRr* is an expected average time to restore a failed network component. The average downtime can be further divided into two parts such as *MTTD* and *MTTRc*. *MTTD* is an expected average time to detect a failed network component and *MTTRc* is an expected average time to recover the failed network component. Lastly, *MTTF* is a mean time to failure once the network component starts working normally. Therefore, the availability

Table 3: Network downtime according to the number of nines

Availability	Downtime per year	Downtime per month	Downtime per week
90%	36.5 days	72 hours	16.8 hours
99%	3.65 days	7.2 hours	1.68 hours
99.9%	8.76 hours	43.8 minutes	10.1 minutes
99.99%	52.56 minutes	4.32 minutes	1.01 minutes
99.999%	5.26 minutes	25.9 seconds	6.05 seconds

of the network component can be calculated by the formula as shown in Equation 1.1. Table 3 shows the operational performance (i.e., downtime) according to the number of nines. As we have more nines, we can say that the network is more stable.

$$Availability = \frac{MTTF}{MTBF} = \frac{MTBF - MTTD - MTTRc}{MTBF} \tag{1.1}$$

Many technologies have been developed to increase network availability and ensure the network reliability requirements. A traditional HA architecture supports link bundling, multipath routing, system redundancy mechanisms along with efficient state synchronization, and failure detection and handling protocols. These HA mechanisms are implemented in each network device as a distributed protocol to handle the network problems according to the dedicated network topologies.

The emerging concept of SDN decouples the control plane from the underlying network devices and abstracts it out as a centralized service. Many NSPs are very supportive of its deployment due to potential benefits such as operational cost reduction and enhanced system resilience. However, unlike traditional networks, the existing HA mechanisms may face many critical challenges to achieve the same Service Level Agreement (SLA) of HA for the network services in the SDN environment where the out-of-band

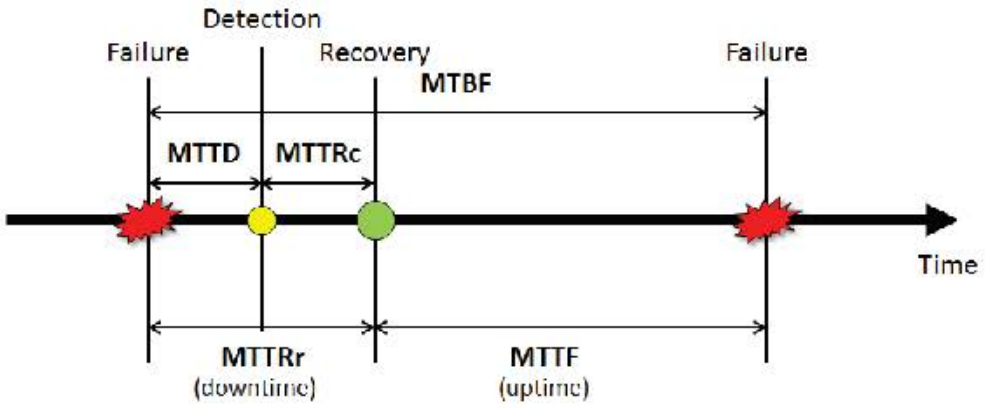


Figure 2: Network availability timeline

connections and controller connections exist between the control and data planes and between controllers, respectively. As illustrated in Figures 3 and 4, unlike traditional networks, the SDN architecture poses more complex network reliability domains by creating new connection network domains between the control and data planes as well as among the control plane. We named the connection network domains as the *control path*. The issue of HA becomes more crucial on the 'SDN controllers' than the 'OpenFlow switches', as well as it is significantly related to the scalability of the SDN controllers, as they are responsible for the intelligent decision of the OpenFlow switch policies.

1.3 Network Scalability

In the traditional network systems, the main network functionalities such as data, control, and management planes are distributed and embedded within the vendor specific networking devices and are managed remotely by EMSs [26], NMSs [60], OSSs,

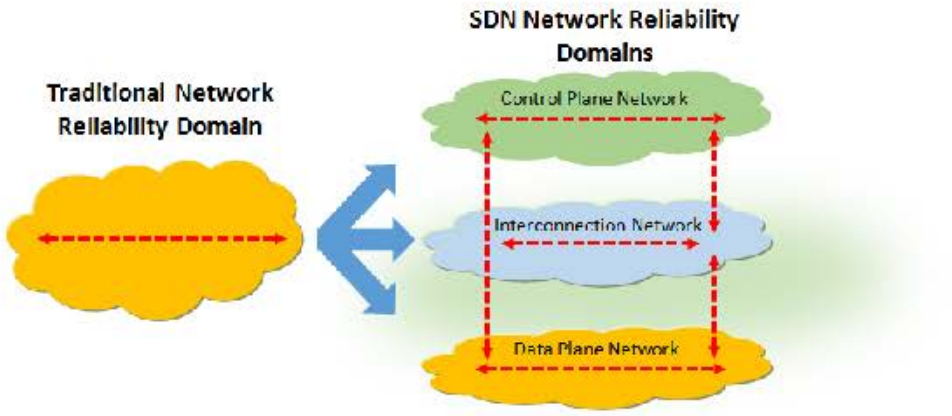


Figure 3: Multilateral SDN reliability domains

and BSSs [65] through provisioning and configuration. As the network systems become bigger, faster, and more complex over multiple administration domains and diverse components, they need to handle multiple protocols with cross-layer interactions, support various applications and services over multi-tenant policies, and are managed over uncertain underlying topology and internal structure. At the same time, the network services and applications are expected to be deployed quickly and more dynamically on the large-scale networking systems while insuring security, reliability, performance, traffic isolation, end-to-end virtualization and priority treatment. However, providing quick and easy dynamic network adaptability is an intrinsically difficult problem for legacy network systems, as they can barely cope with the complexity through the layers of the extensive and expensive remote provisioning and configuration.

More specifically, traffic and resource monitoring is the essential function for

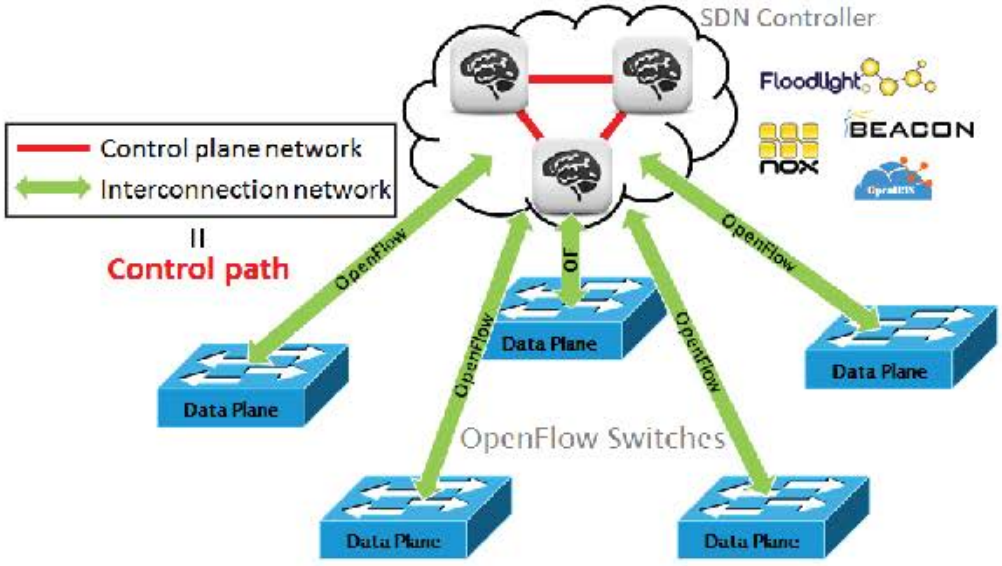


Figure 4: Definition of control path

large-scale enterprises, service providers, and network operators to ensure network reliability, network availability, and security of their resources. For this reason, many large-scale enterprises and providers have been investing in various stand-alone dedicated monitoring solutions. However, they find that a proprietary and dedicated stand-alone hardware-based appliance per feature is inflexible, slow to install, and difficult to maintain as well as being cost prohibitive. Because of such a huge required investment, many enterprises are looking for outsourcing alternatives and providers are also looking for means to reduce this cost.

As networks are evolving towards software defined networks, control and management functions are logically centralized and real-time, scalable, and dynamic monitoring of managed resources is a key to make precise control decisions. In addition

to this, virtualization (e.g., Network Virtualization (NV) and Network Function Virtualization (NFV) [59]) of the underlying computing, network resources including Layer 4 (transport) $\sim$ Layer 7 (application) capabilities, and network services has emerged as a complementary approach along with SDN. Specially, NFV provides more flexible and programmable monitoring functions which are used to be built in specific hardware. To provide such flexible and programmable monitoring functions, virtualization of the monitoring function itself can be a solution. That is, a monitoring function of a particular objective can be instantiated on demand in real-time whenever a new monitoring requirement occurs and can dynamically be deleted once its demand completes. Since the main benefit of NFV is the chaining of its functionality, the virtual monitoring function can be utilized as a part of such a virtual function chaining. Even though SDN and NFV promise flexibility, simplicity, and cost-effectiveness, the abstractions towards the remote and centralized control and management tend to introduce the following challenging problems:

- *Scalability problem*: The proposed SDN architecture tends to open up control messages between the controllers and the forwarding devices to the communication networks, which is spatiotemporally concentrated around the centralized controller. Several SDN scalability research studies such as DevoFlow [22], DIFANE [80], ONIX [50] show that SDN imposes excessive control traffic overheads in order for the controller to acquire global network visibility. More significantly, the overhead will be further increased by traditional network management events as well as the application specific control traffic, as they may use the same physical network paths, buffers, and I/O channels at the same time. If the overheads are not

controlled properly, they can cause various scalability problems on the networking devices, controllers, and the network itself including slow message processing, potential message drop, delayed root cause analysis, and late responses to urgent problems.

- *Inaccurate and unreliable management problem*: In traditional network systems, the network management practice mainly takes remote approaches coping with the network-centric infrastructure. However, since the network events occurring within the network should be inferred by the remote management systems, the potential network problems are often accumulated and enlarged, and their diagnosis is delayed, inaccurate, unreliable, and not scalable. SDN's remote and centralized control tends to extend the legacy network management problems into the control plane.

- *Multiple management pillar problem*: Although SDN's management plane is a relatively unexplored area, either the SDN controller agnostic application of the incumbent management protocols or the full integration of the management plane into the controller protocols such as OpenFlow cannot be the viable approach for the highly dynamic SDN management. Moreover, there are growing expectations for the fine grained management of the customer specific services and applications. Many existing SDN approaches evidence that SDN allows a variety of heterogeneous application interfaces and protocols in the data plane. For example, according to the most recent OpenDaylight controller architecture, SDN control and management provides an expandable facility for the protocol specific management. Although

OpenFlow gained its visibility as the most fundamental pillar to support SDN, it is actually only one of many programming interfaces to the data plane. Multiple other interfaces and protocols such as OVSDB [66], SNMP [72], and various NFV applications also play a significant role in the evolution of the SDN management architecture.

- *Heterogeneous deployment problem*: Moreover, from the practical network operation point of view, SDN deployment may take a gradual transition instead of an all-in-one-night transition. Therefore, integrating existing services and protocols with SDN is an essential step for the transition. Also, some of the network systems may want to employ the SDN functions only partially. For example, inter data center networks may want to use the provisioning functionality only, but use their existing management tools. This will result a heterogeneous management environment. A complex combination of multiple and heterogeneous management channels introduces a significant scalability problem.

We have investigated various network service aspects including agility, accuracy, reliability, and scalability in order to identify an effective SDN network management system. This dissertation intensively focuses on scalability issues. We propose a filtering and common processing module that facilitates various communication interfaces to collect network events. It also provides common filtering and event mitigation functions to simplify the event processing for the user-defined monitoring modules. In order to validate the functionalities of our proposed schemes, we implemented the proposed schemes and

metrics in OpenFlow with OpenWrt [54] based switches. In collaboration with Electronics and Telecommunications Research Institute (ETRI), we also implemented our proposed modules in an intelligent management middlebox called Software-defined Unified Monitoring Agent (SUMA) [8] that becomes one logical point of intelligence for the integrated management services. SUMA is an essential switch-side middlebox that provides control and management abstraction and filtering layer among vNMS, SDN controllers, legacy NMS, and OpenFlow switches. SUMA performs light weight event detection and filtering, and the correlation will be conducted in vNMS. The two-tier framework is used to balance the performance impact between network devices and controllers, to provide scalability, and to ensure dynamic deployment.

1.4 Objectives of the Dissertation

The objectives of the dissertation are to suggest new ways to remove or reduce problems of the existing solutions and the current OpenFlow specification and to develop management frameworks that improve HA and scalability of the current Software-Defined Networking systems.

The proposed framework handles two different issues (e.g., SDN HA and scalability) and consists of two separate frameworks such as the *SDN Control Path High Availability Management Framework* and the *SDN Scalability Management Framework* as illustrated in Figure 5. Each framework is divided into several components that deal with specific issues inherent in the SDN architecture and the current OpenFlow specification. The *SDN Control Path HA Management Framework* includes several components

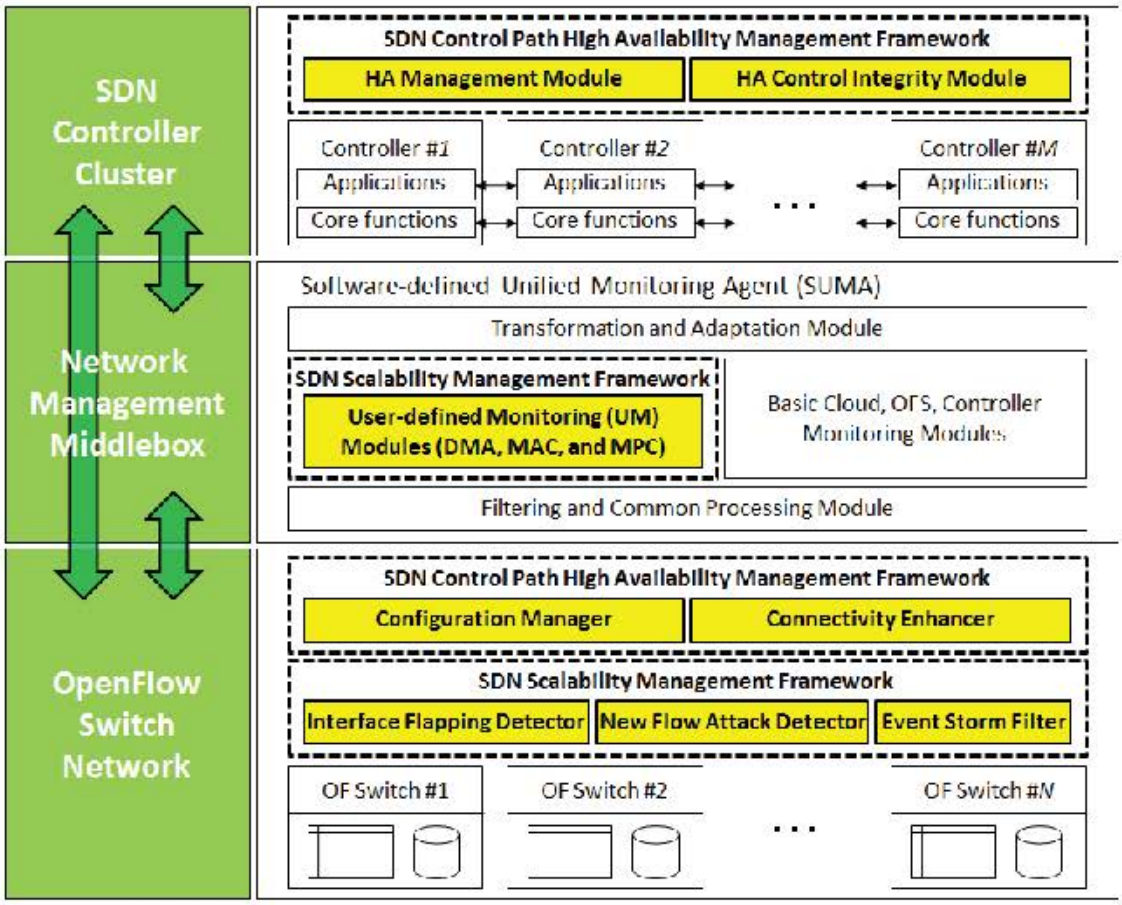


Figure 5: Overall system architecture

in the SDN controller and the OpenFlow switch. The *HA Control Integrity Module* in the SDN controller and the *Connectivity Enhancer* in the OpenFlow switch will provide enhanced HA performance. The *HA Management Module* in the SDN controller and the *Configuration Manager* in the OpenFlow switch will communicate each other to provide automated HA manageability. The *SDN Scalability Management Framework* includes several components and is designed and implemented in two different platforms such as

the OpenFlow switch as an embedded approach and the SUMA as an agent-based approach. The proposed components such as *UM Modules*, *Interface Flapping Detector*, *New Flow Attack Detector*, and *Event Storm Filter* provide scalability against several SDN scalability threats.

1.5 Scope and Contribution of the Dissertation

In this dissertation, we focus on two aspects of SDN management such as HA and scalability. The main contributions of this dissertation are as follows.

- We analyze the current OpenFlow specification and identify critical and practical HA issues that are newly introduced in SDN. The SDN control path HA management framework that includes various management modules and algorithms is designed and implemented to improve performance and manageability of HA. Two Korean patents [48, 69] have been published and a conference paper has been published in [68].

- We identify new types of scalability threats and propose distinctive approaches compared to the existing solutions to resolve SDN scalability issues. One Korean patent [81] has been published. This research has been published in two conference papers [6, 8] and one journal paper [7] in collaboration with ETRI.

1.6 Organization

The rest of this dissertation is organized as follows. In Chapter 2, we review related work dealing with the HA and scalability issues of SDN. Before we cope with

the HA and scalability issues of SDN, we discuss the traditional network availability in Chapter 3. In Chapters 4 and 5, we identify new problems of SDN in regards to HA and scalability issues and propose their practical solutions. Finally, Chapter 6 summarizes and concludes this dissertation and discusses future research goals.

CHAPTER 2

RELATED WORK

As the control plane in SDN is separated from the data plane and becomes a remote and centralized controller of the network, two major operational problems arise. First of all, HA issues in the SDN controller become very critical. One centralized controller for the network means a single point of failure. Since the controller is the brain of the network, the network could be easily disrupted by a malfunction in the SDN controller. In addition, since the underlying forwarding devices (e.g., OpenFlow switches) don't have their own decision engines, delays may be experienced while recovering from switch failures (e.g., hardware and software) as compared to legacy networks. Therefore, it is also critical to have a fast recovery mechanism to improve HA of the data plane. These concerns for HA issues motivated several research projects on HA in SDN. We will discuss them in detail in the following sections. Secondly, as the number of underlying network devices, protocols, and applications running on top of the SDN controller drastically increases, the capacity of the controller may not be enough to handle all the requests from the network, protocols, and applications. In addition, an OpenFlow switch may experience congestion when it receives more packets than its peak capability or is under malicious attacks. These issues motivate researchers to study scalability of the SDN controller. Therefore, in order to provide a highly reliable and robust SDN environment, we have to deeply consider these two major properties of the SDN controller.

In terms of HA in SDN, two types of issues have been studied so far. One is data plane HA and the other is control plane HA. Data plane HA of SDN can be further categorized into two topics such as fast failure detection on a data plane and HA for application servers that are running in the SDN environments. The scheme for the fast failure detection on a data plane utilizes the OpenFlow switch's link signals to check connectivity among neighboring switches or delegates fault management to the OpenFlow switches by extending the OpenFlow protocol to support the monitoring function. The scheme for the server HA mainly focuses on HA between OpenFlow switches and multiple server replicas [23, 47, 49, 52]. In addition to the above proposed HA strategies, it is also significant to detect failures in the network fast and accurately so the network can recover in a timely manner so as to maintain a highly available system [77]. There is little work done for fast failure detection in the SDN area. The existing research has focused on a data plane network. In comparison with existing research work, our research direction is unique, in that it mainly focuses on the HA issue of controller networks and a network between a controller and OpenFlow switches.

As previously mentioned, the separation of the control plane from the data plane introduces a centralized SDN controller. Since the SDN controller administers the underlying network and manages all the flows over the underlying network devices, it is easy to expect that the SDN controller may have an intrinsic scalability issue. Along with the HA research work, there has been some research conducted on the scalability issue of the SDN controller. That research can be divided into three types. The first type is dedicated

Table 4: High availability mechanisms

Mechanisms	Protocols
Link bundling	Link Aggregation Control Protocol (LACP) [36], EtherChannel [33]
Multipath routing	Equal-Cost Multi-Path routing (ECMP) [31]
System redundancy	Virtual Router Redundancy Protocol (VRRP) [41], Host Standby Router Protocol (HSRP) [35], Resilient Packet Ring (RPR) [39]
State synchronization	Non-Stop Routing (NSR) [38], Non-Stop Forwarding (NSF) [37], Stateful Switch-Over (SSO) [40]
Failure detection and handling	Ethernet Automatic Protection Switching (EAPS) [30], Ethernet Ring Protection Switching (ERPS) [32], Fast Re-Routing (FRR) [34]

to improving the capacity of the controller itself by using multi-cores with parallel processing and multi-threads [3]. The second type is devolving some control functions to the OpenFlow switches [22, 47, 56, 80]. These hybrid approaches allow some degree of intelligence to the OpenFlow switches. By offloading some control functions from the controller, they expect to reduce workloads imposed on the controller. We will see in detail what functions/intelligence are left in the OpenFlow switches. Last but not least, the third type of solution is clustering multiple controllers [1, 50, 51, 75]. These approaches show how they can synchronize global visibility of network state changes across the multiple controllers in the cluster.

2.1 Related Work of Traditional Availability Issues

HA is a well-established research topic and many technologies have been developed to increase network availability and ensure network reliability requirements. As

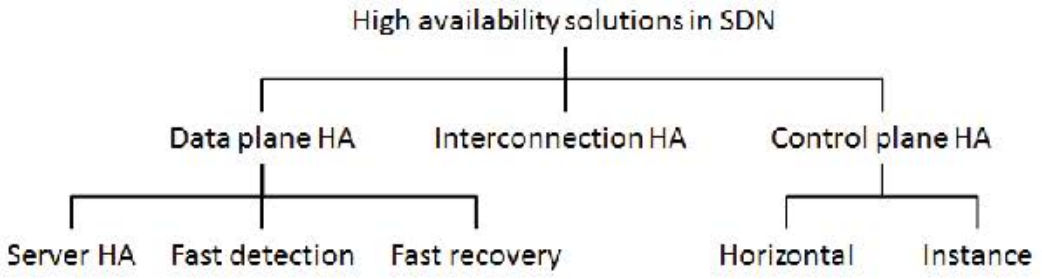


Figure 6: High availability classification in SDN

shown in Table 4, the traditional HA architecture supports link bundling, multipath routing, system redundancy mechanisms along with efficient state synchronization, and failure detection and handling protocols. These HA mechanisms are implemented in each network device as a distributed protocol to handle network problems according to the dedicated network topologies. Most of the implementations of these HA mechanisms are proprietary. Therefore, they are not readily available in the SDN environment. Even though LACP [36] and VRRP [41] can be easily adopted to the SDN system, they don't cover the synchronization between the SDN controllers, correlation of failures between the control plane and the data plane, and interconnection HA. Therefore, we need sophisticated HA mechanisms specifically designed for SDN.

2.2 Related Work of High Availability Issues in SDN

In Section 2.1, we have presented types of HA techniques and discussed the current HA mechanisms in traditional networks. HA is a well-known research topic and

Table 5: High availability research: comprehensive view

Ref.	Data plane HA			Interconnection HA	Control plane HA	
	Server HA	F.D.	F.R.		Horizontal	Instance
[77]	√					
[23]		√				
[47]		√	√			
[49]			√			
[42]					√	
[75]					√	
[50]					√	
[1]					√	
[3]						√

F.D.=Fast detection, F.R.= Fast recovery

well-established. However, these HA features don't fully consider the correlation between failures of the control plane network and the interconnection network that are newly introduced in SDN. There have been a few recent studies that focus on the failures of OpenFlow switches or links connecting them to facilitate the HA feature in the data plane of SDN and the controller cluster networks for improving both HA and scalability. In this section, we categorize HA issues in SDN into two topics; data plane HA and control plane HA and talk about the current research work. Table 5 presents a comprehensive view of the current network HA research in SDN. The details are explained in the following sections.

2.2.1 Data Plane High Availability

As we mentioned, data plane HA in SDN can be further categorized into two topics application server HA and fast failure detection. First of all, we discuss the current research work related to the application server HA. The study on application server HA in SDN can be found in [77]. The authors proposed RuleBricks that provides HA in existing OpenFlow policies. It primarily focuses on embedding HA policies into OpenFlow's forwarding rules. They implemented RuleBricks by utilizing an expressive brick-based data structure instead of naive tree-based data structure. They show that RuleBricks maintains linear scalability with the number of replicas (i,e,. *backup rules*) and offers approximately 50% reduction in the *active rule set*.

Now, we discuss the current research work in the area of fast failure detection and recovery on the data plane. As we discussed in the definition of network availability, it is very important to quickly detect failures in the network as well as to recover the network as soon as possible after failure detection. Fast failure detection and network recovery in a timely manner maintain a highly available system. There have been a few studies conducted on fast failure detection and recovery in SDN and most of them have focused on the data plane network. Desai et al. [23] proposed an algorithm that utilizes the OpenFlow switch's link signal to check the connectivity among neighboring switches for fast failure detection. This scheme notifies all the neighboring switches of the link failure in order to refrain from sending messages in the direction of the failed link so it can minimize unnecessary traffic in the network and reduce the effects of link failures. Their algorithm enables failure detection faster than the controller which identifies failed

links through heartbeat messages and then sends out an update. However, their algorithm does not contribute to the recovery of the network. Kempf et al. [47] also considers fast failure detection and recovery by extending the OpenFlow protocol to support a monitoring function on OpenFlow switches. They followed the fault management operation of MPLS-TP for the implementation and achieved fault recovery in the data plane within 50 ms. Kim et al. [49] proposed an SDN fault-tolerant system, named CORONET (controller based robust network), that mainly focuses on recovering the data plane network from multiple link failures. Their proposed modules can be implemented and integrated into the NOX controller. They summarized challenges on building a fault-tolerant system based on SDN but they didn't describe the proposed modules in detail.

2.2.2 Control Plane High Availability

Along with data plane HA, control plane HA has also been studied for various aspects. Hellen et al. [42] discussed about controller's physical placement in the network. They tried to optimize the number of controllers and their location in the network. By connecting an OpenFlow switch to the closest controller in the network, it can reduce control delay and contribute to improvement of network high availability. Tootoonchian et al. [75], Koponen et al. [50], and Berde et al. [1] proposed HyperFlow, ONIX, and ONOS, respectively. These proposed frameworks establish one logical controller consisting of physically distributed controllers in the cluster. Since they run on multiple physical controllers, the slave controllers can operate the network when the master controller goes down. Even though HyperFlow, ONIX, and ONOS consider some aspects of reliability of

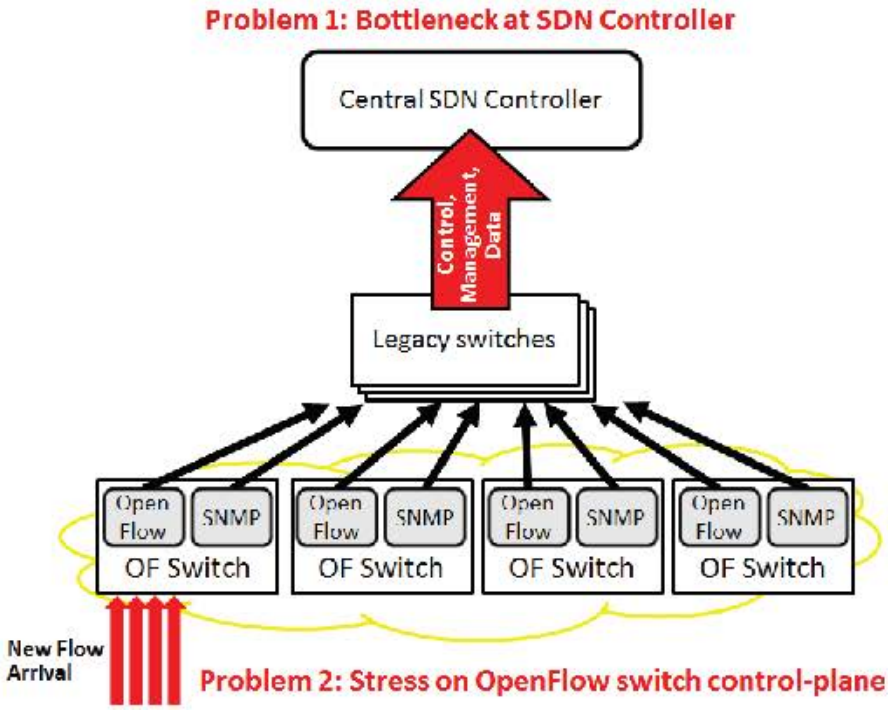


Figure 7: Two major scalability issues in SDN

the controller cluster via the distributed physical controllers, their main concerns are scalability and synchronization of network status among multiple physical controllers. Cai et al. [3] proposed the Maestro controller which supports software HA. A task manager of Maestro manages incoming computations and evenly distributes work to each SDN controller instance at each core of the processor. Since it exploits a multi-core architecture, it can re-distribute the work evenly at the time of the core crash or software crash.

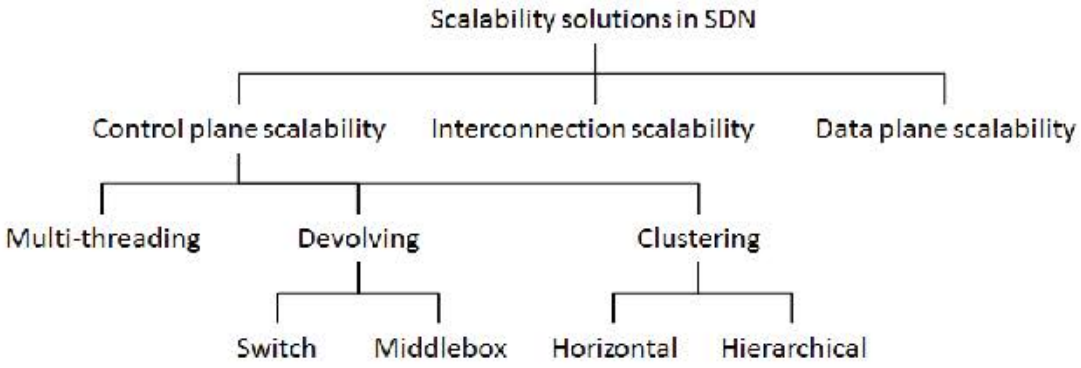


Figure 8: Scalability classification in SDN

2.3 Related Work of Scalability Issues in SDN

Along with the HA research work, there has been some research conducted on scalability issues in SDN. Figure 7 illustrates the scalability problems that can be addressed in the current SDN architecture. The first problem can be seen in the controller. As introduced, SDN relies on a centralized controller to operate the underlying network and opens up control messages to communicate between the controller and the forwarding devices. As the size of the underlying network gets bigger, relying on a single controller for the entire network might not be feasible. The second problem can be observed in the OpenFlow switch. Unlike a traditional network, the forwarding device in SDN has to communicate with the SDN controller to make a decision for forwarding or routing and to get network policies. Specially, it is a mandatory procedure for an OpenFlow switch to send new flow packets to the SDN controller in an encrypted format such as the packet-in message in order to cope with them. This can create additional workload and saturate the

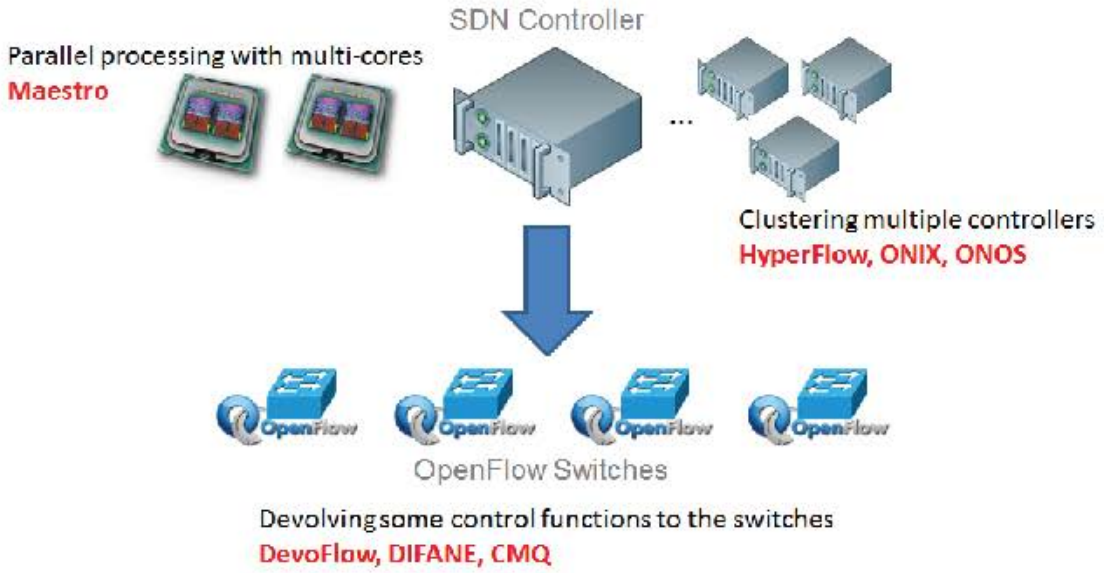


Figure 9: Existing solutions to the SDN scalability issues

OpenFlow switch.

Current scalability research in SDN primarily focuses on the SDN controller. As shown in Figure 9, that research can be divided into three types such as improving the capacity of the controller itself by using parallel processing and multi-threads, devolving some control functions to the OpenFlow switches, and clustering multiple controllers. Table 6 presents a comprehensive view of the current scalability research in SDN. Details are explained in the following sections.

2.3.1 Controller Enhancement with Multi-threading

The first type of solution tries to improve the capacity of the controller itself. Cai et al. [3] proposed the Maestro controller for scalable OpenFlow network control.

Since the SDN controller is the only brain of the network that copes with all the requests from the underlying network devices, it could be a performance bottleneck in the network system. The authors exploit parallelism to improve the capacity of the controller. They implemented Maestro in order to support multi-core processors with parallel processing and multi-threads. Their experiments show that the throughput of Maestro can achieve near linear scalability on a multi-core processor server.

2.3.2 Devolving Control Functions

The second type of solution is devolving some control functions to the OpenFlow switches. There are several well-known papers DIFANE [80] and DevoFlow [22]. DIFANE runs a partitioning algorithm that divides the rules evenly and devolves those partitioned rules across *authority switches*. These *authority switches* will handle new flows instead of the controller. DevoFlow mainly reduces the interactions between OpenFlow switches and the SDN controller using filtering and sampling such as rule aggregation, selective local action, and approximating techniques. Another devolving solution, called Control-Message Quenching (CMQ), is proposed by Luo et al. [56]. The switch with CMQ sends only one packet-in message for each source-destination pair, suppresses and enqueues the following un-matching packets until the switch receives a packet-out or a flow-mod message and installs the matching rule in its flow table. This reduces unnecessary packet-in messages from the OpenFlow switch to the controller. Lastly, the work done by Kempf et al. [47] also can be considered as one of the devolving schemes. The authors claimed that the centralized fault management has serious scalability limitations.

Table 6: Scalability research: comprehensive view

Ref.	Focus		Scalability method			Devolving		Cluster	
	Cont.	Switch	M.C.	Devol.	Cluster	Switch	M.B.	Horizontal	Hier.
[3]	√		√						
[80]	√			√			√		
[22]	√			√		√			
[56]	√			√		√			
[47]	√			√		√			
[75]	√				√			√	
[50]	√				√			√	
[1]	√				√			√	
[51]	√				√			√	
[79]	√				√				√
[70]	√				√				√
[53]	√				√				√

Cont.= Controller, M.C.= Multi-cores with multi-threads, Devol.= Devolving, M.B.= Middlebox, Hier.= Hierarchical

Therefore, they proposed a scheme which delegated fault management to the OpenFlow switches by extending the OpenFlow protocol to support the monitoring function.

2.3.3 Clustering Multiple Controllers

The last type of solution is clustering physically distributed controllers into a logically centralized controller in order to increase the capacity of the controller. There are two types of clustering techniques such as horizontal clustering and hierarchical clustering. In the horizontal clustering, each controller plays a role of master or slave. They could have the same functionalities or may have different functionalities based on the

configuration and implementation. In HyperFlow [75], the authors tried to provide SDN controller's scalability by using as many SDN controllers as necessary while keeping global visibility of link state changes. However, this scheme has lower visibility for the flow-level statistics. Koponen et al. [50] proposed ONIX which is also a distributed control platform for large-scale networks. And, it provides more general APIs than earlier systems, so it is easier to distribute controllers and collect fine-grained flow statistics with the distributed controllers. Berde et al. [1] proposed ONOS. It is an experimental open source distributed SDN operating system which provides scalability for the SDN control plane and achieves strong consistency of global network visibility. Krishnamurthy et al. [51] tried to improve the performance of the current distributed SDN control platforms by proposing a novel approach for assigning SDN switches and partitions of SDN application state to distributed controller instances. The authors focused on two metrics such as minimizing flow setup latency and minimizing controller operating costs. Their scheme shows a 44% decrease in flow setup latency and a 42% reduction in controller operating costs.

The second clustering technique uses a hierarchical structure. Controllers in the cluster can be classified into two types of controllers; a super controller and a regular controller. Yeganeh et al. [79] proposed an efficient and scalable framework that offloads the control applications by separating the controllers into two different roles such as a root controller and a local controller. The root controller processes rare events and while highly replicated local controllers cope with frequent events. The local controllers are not connected each other. Therefore, they only handle the local events that require the local

visibility. However, since the root controller maintains the network-wide global visibility, the root controller is involved in packet processing that requires the global network state. Park et al. [70] proposed a novel solution, called RAON, that recursively abstracts the controller's underlying networks as OpenFlow switches to reduce the complexity. In this architecture, the networks of the lower-level controllers are abstracted as big OpenFlow switches. This abstraction extracts the relationship between two different networks that are operated by physically different controllers. Therefore, all the ingress and egress ports of the network become the ports of the logical OpenFlow switches. Lee et al. [53] proposed a hierarchical controller structure with a super controller that collects global visibility from the lower-level controllers. Their main contribution is defining northbound message formats to realize the hierarchical controller in the field. They defined three different types of messages; normal messages, bandwidth_event messages, and delay_event messages. New types of messages such as bandwidth_event messages and delay_event messages are added in order for a super controller to quickly respond to abnormal events from the underlying network operated by the lower-level controllers.

CHAPTER 3

MEASUREMENT AND ANALYSIS OF AN ACCESS NETWORK'S AVAILABILITY

Before we cope with the details of the SDN high availability issues, we will discuss the network availability in a traditional network. In this chapter, we present our work on the measurement and analysis of the access network's health. Understanding the health of a network via failure and outage analysis is important to assess the availability of a network, identify problem areas for network availability improvement, and model the exact network behavior. However, there has been little failure measurement and analysis work devoted to access networks. We carry out an in-depth outage and failure analysis of a university campus network (University of Missouri-Kansas City) using a rich set of node outage and link failure data and topology information. We investigate network availability, the attributes of hardware/software and misconfiguration problems of the networks, the relation of link failure and node outage, and correlations between layers of a hierarchical network. For this dissertation, we mainly focus on network availability.

3.1 Campus Network Architecture and Data Sets

In this section, we describe the architecture of the campus network and the data sets we used for the availability measurement and analysis. The campus network of our study is designed in a hierarchical manner which is a common practice of campus or enterprise networks [10]. It provides a modular topology of building blocks that allow

the network to evolve easily. A hierarchical design avoids the need for a fully-meshed network in which all network nodes are interconnected. The building block components are the access layer, the distribution layer, and the core (backbone) layer as shown in Figure 10. The building blocks of modular networks are easy to replicate, redesign, and expand. There is no need to redesign the whole network each time a module is added or removed. Distinct building blocks can be put in-service and taken out-of-service with little impact on the rest of the network. This capability facilitates troubleshooting, problem isolation, and network management. In a hierarchical design, the capacity, features, and functionality of a specific device are optimized for its position in the network and the role that it plays. The number of flows and their associated bandwidth requirements increase as they traverse points of aggregation and move up the hierarchy from the access layer to the distribution and core layers.

In earlier years - until 2007, the UMKC network had 2 core routers in the core layer, 38 routers in distribution layer, and 373 nodes in the access layer. Since then, the core layer has increased to 3 routers. The new core router was added more recently to aggregate some part (e.g., dormitory area) of our campus wired and wireless networks. In the distribution layer, there are currently 54 routers. The access layer has about 571 nodes and includes wireless access points, switches that connect to end-systems directly, and switches that aggregate other switches.

We collected the node outage data as well as the link failure data from the university campus access network. As for network topology, we had the direct and complete network topology information available for the network operators. We used the naming

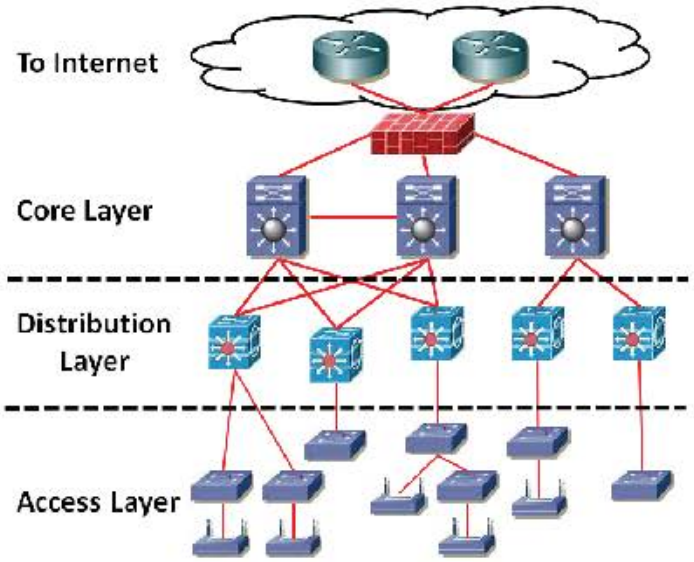


Figure 10: Hierarchical access (university campus) network design

conventions of devices to classify and relate devices, and utilized the topology information tool, called 'Intermapper'. Additionally, we incorporated vendors' documents in regards to the causes and recommended actions, and discussed the network operators' anecdotal comments on special events and actions. To the best of our knowledge, those data are the most extensive and complete data used in network failure and outage analysis.

Node outage data was gathered by SNMP polling and trap, and it is from April 7, 2005 till April 10, 2009 with 42,306 outage events. The polling time varies from 2 to 5 minutes depending on the monitored devices. The outage event time is recorded in minutes, and the outage duration is measured in seconds. Link failure data, which is UPDOWN messages from each device sent to a central syslog server, was collected from the syslog. The period of data is from October 1, 2008 to October 5, 2009. Among the

many types of syslog error messages, we only consider 'LINK-3-UPDOWN' messages as pertaining to failure analysis. The 12 month data contains roughly 46 million syslog messages, of which 3.8 million messages represent 'LINK-3-UPDOWN'. Syslog data has a slightly different format depending on the device vendors and router OSes. However, the campus network consists of routers and switches from mostly Cisco, providing a similar format of syslog messages. Note that a link failure can occur due to software/hardware malfunction, natural or human-caused incidents, and it may not lead to service outage due to redundancy or recovery mechanisms.

There may be some possible artifacts in the data, however, due to in-band (the monitoring data follows the same physical path as the user data) monitoring, the SNMP polling interval, and nature of protocol. Failure or outage reporting can be affected by the topology of the network. Any failure that is on the path to the monitoring system would result in an outage being reported for all devices on the path, though it is possible that the issue only affected one host. If connectivity is lost between the sending device and the syslog server, the syslog event would not be recorded. Additionally, as syslog uses the UDP protocol, data can possibly be lost due to transient network congestion, CPU load, OS patching, EIGRP reconvergence, STP (Spanning Tree Protocol) recalculation, etc.

3.2 Network Availability Measurement

In this section, we investigate the availability of network nodes over four years. The node availability is the percentage of the uptime of a node. For each node i, let

$NodeAvail(i)$ represent the node availability over a month, and it is computed as below.

$$NodeAvail(i) = \frac{TTBF(i) - TTTR(i)}{TTBF(i)} \times 100 \tag{3.1}$$

where $TTBF(i)$ is the monthly Total Time Between Failure of node i, and $TTTR(i)$ is the monthly Total Time To Repair of node i. This formula shows that we can improve the network availability by increasing the time between failures and reducing the time to recover. As we will see from Chapter 4, we focus on reducing the time to recover. Then, we compute the mean node availability (MNA) of all the nodes in the network.

$$MNA = \frac{\sum_{i=1}^{m} NodeAvail(i)}{m} \tag{3.2}$$

where m is the number of nodes in the network. The data set of the TTR per device is derived from the node outage data shown in Figure 11 and the monthly mean node availability for the period from April 2005 to March 2009 is shown in Figure 12. We only use the events of the unplanned outages. We exclude the planned outages from the results to focus on the impact of the unplanned outages on the network availability. We observe that the network maintains two or three-nine availabilities in most months. It appears to have fairly healthy performance, even though, to the best of our knowledge, there are no measurements available for comparison. Delving into the details, we notice one big drop in the availability in August 2006. After consulting with the network operator, we discovered that there was a fire near a building that took out the fiber that month. We also observed slightly lower availabilities in several months during 2007 and 2008. To concisely explain those occasions, we list possible reasons for the unidentified outage events below, based on the consultations with the network operator. Then, we summarize

Table 7: Long term outages in the access layer

Year	Month	Causes of Degradation
2006	Jun	Reason 1&2&3
	Aug	Fire accident
2007	Apr	Reason 2
	Dec	Reason 2
2008	Jun	Power outages all over campus
	Jul	Backup link installation & OS bugs
	Nov	Reason 2&3

the causes that made the performance degradation for each case, in Table 7.

- Reason 1: Issues that were either out of our control to correct any more quickly (e.g., power problems)
- Reason 2: Issues that didn't justify an on-call response, thus were dealt with in the morning
- Reason 3: Issues that we were working on but took a while to fix
- Reason 4: Issues that affected monitoring but not operation

3.3 Network Availability Analysis with Splunk

We also discuss network availability analysis using Splunk and tailored scripting. Splunk is a big data analysis tool and provides easy classifications and statistics in a convenient format by efficiently capturing, indexing, and correlating big data. It analyzes the similarity between each line of the given data and recognizes the format of the messages

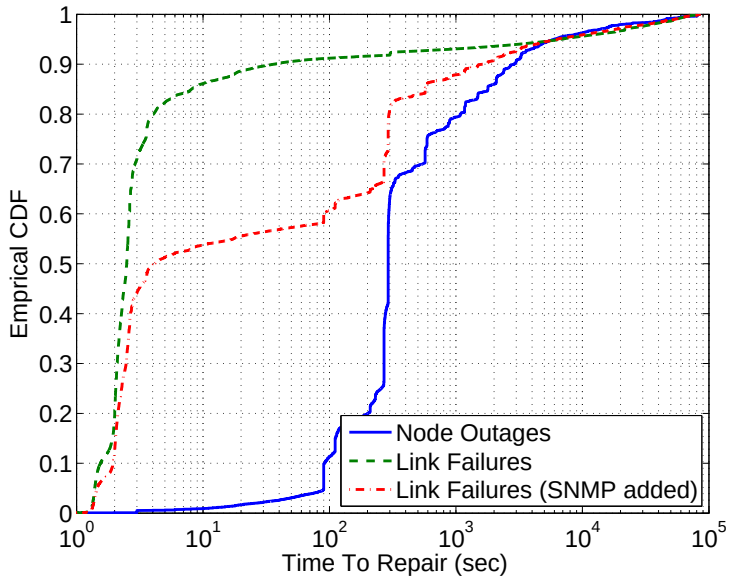


Figure 11: Node outages vs link failures

or anomalies. It is very useful to quickly check various statistics of big data in real-time. Therefore, it enables us to have agile visibility of data and manage systems efficiently.

As the size of the network increases, network operators usually focus on only important links that are uplinks from a switch to other switches in the upper layer. Considering the limited human resources, it's impossible for them to track all the network messages caused by the very end links due to the sheer amount of messages being generated daily. Currently, the issues with individual interfaces are not monitored well nor fixed unless a user contacts the network operators. However, to improve the user experience, we need to harness the syslog messages by providing an automatic tool that analyzes network log messages and detects detrimental network events based on the institutional network policies.

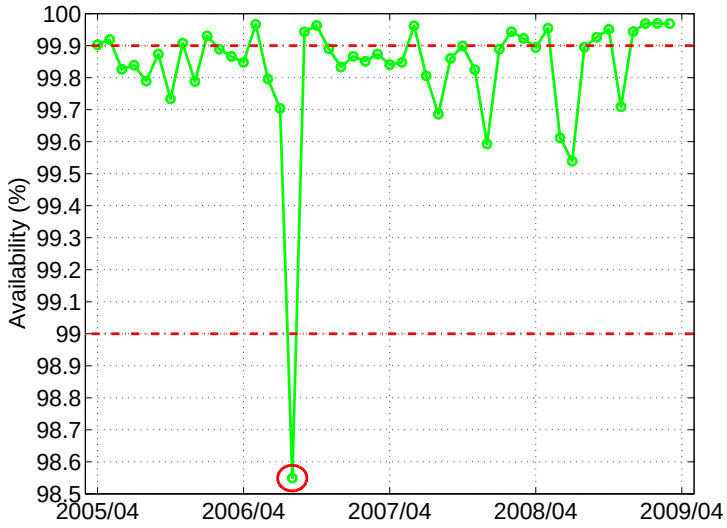


Figure 12: Node availability (SNMP)

In order to quickly identify a network anomalies, we conduct quantitative analysis that ranks the number of node outages and link failures. We use Splunk for this analysis so that we can identify the problematic areas in our campus network taking the spatial and temporal aspects into consideration. For example, as shown in Figure 13, Splunk identifies that our network has many node outages in the "*D*" field of our campus. This is a soccer field, which is a wide-open area. Since no students expect Wi-Fi availability in this area, no complaints have been filed and it was left unfixed. Splunk can also be used to detect a problematic network component. In Figure 14, Splunk indicates that we have many link failures in one of switches in the "*m*" building. The possible reasons could be related to a bad port on the switch, a bad adapter on a client's NIC, or very old cables such as CAT3. In this case, old cables caused these errors. After the new wiring installation, these problems were resolved. Since this type of error only impacts

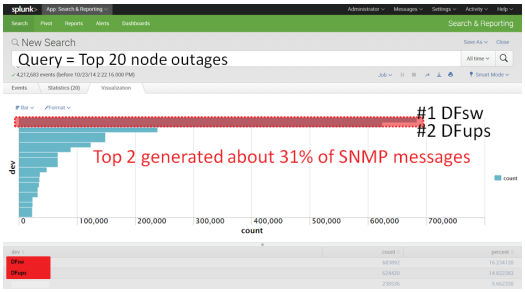


Figure 13: Statistical analysis of node outages using Splunk

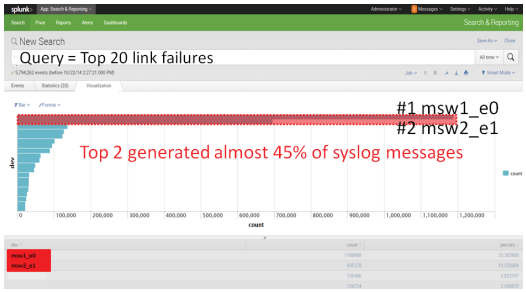


Figure 14: Statistical analysis of link failures using Splunk

individual end devices, it does not receive urgent attention in the current configuration. These network events captured by Splunk are hard to monitor by network operators since these errors don't have a significant impact on the network operation. There is no way for them to get this information unless they carefully look through all the node outage and link failure events. In order to improve the user experience, we need to actively detect these detrimental network events based on the institutional network policies. Network availability analysis with Splunk can help network operators search problematic areas and devices quickly and easily.

3.4 Summary

We conducted analysis of node outage and link failure data on a university campus network (UMKC) in order to understand the general characteristics of an access network including network availability. In order to precisely analyze the characteristics of the campus network, we incorporated vendors' documents in regards to the causes and recommended actions, and the network operators' input on special events and actions as well as long periods of network data such as syslog messages and SNMP data. This study

on the campus network provides insights on the behaviors and conditions of access network availability, and potential end-to-end availability expectations. It also suggests that Splunk can be used as an agile network analysis tool as it helps the network administrators identify weak areas for the overall network reliability improvement.

CHAPTER 4

SDN CONTROL PATH HIGH AVAILABILITY

In this chapter, we present our approaches to current SDN HA problems. We elaborate our proposed *SDN control path HA management framework* that includes several algorithms and describe its implementation. As aforementioned, SDN controller needs to be connected to its underlying network devices and communicate with them to manage flow requests from the network and impose network administrative policies into the network. This introduces new physical links between the controller and network devices. In addition to this, the controller can be configured as a cluster having multiple controllers for network reliability. In this case, there would be a separate network that connects the multiple controllers in the cluster. In this work, the links that connect the controllers in the cluster as well as between the controller and underlying network devices are called *control paths*. We will show various factors that impact the overall SLA of HA for the network services in SDN. Specifically, we will elaborate important practical SDN HA issues and propose simple and effective strategies to the corresponding problems, namely, 1) coupling logical and physical *control path* redundancy, 2) controller cluster structure agnostic virtualization, and 3) fast and accurate failure detection and recovery.

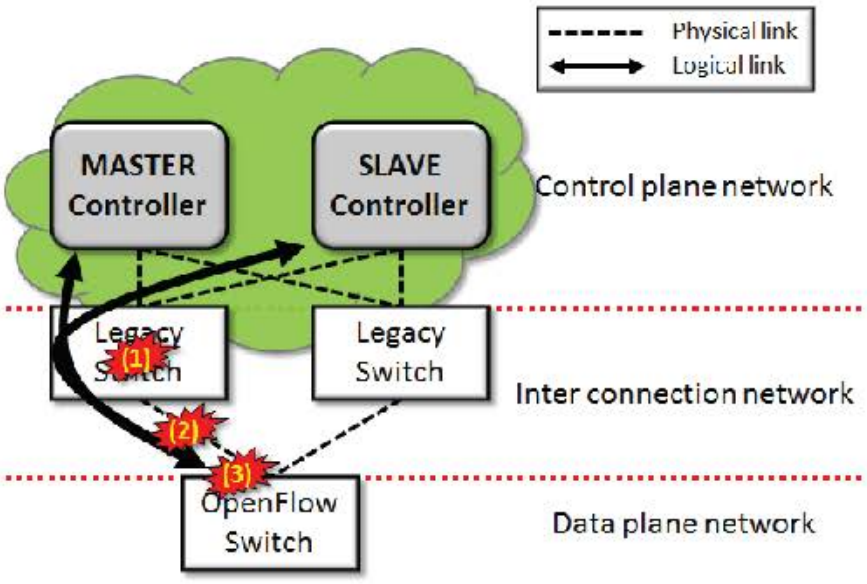


Figure 15: Illustration of unintended single points of failure (see the yellow numbers): Multiple logical connections are overlapped such as (1) the legacy switch between the controller and the OpenFlow switch, (2) the link between the legacy switch and the OpenFlow switch, and (3) the interface of the OpenFlow switch

4.1 Network Management Issues: Network High Availability

In this section, we describe our observations on HA problems in SDN with a focus on the *control path* which has not been considered by the existing research. In order to identify the limitation of the HA feature of the current specification and verify feasibility of our proposed schemes, we set up an Open vSwitch (OpenFlow switch) [62] and two Floodlight [13] controllers running as master and slave as shown in Figure 15. According to the latest specification, the master controller handles all the messages from OpenFlow switches and communicates with the OpenFlow switches. Meanwhile, the slave controller does not receive OpenFlow switch's asynchronous messages such as packet-in and

flow-removed messages except for port-status messages. Thus, the slave controller only recognizes topology changes in the network but doesn't execute any controller-to-switch commands that send packets or modify the state of the OpenFlow switch. The connection between the OpenFlow switch and the two controllers are established through two legacy switches for HA so that we can create redundant network between the controllers and the OpenFlow switch. We use packETH [67] to generate data plane traffic which causes control plane traffic as well. We observe these traffic flows through this work. We discuss three HA problems in detail in this section and our solution strategies are explained in the following section.

Our work is unique, in that we address the HA problems of 1) the *interconnection network* between a controller(s) and OpenFlow switches (i.e., control and data planes) and 2) the network that connects multiple controllers in the cluster. We identify cruciality of *control path* HA and address the important and practical issues of *control path* HA for SDN. We propose multiple effective strategies in order to overcome the *control path* HA issues.

4.1.1 Single Point of Failure of Multiple Logical Connections

As the control plane is the so-called brain of a network, it is vital to provide HA for the controllers in order to maintain continuous connections and fundamental network reliability between controllers and OpenFlow switches for the deployment of an operational SDN. As the essential first step towards HA, a cluster of multiple and networked controllers such as the master and multiple slaves would serve a network rather than a

single controller. As aforementioned, the paths between these controllers and underlying OpenFlow switches are called the *control path*. More precisely, the *control path* in this work also includes the *control plane network* which is the network connection among the controllers in the cluster. Therefore, the *control path* consists of the *control plane network* and *interconnection network* which represents the network domain between the control and data planes. Therefore, a fundamental step towards *control path* HA is to allow an OpenFlow switch to connect with multiple controllers in a cluster simultaneously. However, as shown in Figure 15, we observed that multiple logical connections from an OpenFlow switch to controllers don't fully utilize the physical redundancies and go through the same physical interfaces and links. This causes a logical single point of failure (SPOF) in the *control path*.

According to the latest OpenFlow specification 1.4.0 [64], OpenFlow switches use an IP address and a transport port number to identify a controller in the network when the OpenFlow switches establish a secure communication between them. We find that this simple approach doesn't fully exploit the benefits of the multiple physical paths between controllers and an OpenFlow switch. It also doesn't provide a flexible way to control which *control path* to choose in order to reach the controller. Moreover, this may cause a single point of failure of multiple logical connections even if there are physically multiple paths between the controllers and OpenFlow switches. Figure 15 illustrates the issue of single point of failure despite the existence of physically redundant network components from the OpenFlow switch to the two controllers for HA of the control plane. As seen from the figure, the current OpenFlow configuration policy makes both of the logical

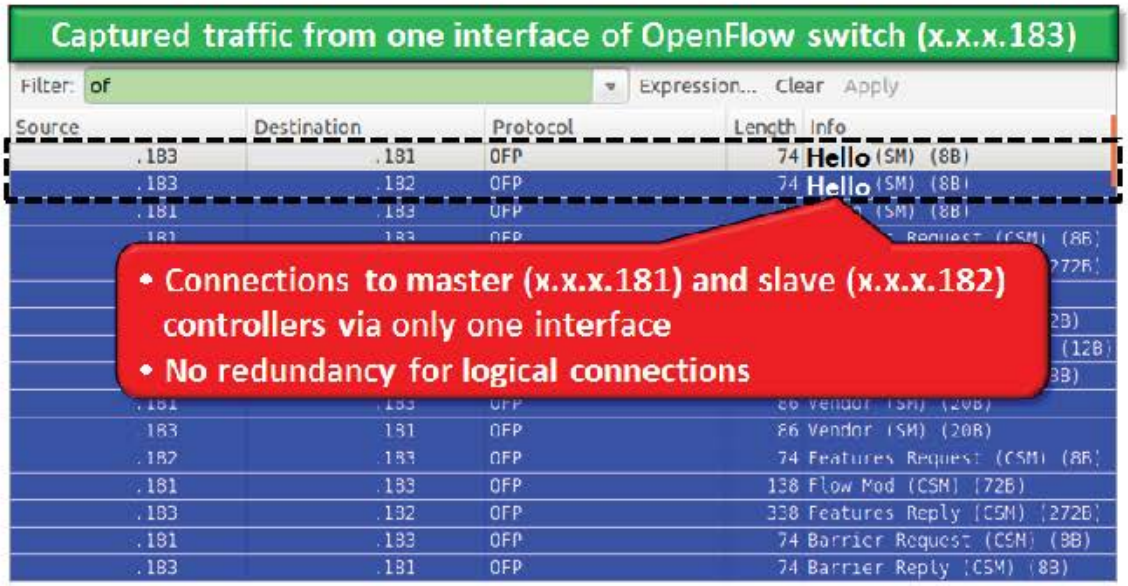


Figure 16: Traffic of one interface of an OpenFlow switch which establishes connections to two controllers shows both logical connections traverse a single physical interface

connections go through the first interface of the OpenFlow switch. Through real experimentation, we have identified and verified that this is indeed a single point of failure of multiple logical connections and could delay the failure recovery time which can degrade HA performance. Figure 16 shows the packets captured from one of the interfaces of the OpenFlow switch connected to multiple controllers. When the OpenFlow switch connects to two controllers, two logical connections use only one interface of the OpenFlow switch. As we can see from the figure, the hello messages surrounded by the dotted box indicate that the logical connections from the OpenFlow switch to two controllers through the same interface.

This configuration works well when we have a failure only on the master controller. The second logical connection to the slave controller will immediately recover communication. However, if we have some failures on the intermediate network components between the OpenFlow switch and the controllers, then both of the logical connections will be lost. Since each logical connection doesn't exploit the physical redundancy, this makes three unintended single points of failure as described in the figure such as (1) the legacy switch between the controller and the OpenFlow switch, (2) the link between the legacy switch and the OpenFlow switch, and (3) the interface of the OpenFlow switch. Since the HA mechanism of SDN is not specifically intended to provide fast switch-over time, the network may experience long recovery time and ultimately service disruption. For example, when one of the components that causes the logical single point of failure fails, an OpenFlow switch needs to find the slave controller through other physical paths. The MAC addresses mapped to the intermediate legacy switches need to be updated or established accordingly in order to re-establish the logical connections. As we may have multiple hops between the OpenFlow switch and the controller, it is difficult to predict the switch-over time. This points out that we need to effectively disperse the logical connections to fully exploit the available physical redundancy, so that HA failover would take place seamlessly without requiring a re-connection process.

Hence, we propose an HA algorithm in order to exploit the physical redundancies, align multiple logical connections along with physically redundant network components on the *control path*, and guarantee a seamless switch-over.

4.1.2 Configuration of Explicit and Distinctive Controller Information

When there are multiple controllers in the controller cluster, it would be desirable to have multiple logical connections from one OpenFlow switch to multiple controllers at the same time so that we can minimize failover progress at the time of failover. However, the current OpenFlow specification requires that when we want to connect an OpenFlow switch to multiple controllers, each controller's information should be explicitly and distinctively configured on the OpenFlow switch. Furthermore the current OpenFlow specification [64] requires additional operations for adding or removing a controller in the controller cluster (e.g., edit-config) [24, 25]. To the best of our knowledge, the currently available SDN HA features [12] do not support an automated configuration for newly added or deleted controllers and OpenFlow switches in the network. Therefore, whenever there are changes in the topology of a controller cluster, a network operator should manually perform the configuration of controllers and/or OpenFlow switches. It is noteworthy that the VRRP (Virtual Router Redundancy Protocol) [41], which is one of the possible protocols that can be used to implement *control path* HA, is an IP level HA solution. Thus, it allows only one logical connection at a time even if there are multiple controllers in the HA domain. The VRRP does't support an OpenFlow switch's preparation of a backup path to slave controllers along with the working path to the master controller.

As the number of network components of SDN such as controllers and OpenFlow switches increases, the number of connections between the controllers and OpenFlow

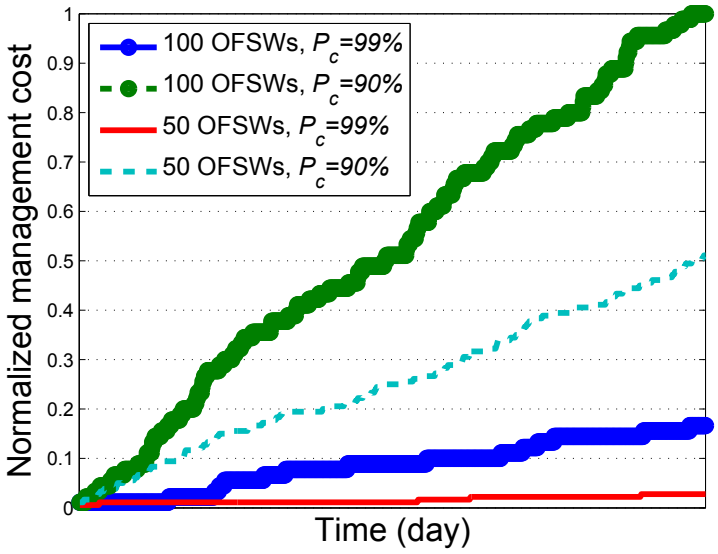


Figure 17: Management cost for the given network size: The management cost increases as the number of OpenFlow switches and the probability of the cluster configuration change increase

switches in the network also becomes larger through either out-of-band or in-band connections. Network administrators have to configure each OpenFlow switch with multiple controllers' individual information. Furthermore, whenever there are configuration changes in the cluster of controllers, they need to reconfigure all the OpenFlow switches managed by the corresponding controller cluster. This would become more time-consuming, tedious, and error-prone, thus increasing operating expenditure (OPEX) due to the augmented complexity of network management.

Figure 17 shows the management cost of the SDN network with the HA feature obtained from the results of the simulation. Let P_c denote the probability that the cluster topology is stable over the course of its running time. Therefore, we can expect more

reconfiguration, when the value of P_c is small. We assume that each topology change of the cluster takes a constant time of the reconfiguration for each OpenFlow switch. We varied the number of OpenFlow switches in the network and P_c. The management cost for the given network increases as the number of OpenFlow switches increases and P_c decreases. To address this problem, we propose a robust and simple management framework that keeps the management cost constant by virtualizing the IP addresses of the controllers. Therefore, the logically centralized controller (i.e., controller cluster) of physically distributed controllers can be identified by OpenFlow switches with only one virtual identity (e.g., virtual IP address). The OPEX of the proposed scheme will not be affected by the size of the network. We will discuss the details of our solution in Section 4.4.

4.1.3 Unrecoverable Interconnection Network Failure

As network systems become complex and convoluted, fast and accurate failure detection becomes more difficult. Fast and accurate failure detection and recovery are indispensable to maintain HA of a network. Note that availability is formally defined as the fraction of time that a system is operational. Thus, there are two approaches in order to increase availability, namely increasing the uptime of a system or reducing the downtime of a system. However, little can be done to increase the uptime, since the commodity systems do fail in practical operations and failures occur often. Meanwhile, we may reduce the downtime by detecting failure early and recovering it as soon as possible. Traditional failure detection and mitigation mechanisms use a heartbeat based scheme. It indicates a

failure when no heartbeat is received from a remote node for a predefined threshold duration. There are a couple of problems with such approaches. It is difficult to identify the exact root cause of the failure since the absence of heartbeats could have originated from possibly various scenarios of a failure(s). The detection time of any failure could be unnecessarily long depending on the configuration of parameters of heartbeat schemes such as a heartbeat interval and the number of heartbeats to conclude a failure. Short heartbeat intervals and a small number of absent heartbeats increase the control signal overhead and may cause inaccurate premature reactions, while reducing the failure detection time. Hence, we focus on how fast and accurate we can detect and recover failures by utilizing additional information such as network topology and link signals in order to overcome the current limitation of the OpenFlow specification and reduce the downtime.

Even if the OpenFlow specification [64] has been evolved to provide better network capability and operability, little about HA of the *control plane network* as well as the *interconnection network* that bridges the control and data planes has been considered. As mentioned earlier, in order to improve HA, we have to consider not only failures of controllers themselves but also the correlation between failures of controllers and OpenFlow switches including their *interconnection network*. This could be critical because in some cases we may lose the connection between an OpenFlow switch and a controller and may not be able to recover the failure by a controller or an OpenFlow switch alone. In the following example, we show that an OpenFlow switch loses its master controller and can not get connected to any other slave controllers.

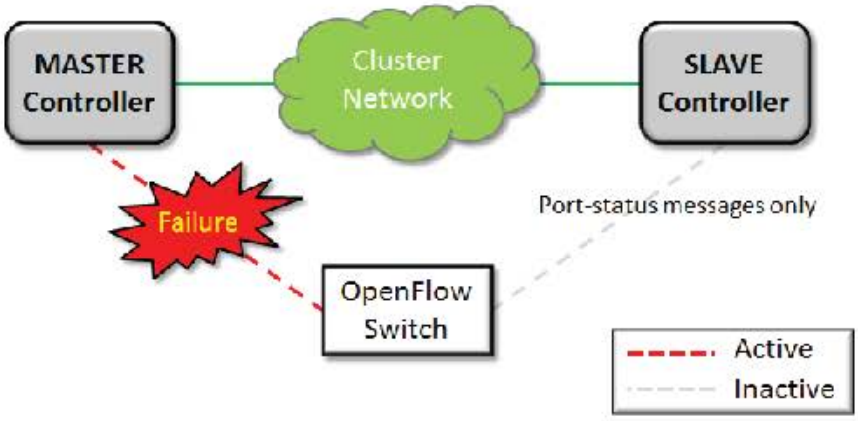


Figure 18: Scenario when an OpenFlow switch loses its master controller: The connection between the slave controller and the OpenFlow switch transfers only port-status messages

In Figure 18, we have two controllers such as the master and slave, and the OpenFlow switch that has established connections to both of the controllers. The controllers are connected through a legacy switch for the purpose of the operational synchronization among controllers in the controller cluster. According to the OpenFlow specification, the slave controller cannot manage the OpenFlow switch. Therefore the connection between the slave controller and the OpenFlow switch is initially inactive. In addition, another important issue of the OpenFlow specification is that only a slave controller initiates the role change request when the slave controller detects that its master controller is down. The failure detection of the master controller is done by periodic heartbeat messages from the master controller to the slave controller. Typically, if a slave controller doesn't receive the heartbeats messages three consecutive times, then the slave controller initiates the procedure to become the master controller.

Suppose that we have a link failure that causes complete disconnection between

the master controller and the OpenFlow switch as depicted in Figure 18. Since the OpenFlow switch lost its master controller, it has to find a slave controller. However, since the master controller and the slave controller can communicates with each other without any error, the slave controller will not change its role from the slave to the master even if the OpenFlow switch lost its connection with the master controller. Note that the current specification doesn't allow an OpenFlow switch to initiate its controller's role change. This situation exposes the possibility that some part of the network could be non-operable due to the disconnection between the controller and OpenFlow switches and may have a cascade effect on the network and lead to performance degradation. This is mainly because the current HA feature doesn't fully consider the correlation between failures of the *control plane network* and *interconnection network*. Therefore, we design an HA algorithm that exploits additional information such as network topology and link signals along with the heartbeat messages in order to detect failures in a fast and accurate manner. Our main contribution of this approach is to sophisticatedly integrate various failure detection procedures through all the HA network domains including the control, data, and *control path* domains.

4.2 SDN Control Path HA Management Framework: Overview

In the following three sections, we propose and discuss our strategies to improve the overall performance and manageability of HA. We have implemented the proposed algorithms and modified the OpenFlow reference implementation on our real network testbed, which consists of one OpenFlow switch, two legacy switches, and two SDN

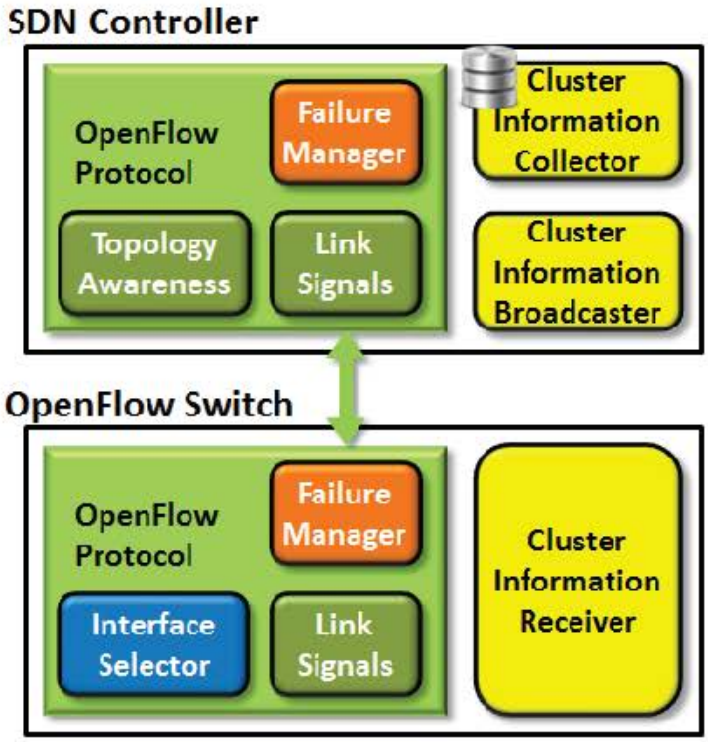


Figure 19: System architecture with the three HA components

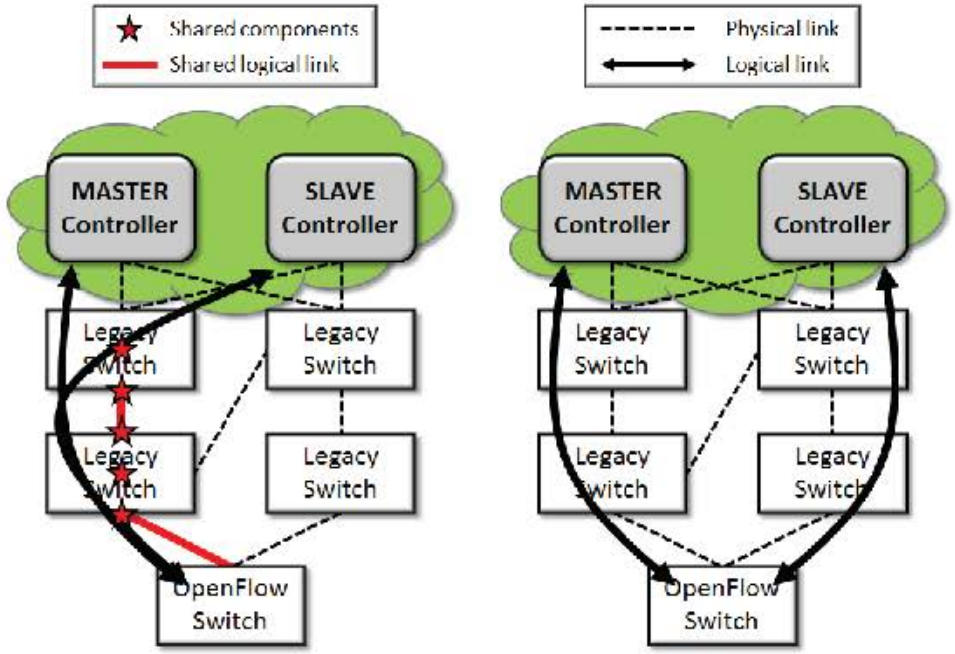
controllers. Figure 19 describes our proposed system architecture with the three HA approaches. We also illustrate operational examples for each approach in the following sections.

4.3 Coupling Logical and Physical Control Path Redundancy: Approach

In order to fully exploit the physical redundancy of the network and alleviate/remove the single points of failure, we have improved the OpenFlow reference implementation by enabling an interface selection feature. As described in Section 4.1.1, the logical connections do not exploit physical network redundancy. This causes multiple single points

of failure of the logical connections. This is mainly due to the current OpenFlow specification which identifies a controller only with an IP address and a transport port number. We have added an L2 MAC address to enable the selection of interfaces on the OpenFlow switch along with an IP address and a transport port number. After the modification, it becomes possible to separate the overlapped logical connections into the two separate logical connections aligned through two separate physical paths. The proposed approach can balance the number of logical connections through multiple interfaces on an OpenFlow switch. As illustrated in Figure 20, a little bit of improvement provides better flexibility and makes it possible to separate the overlapped logical connections into two separate physical paths. *Interface Selector* in Figure 19 automatically scans available interfaces that are connected to a controller and distributes multiple logical connections evenly. For simplicity of the comparison, Table 8 shows the difference between the existing OpenFlow specification and the proposed approach. Figure 21 shows two logical connections are separately deployed through the diversity of the physical network. Different source MAC addresses (in the black dashed boxes) for the two logical connections indicate that they are assigned through the multiple interfaces of the OpenFlow switch. In this case, the logical connection to the master controller is assigned to the first interface of the OpenFlow switch and the other logical connection to the slave controller is assigned through the second interface of the OpenFlow switch.

The proposed approach can effectively be applied to and be even more beneficial in the case of the network configuration with an in-band controller. This is because the degree of single points of failure increases as the hop count increases from one of the



(a) (b)

Figure 20: Illustration of D_{spof}: (a) Overlapped multiple logical connections, D_{spof} = 7 (b) Elimination of unintended single point of failure by aligning separate logical connections via redundant physical network with our approach, D_{spof} = 0

OpenFlow switches to the controller. The term, D_{spof}, is used to represent the number of SPOF that is possessed by a single logical connection. D_{spof} increases when logical connections share (1) an intermediate switch, (2) a link between switches, and (3) an interface of a switch. The higher D_{spof} is, the higher probability it takes longer time to recover the connection to the controller. Figures 20(a) and 20(b) further explain the benefit of the proposed approach by comparing operational scenarios when our solution is applied or not. As we can see, D_{spof} of the current logical connections is 7 while the proposed connection shows D_{spof} is 0. Our solution can effectively reduce D_{spof} by

Table 8: Difference between the existing OpenFlow configuration and the proposed OpenFlow configuration

	Parameters used for the connection to controller
Existing	IP address, transport port number
Proposed	**MAC address**, IP address, transport port number

ensuring that logical paths align through physical redundancy. Figure 22 shows that the availability of logical connections with/without *Interface Selector*. We assume that each component that can cause a single point of failure between the OpenFlow switch and the controller has the availability 99%. Availability 100% means no downtime or no error. As we can see from Figure 22, availability of the logical connections with *Interface Selector* keeps a constant value. On the other hand, availability of the logical connections without *Interface Selector* is decreased as D_{spof} increases.

4.4 Controller Cluster Structure Agnostic Virtualization: Approach

In this approach, we propose a virtualization technique to make the physically distributed multiple controllers into one logically centralized controller that can be identified by an OpenFlow switch via one virtual IP address. It also automates the configuration process, which originally was a time-consuming and error-prone manual process, for every change of the controller cluster. Thus, each OpenFlow switch doesn't have to know about the distinct IP addresses or port numbers of the controllers or dynamics of the controller cluster regardless of how large the controller cluster is. By having the virtualized information for the controller cluster, we can reduce a significant amount of time for the

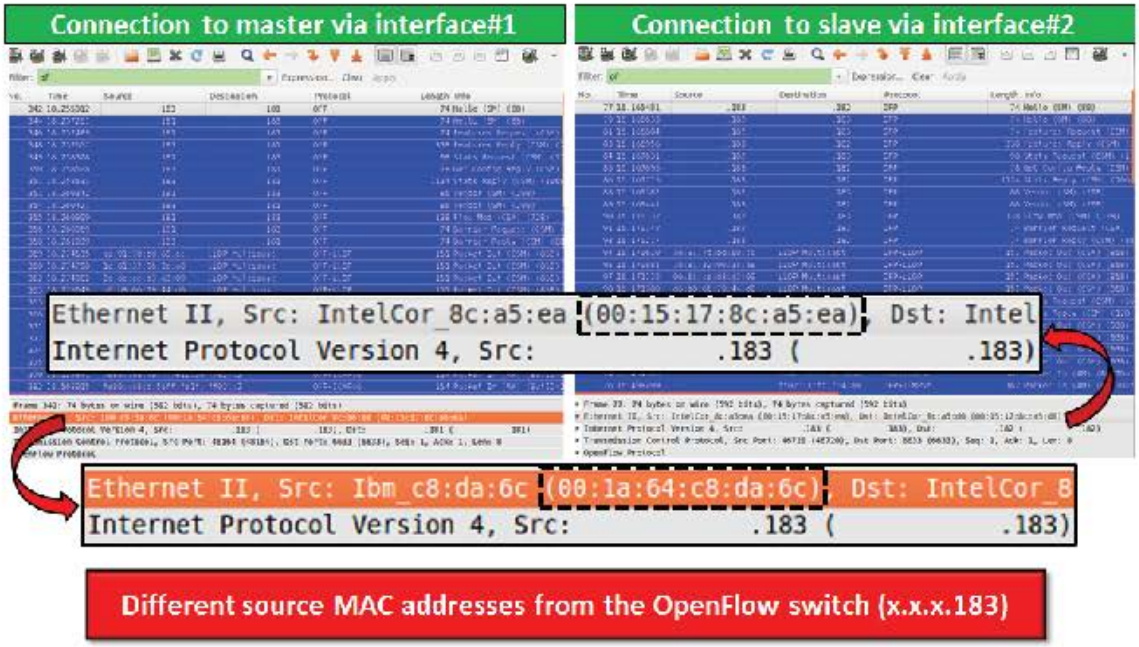


Figure 21: Logical connections deployed separately through different interfaces by exploiting the diversity of the physical network

configuration and keep a constant time consumption for OpenFlow switch configuration.

Here we explain how we keep only one virtual IP address to maintain the identification for the cluster of multiple controllers in Algorithm 1. Virtualization that uses only 'one' virtual IP address to maintain the identification for the controller cluster can be done with an approach such as the VRRP. The first step of the proposed virtualization is similar to the VRRP. An assigned virtual IP will be associated with only one controller in the cluster at any given time. Since the virtual IP would be associated with only one controller in the cluster at any given time, an OpenFlow switch, which is just added in the network, can connect to the associated controller with only one virtual IP address to

Algorithm 1 Cluster Virtualization Algorithm – runs on each controller in the cluster

```
1:  if this.role == NULL then
2:      while this.role == NULL do
3:          multicast this.priority;
4:          check CIBroadcaster.priority;
5:          if no CIBroadcaster || this.priority > CIBroadcaster.priority then
6:              this.role = CIBroadcaster;
7:              this.IP = virtual IP;
8:          else
9:              this.role = backup;
10:         end if
11:     end while
12: else if this.role == CIBroadcaster then
13:     while this.role == CIBroadcaster do
14:         multicast heartbeat packets periodically;
15:         listen to new.priority;
16:         if received new.priority then
17:             send this.priority;
18:             if this.priority < new.priority then
19:                 this.role = backup;
20:             end if
21:         end if
22:     end while
23: else if this.role == backup then
24:     listen to heartbeat packets;
25:     if no heartbeat packets then
26:         this.role = NULL;
27:     end if
28: end if
```

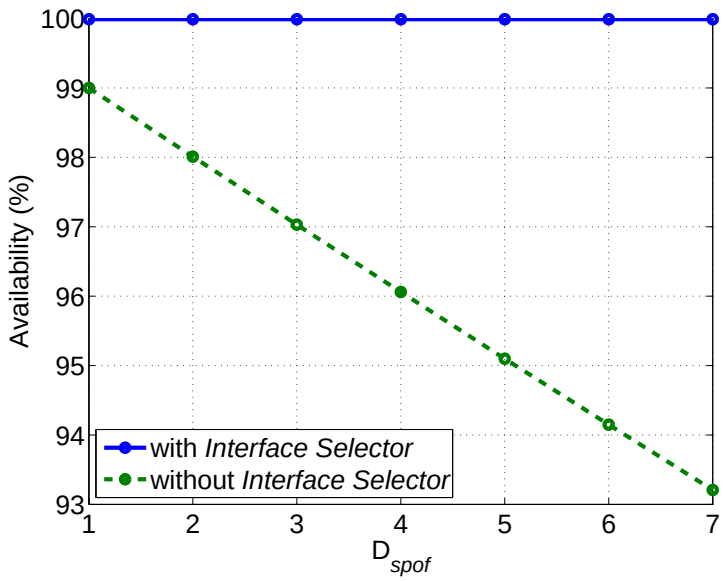


Figure 22: Availability of logical connections with/without *Interface Selector*

get the cluster information. Therefore, we can hide the structure of the controller cluster behind of the virtual IP address. There is a module in a controller that can collect other controllers' information in the cluster ('Cluster Information Collector' in Figure 19). An OpenFlow switch uses the one virtual IP address and can communicate with the associated controller to obtain the rest of the cluster information. The controller associated with the virtual IP address would be the cluster information broadcaster ($CIBroadcaster$). The $CIBroadcaster$ will send the up-to-date cluster information to the newly connected OpenFlow switches in the network. Other controllers will remain as backup broadcasters and listen to the heartbeat messages from the $CIBroadcaster$. A new $CIBroadcaster$ selection will occur when the current $CIBroadcaster$ is down or a new controller added to the cluster has the highest priority. This election process has nothing to do with the

decision of the master or the slave controller of the cluster. This means even one of the slave controllers can be the $CIBroadcaster$.

Algorithm 2 Cluster Information Consistency Algorithm – runs on each controller in the cluster

```
1:  if just added to the cluster and booted up then
2:     repeat
3:        createHelloMessage();
4:        multicast hello message to the cluster;
5:        wait for responses;
6:     until number of responses == number of controllers in the cluster;
7:     save the number of responses and current information it received;
8:  else if configuration changed then
9:     sendUpdateInfo();
10:    // send updated cluster information to all the OpenFlow switches in the network
11:    repeat
12:       createUpdateMessage();
13:       multicast update message to the cluster;
14:       wait for responses;
15:    until number of responses == number of controllers in the cluster;
16: else if received hello message then
17:    this.sendIPaddress();
18:    this.sendPortnum();
19:    // send its IP address and port number
20: else if received update message then
21:    this.updateDB();
22:    // update its cluster information database
23: else if received a request for cluster information then
24:    this.sendClusterInfo();
25:    // send cluster information to requesting OpenFlow switch
26: end if
```

We have explained how an OpenFlow switch can get all the controllers' information in the cluster using only one virtual IP address. Now, we describe how the cluster keeps consistency of cluster information among the controllers and with the OpenFlow switches. Algorithm 2 illustrates how the logically centralized controller, which is the

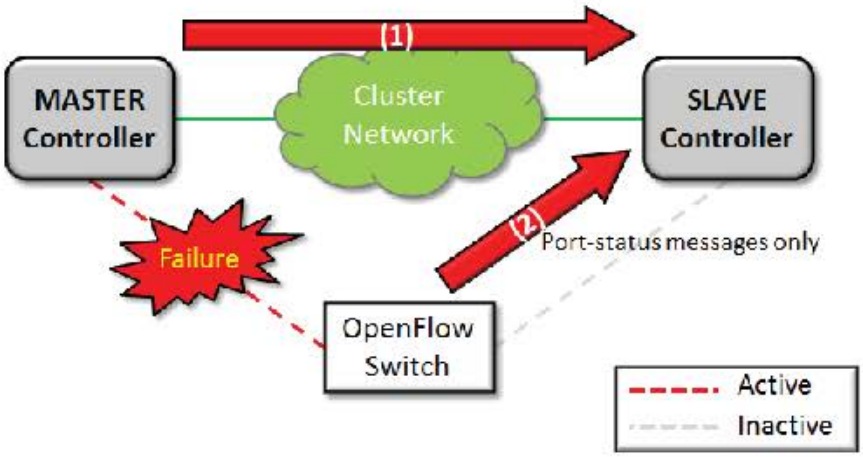


Figure 23: Fast and accurate failure detection and recovery using topology awareness and link signals: (1) The master controller initiates the recovery (Algorithm 3) (2) The OpenFlow switch initiates the recovery (Algorithm 4)

cluster of multiple controllers, keeps consistency of cluster information. There are two types of messages to maintain consistency of cluster information. The first one is the 'hello' message. It is sent by the controller, which has just been added into the cluster. And then the controller that just sent a 'hello' message tries to collect the current cluster information from other existing controllers in the cluster. The second message is an 'update' message. If there is any change on the controller's configuration, then it sends the updated information to the OpenFlow switches and the other controllers in the cluster. Therefore, whenever there are changes to the configuration in the cluster, the consistency of the cluster information is automatically achieved and no human-intervention needs to be involved.

Table 9: Recovery time of the proposed schemes initiated by an OpenFlow switch or a controller

Methods	Recovery time (RTTs)
Existing specification	∞
Approach initiated by controller	4 RTTs
Approach initiated by OpenFlow switch	1 RTT

4.5 Fast and Accurate Failure Detection and Recovery: Approach

In this section, we discuss how to promptly detect *control path* failures. As we discussed in Section 4.1.3, fast and accurate failure detection is very important in order to maintain HA especially for reliable operation of controllers. We design an algorithm that exploits additional information such as network topology and link signals along with the heart-beat messages to detect failures in a fast and accurate manner. Note that our main contribution to this approach is to orchestrate the failure detection procedures of all the HA network domains that include the controller cluster network and the network connection between the controllers and OpenFlow switches. According to the most recent SDN protocol specifications (such as OpenFlow and OpenDaylight), there is no orchestration for the failure detections and recoveries of all the separate HA network domains. Hence, our proposed protocol will be the first algorithm exploiting the network topology and the low level link signal along with the heart-beat messages. The algorithm not only detects failures quickly, but it also proposes another potential enhancement. Ample information from various sources may be able to overlap failure detection and recovery windows. Algorithms 3 and 4 describe the simple yet effective idea to expedite failure detection and

Algorithm 3 Detection Algorithm – runs on the master controller in the cluster

```
1: downlink signal is detected by the master controller;
2: check connectivity with the OpenFlow switch;
3: if connection is okay then
4:     alert an operator for a repair physical repair of the link;
5: else
6:     check available slave controllers and their reachability to the OpenFlow switch;
7:     inform an appropriate slave controller to become the master;
8: end if
```

Algorithm 4 Detection Algorithm – runs on an OpenFlow switch

```
1: downlink signal is detected by an OpenFlow switch;
2: if link failure to its master controller then
3:     inform (a) slave controller(s) if reachable;
4: else
5:     report to the master controller about the link failure for flow table update;
6: end if
```

recovery that runs on the controllers and switches, respectively.

Figure 23 gives a simple illustration of the logical flows of the algorithms. Note that the slave controller keeps receiving port-status messages from the OpenFlow switches and has up-to-date information of connectivity (i.e., topology) of the underlying OpenFlow switches. We propose two different solutions. In the first approach, the master controller notifies the slave controller about the disconnection to one of the OpenFlow switches (Alg. 3). If the slave controller has a connection to the OpenFlow switch that has lost its connection to the master controller, then it will notify the original master controller to change the role from the master to the slave. And then it changes its role from the slave to the master so that it becomes the new master controller and serves the OpenFlow switch that previously has lost its original master controller. We assume that a master election mechanism can be applied when there are multiple slave controllers in the cluster

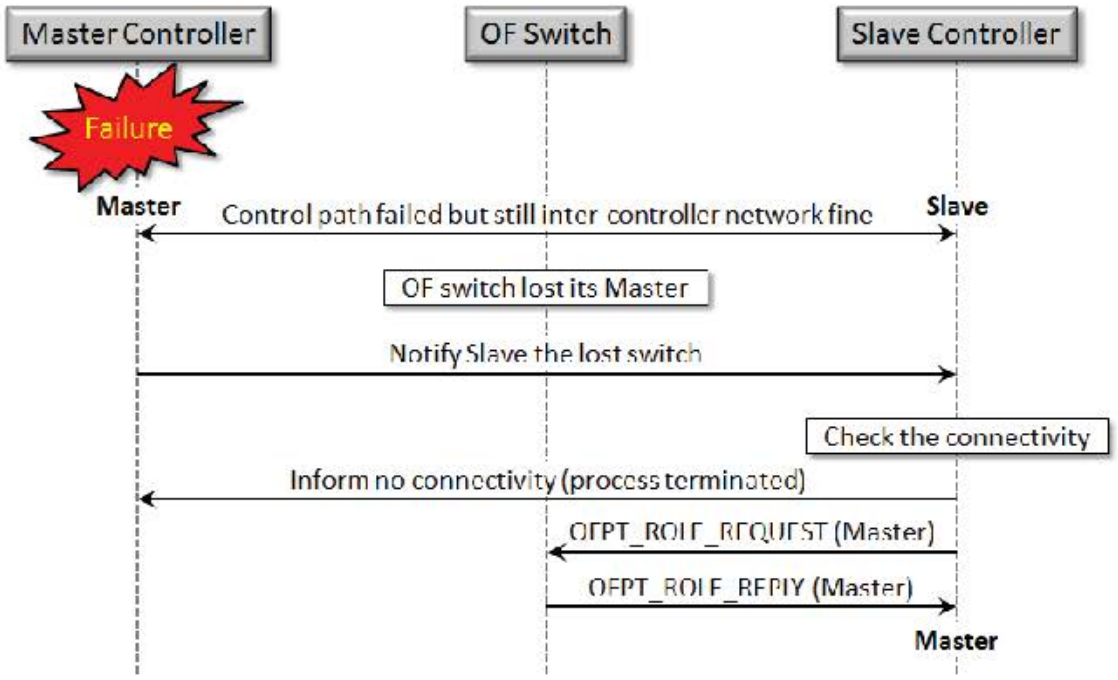


Figure 24: Initiated by the master controller (Algorithm 3)

as proposed in [24, 25]. The second approach (Alg. 4) allows that the OpenFlow switch notifies the slave controller about its disconnection to the original master controller. Once the slave controllers receive this notification from the OpenFlow switch, it will change its role to the master. If there are multiple slave controllers in the controller cluster, we need to run the master election process. After a new master controller is decided, the original master controller will change its role to slave according to the notification from the new master controller. Detailed procedures of Alg. 3 and 4 that show the interactions among the master/slave controllers and the OpenFlow switch are illustrated in Figures 24 and 25, respectively.

Table 9 and Figure 26 show the performance of the two proposed algorithms. The

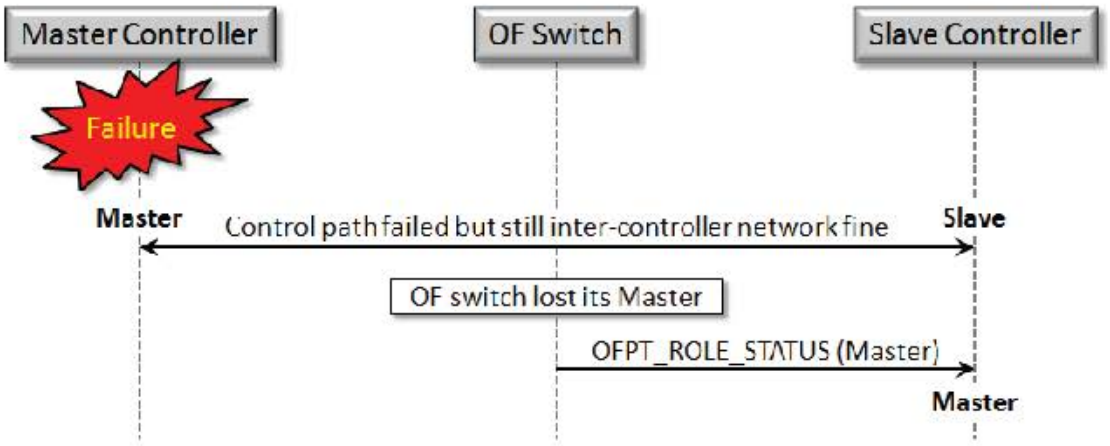


Figure 25: Initiated by an OpenFlow switch (Algorithm 4)

one running on the OpenFlow switch has a definitely faster response time than the one on the controller due to its simpler procedure as seen in Table 9. The modification on the OpenFlow switch is reasonably simple and easily adapted to the current specification (OpenFlow 1.4). Figure 26 shows that the recovery time linearly increases as the network size increases. The network size is measured by the maximum hop counts between the controller and the OpenFlow switch.

4.6 Experiment and Emulation Setup

Here we describe our experimental testbed and emulation setups. We then verify the functionalities of our proposed schemes in the physical testbed. As illustrated in Figure 30, three servers and two legacy switches are used to configure the experimental testbed. One of the servers is running either Open vSwitch [62] or Mininet [58] so that it can play the role of the OpenFlow switch or can emulate a large network topology along

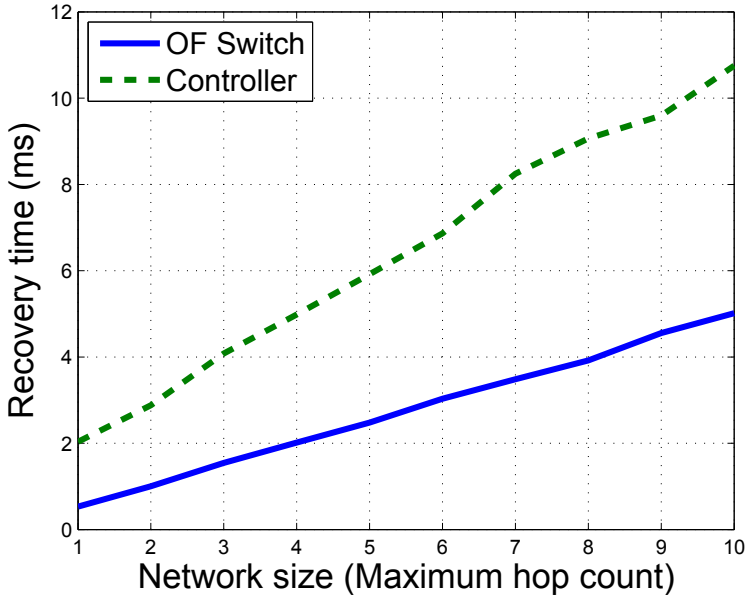


Figure 26: Comparison of recovery schemes initiated by an OpenFlow switch and a controller

with hosts. Note that each emulated OpenFlow switch created by Mininet is directly connected to the controllers through Server #3's physical interfaces *eth0* and *eth1*. The remaining two servers are used as an SDN controller. All the servers are running Ubuntu 12.04 LTS, which is the most compatible operating system with various SDN controllers and other software packages, and the two servers for the SDN controller specially run the software packages such as *keepalived* (VRRP) [46] and *bonding driver* [2] for the HA configuration.

4.7 SDN Control Path HA Framework Implementation

Our HA framework consists of two parts: the modules in the SDN controller and those in the OpenFlow switch. For the implementation of the SDN controller part, we mainly use the Floodlight controller [13] which is one of the well-known SDN controllers and written in Java. For the implementation of the OpenFlow switch part, we use RESTful (Representation State Transfer) API to communicate between SDN controllers and OpenFlow switches. In addition, since we need a direct modification of the OpenFlow switch for our first HA solution, we also use Open vSwitch [62], which is the OpenFlow reference implementation, to implement our first HA solution that can select the multiple interfaces for its logical connections to the multiple SDN controllers. The Open vSwitch is an open source virtual switch licensed under the Apache 2.0 license. It supports standard management interfaces and protocols as well as the OpenFlow protocol.

Figure 28 shows the implementation details of the proposed solutions in the SDN controller and the OpenFlow switch. We elaborate the implementations of our proposed solutions with the first HA solution. Our objective of the first HA solution is to give the OpenFlow switch a capability of exploiting and selecting an interface for its logical connections to the SDN controller. We extensively analyze the source codes of the Open vSwitch and identify the important parts of the source codes to implement the interface selection module. Browsing the source codes of the Open vSwitch shows many directories for the functional and administrative implementation such as *ofproto*, *ovsdb*, *datapath*, *utilities*, *rhel*, *vswitchd*, and *lib*. As shown in Figure 27, *vswitchd* is the main

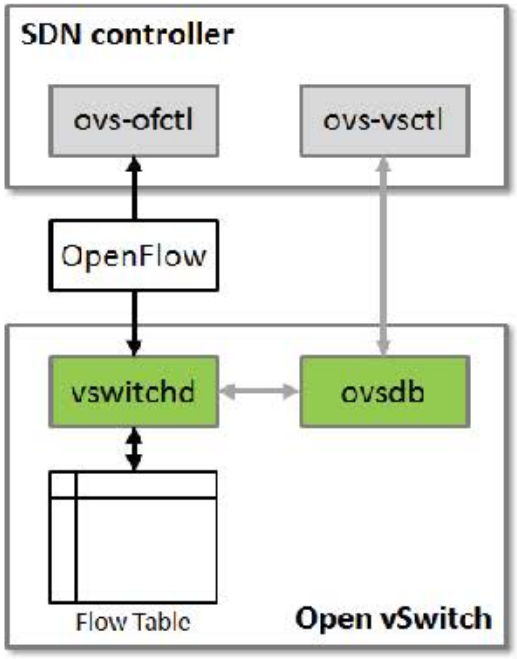


Figure 27: Simplified Open vSwitch architecture

module that communicates with the SDN controller for the packet processing and management and uses many library functions defined in the *lib* directory. The *ovs-ofctl* command is used for administering and monitoring OpenFlow switches. We can query the OpenFlow switches to add, retrieve, and delete flow entries to/from the flow table of the OpenFlow switches. The *ovs-vsctl* command is mainly used for administering OpenFlow switches' connections to the SDN controllers. It communicates with the Open vSwitch database (ovsdb) [66] running in the OpenFlow switches. *ovsdb* stores Open vSwitch configuration information and we can query the *ovsdb* to see the current configuration and apply any configuration changes using the *ovs-vsctl* command. As we mentioned earlier, each connection from the OpenFlow switch to the SDN controller is

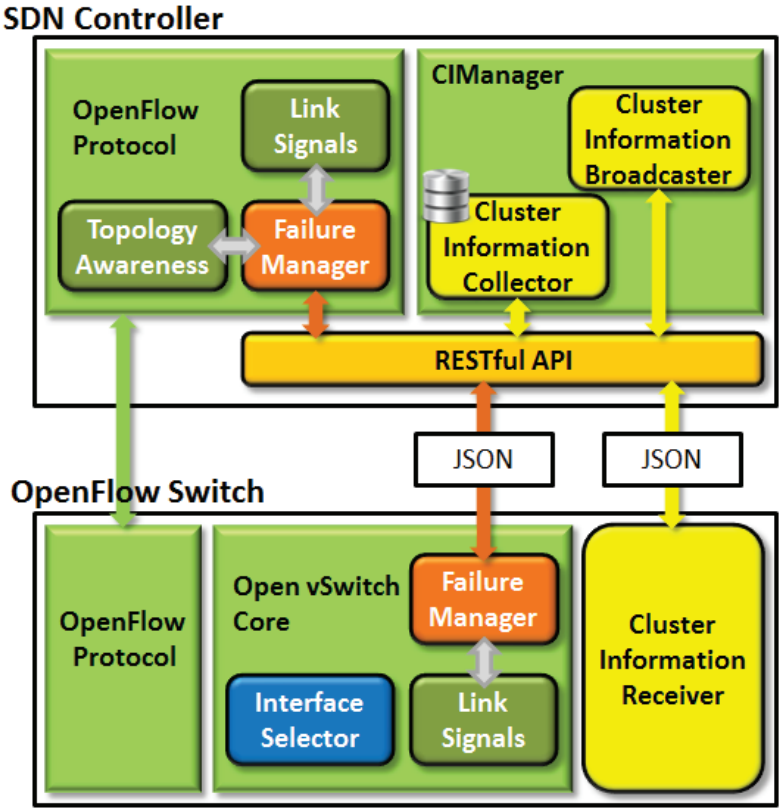


Figure 28: SDN control path HA framework implementation

a TCP connection. Therefore, the best place in the Open vSwitch to locate our modification would be the part that establishes a new TCP connection. We can find the source code from ∼/openvswitch/lib/stream-tcp.c. The *new_tcp_stream* function is the one we modify to add our proposed solution, *interface selector*. In order to control physical network components (e.g., interfaces), we can use the *ioctl* function. The *ioctl* function with the *SIOCGIFINDEX* flag returns the interface index of the interface. Currently, the Open vSwitch uses only one interface by default to establish multiple connections to the

Table 10: RESTful API URIs for the second HA solution

URI	Method	Description
/wm/cluster/configuration	GET	Get the current configuration information from the configuration information manager (*CIManager*)
/wm/cluster/configuration	POST	Send its configuration information to the *CIManager*
/wm/cluster/configuration	DELETE	Delete its configuration information from the *CIManager*
/wm/cluster/clear	GET	Clear all the configuration information from the *CIManager*

SDN controllers. This hinders the logical connections from exploiting the physical redundancies between the OpenFlow switch and the SDN controllers. In order to solve this problem, we modify and expand the function so that the OpenFlow switch can exploit the physical redundancies between the OpenFlow switch and the SDN controllers when it established logical connections.

For our second solution, we mainly use a scripting language with the *curl* command [21] and RESTful APIs to exchange extra messages between the OpenFlow switch and the SDN controllers as well as between the SDN controllers. We implement a Floodlight module [27] that exchanges and stores the configuration information of the SDN controllers in the cluster. This module can be accessed by the OpenFlow switches and SDN controllers by the URIs shown in Table 10. For example, the "/wm/cluster/configure" path can be used by the OpenFlow switches and SDN controller. A series of the scripting codes are implemented to trigger the execution of those RESTful APIs in order for the SDN controller to send its configuration information to the *CIManager* as well as to

```
{
   "controller1":"<IP address of controller1>:<Port number of controller1>",
   "controller2":"<IP address of controller2>:<Port number of controller2>",

        • • •

   "controllerN":"<IP address of controllerN>:<Port number of controllerN>"
}
```

Figure 29: Configuration information in the JSON format

Table 11: RESTful API URIs for the third HA solution

URI	Method	Description
/wm/core/switch/all/role/json	GET	Retrieve the roles of all the presently connected switches
/wm/core/switch/all/role/json	POST	Set the role of all the presently connected switches
/wm/core/switch/<switchID>/role/json	GET	Retrieve the role of a particular connected switch
/wm/core/switch/<switchID>/role/json	POST	Set the role of a particular connected switch

update or delete its configuration information. There is also a scripting code that can be used by the OpenFlow switches to retrieve the up to date configuration information from the *CIManager*. The up to date configuration information is delivered in the JSON format [43] as shown in Figure 29. The scripting codes use the C++ JSON parser [44] to parse the configuration information in the JSON format.

Lastly, we also use a scripting language with the *curl* command [21] and RESTful APIs to exchange extra messages between the OpenFlow switch and the SDN controllers

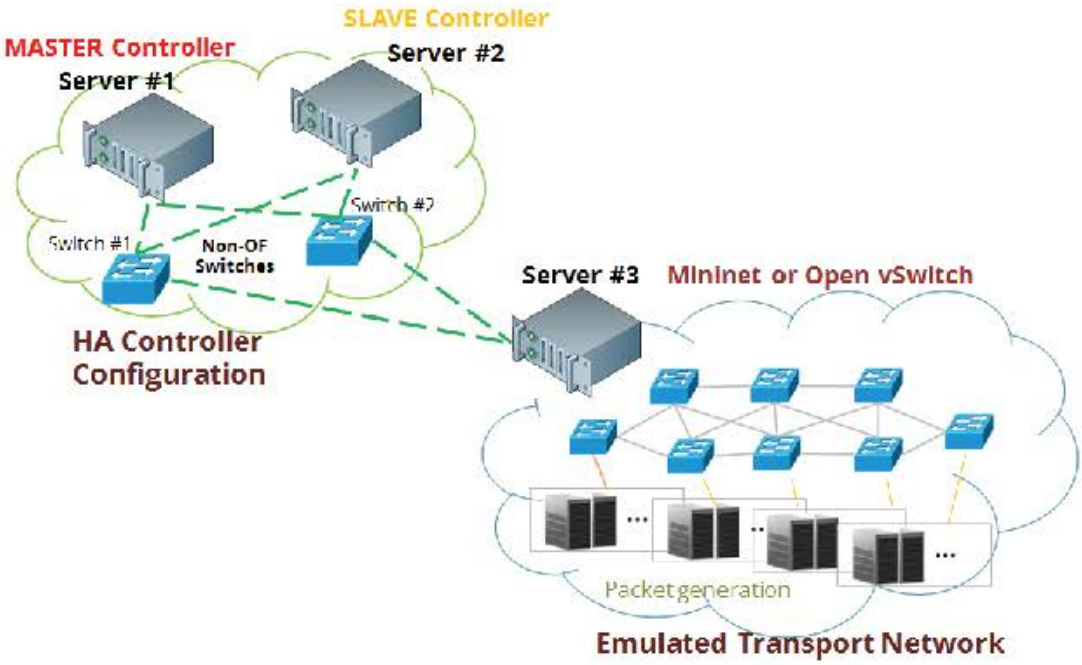


Figure 30: High availability experimental testbed and emulation setups

as well as between the SDN controllers for the third HA solution. The Floodlight controller already supports various URIs related to controlling the roles of the controllers as listed in Table 11 [28]. We utilize these URIs and expand their correlated functions to sophisticatedly handle unexpected disconnection from the SDN controllers the HA mechanism.

4.8 SDN Controller Software Availability Validation: New Approach

In this section, as an extension of the SDN high availability project, we approach SDN high availability problems from a different perspective. We propose to build a new

SDN controller software validation and optimization framework that serves as a fundamental approach to improve network high availability and scalability for SDN. As we will see in the following section, each SDN controller generates various types of control messages to administer the underlying network. Even though the implementation of each SDN controller follows the format of each control message as defined in the OpenFlow specification, the number of each control message used for coping with some network events varies depending on SDN controllers. The proposed solution verifies validity of the implementation of an SDN controller by analyzing the types and number of control messages generated by the SDN controller. It then reduces the detrimental impact of an overloaded system by incorporating vitality of individual control messages. We believe that the proposed validation and optimization facility services a fundamental approach to improve SDN high availability and scalability as the operations of the SDN controllers can be optimized. Detailed objectives are as follows:

- The system reduces control message processing overhead for the controller by facilitating a selective message processing mechanism. It classifies the received messages to identify the essential messages to be processed.

- The system increases scalability of the controller by reducing the spatiotemporal control message concentration towards the controller. It facilitates the registration mechanisms by requesting the levels and schedules of the control messages.

- The system expedites the response time against the urgent issues by delegating actions to the immediate controllers, servers, or switches. The delegated system performs a resolution first and reports to the controller later according to the requested delegation level.

- The system enhances the root cause analysis capability of the controller by providing intelligences related to the classified and prioritized control messages. It correlates the control messages with incoming traffic patterns and relationships among objects.

- The system saves network bandwidth by reducing the amount of control message traffic. It creates fewer control messages according to the controllers registration and delegation requests.

4.8.1 Problem Definitions and Motivation

SDN exposes control messages from an internal device to the communication networks between the controllers and the forwarding devices. As studied in [9, 45, 73], SDN imposes excessive control traffic overhead in order for the controller to acquire global network visibility. More significantly, the overhead will be further increased as many existing SDN controller platforms allow a variety of heterogeneous application interfaces and protocols to the data plane. The overhead will be worsened if the control plane uses an in-band network that shares the same physical network paths with the data plane. If the overhead is not controlled properly in a complex combination of multiple and heterogeneous management channels, they cause various scalability problems for the networking

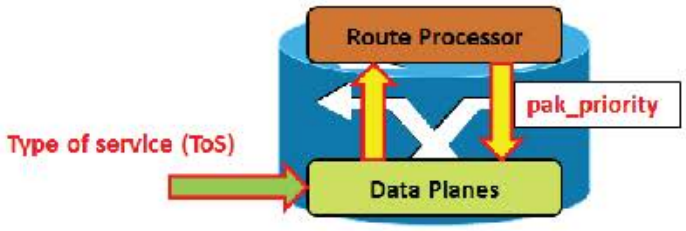


Figure 31: Cisco's PAK-Priority

devices, controllers, and the network itself including slow message processing, potential message drops, delayed root cause analysis, and late responses to urgent problems. Not only logical control centralization, but also virtualization of the underlying computing and network resources including Layer 4 (transport) $\sim$ Layer 7 (application) capabilities adds demands for more flexible and programmable monitoring functions per virtual domain. Moreover, in the practical network operation point of view, SDN deployment may take a gradual or partial transition that will result in more complex heterogeneous management environment.

In a traditional router, there is a classification facility such as Cisco's PAK-Priority [76] for the internally generated packets (i.e., setting a value on the ToS field). As illustrated in Figure 31, packets punted between the control plane and the data plane within a router are classified and handled as "Important" packets and "unimportant" packets according to the predefined ToS values. However, the remote SDN controllers do not support any packet prioritization and classification facility. According to the most recent OpenFlow specification, the SDN controllers drop packets randomly regardless of the importance and urgency of the packets. The detrimental impact of dropping vital packets can be even

worse, if the network and the controllers are in competition with other protocol and application traffics. SDN uses asynchronous control messages to convey data plane state to the centralized controller. For example, in the OpenFlow protocol, switches send asynchronous control messages including a packet-in, flow or port state change, and error to the controller. An asynchronous configuration option that enables the controller to set or unset the asynchronous control messages except error messages is also available. For example, it can disable all the port state change messages from a switch. Although control message overhead can be reduced by disabling some control messages, potentially important information from the switch can be also eliminated. Since some control packets are vital to basic functioning of SDN and should not be subject to random dropping by the controller, a prioritization and classification facility is needed in SDN.

4.8.2 MCVO Implementation

We facilitate a comprehensive classification and prioritization system for creating, handling, and managing the network control messages in SDN. We believe that the facility services a fundamental approach to improve SDN high availability and scalability as the controllers and interconnection networks can drop less impactful and non-vital packets when the resource becomes limited. The MCVO facility is designed to provide a control message classification, verification, and optimization for SDN. As illustrated in Figure 32, the system includes a prioritization and classification, a type, quality, and schedule registration mechanism, and the resolution and delegation protocols. The system also provides mechanisms to correlate the control messages with other intelligence to

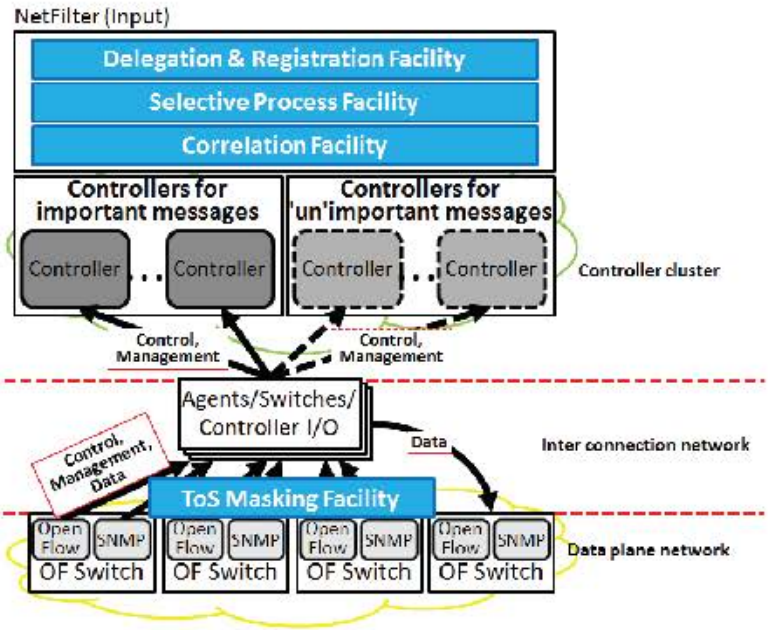


Figure 32: MCVO system architecture

expedite a decision process.

We perform an initial overhead analysis of control messages to understand the impact of dropping packets both with and without the priority-based suppression. We classify the control message types with various frequencies. Control messages include (i) messages for flow entry installation; (ii) messages for flow statistics gathering; (iii) messages for system status gathering such as CPU utilization, energy consumption, and capacity utilization; and (iv) messages for network events such as failure. In this analysis phase, we have found that the priority-based suppression can effectively restrain additional traffic overhead and system resources, while packet drops without priority a scheme can abruptly peak in the traffic overhead causing significant additional data packet drops

especially under network failure events. Instead of building a sophisticated classification mechanism, we develop an MCVO system using 2 bits of the type of service (ToS) field in the IPv4 header according to the importance of the classified control message. This enables the controllers and switches to differentiate the processing sequence as well as to selectively drop received control messages. Control messages from the highest to the lowest priority include (11) messages for network events such as failure,(10) messages for flow entry installation, (01) messages for system status gathering such as CPU utilization, energy consumption, and capacity utilization, and (00) messages for flow statistics gathering. The implementation is based upon the agent where the initial messages from the switches can be annotated and further filtered. The agent also forwards packets to the different controllers according to the importance of the packets. If it is implemented in a controller, two different priority queues will be used.

The MCVO system consists of the following three facilities. The ***selective process facility*** is a process in the SDN controller to classify the urgent or important messages to be processed among the received packets according to the message type and priority. It can identify the essential messages to be processed among the received packets. The controller also can ignore certain control messages, if related decisions are already made or similar information has been seen before. The ***delegation and registration facility*** is a basic controller layer function that can specifically register control message types, quality, and a schedule to receive them. It can also delegate potential action items for other controllers, servers, or switches to expedite the response against urgent problems. The delegation and registration facility expedites the response time against the urgent

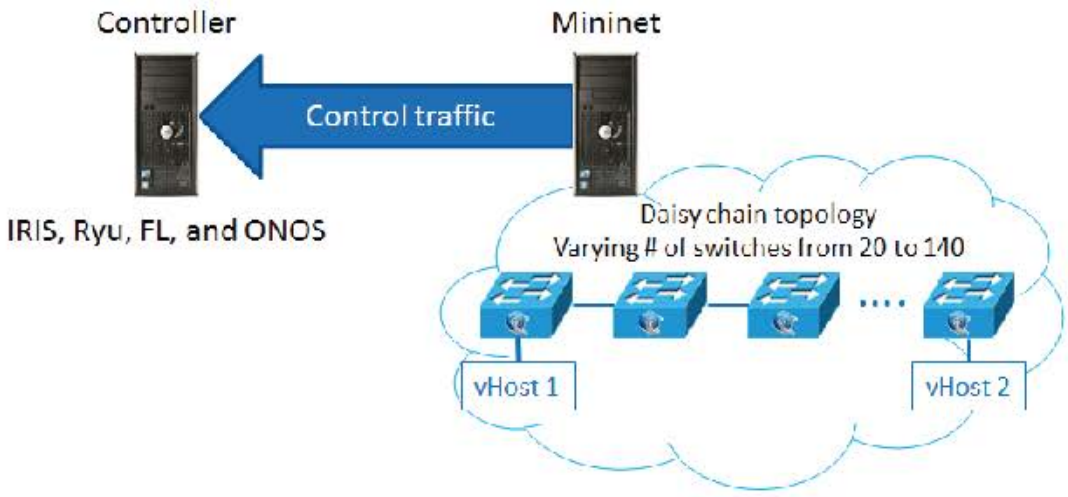


Figure 33: Control message validation experimental setup

issues by delegating actions to the immediate controllers, servers, or switches. The delegated system performs a resolution first and later reports to the controller according to the requested delegation level. The system also saves network bandwidth by reducing the amount of control message traffic. The ***correlation facility*** enhances the root cause analysis capability of the controller by providing intelligence related to the classified and prioritized control messages. It correlates the control messages with incoming traffic patterns and relationships among objects. The system also provides mechanisms to correlate control messages with other intelligences to expedite the decision process.

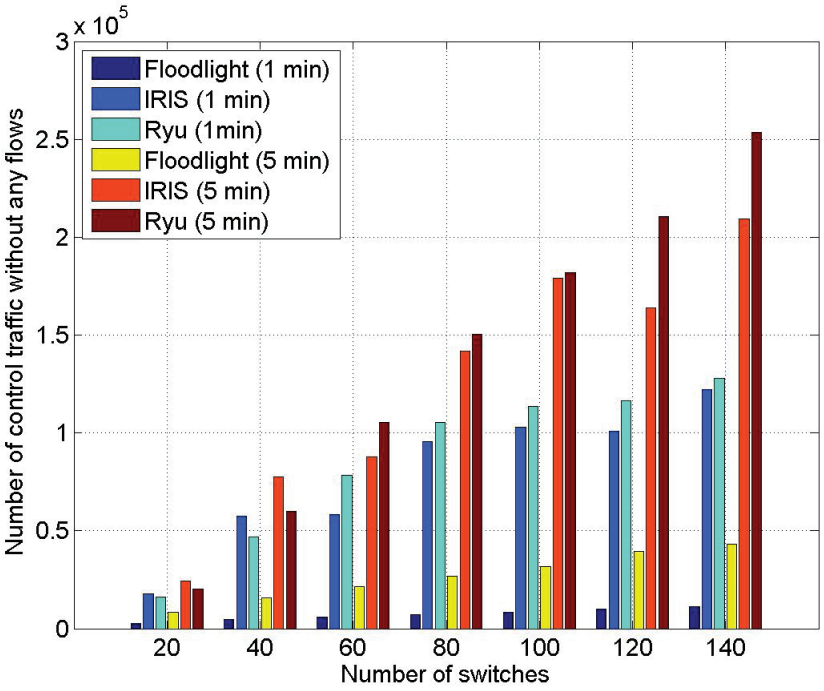


Figure 34: Combined Controller Scalability Measurement

4.8.3 Control Message Evaluation

We have made initial progress by capturing and analyzing control messages from various SDN controllers using the experimental setting in Figure 33. According to Figure 34, both IRIS and Ryu have are relatively large number of control messages. Specially, more than 50% control messages are generated during the first one minute over the course of a five minute duration. As illustrated in Figure 35, Floodlight [13] (FL) generates more echo_reply messages as the number of switches increases due to the message retransmission. The default timeout is configured in 60 seconds. The retransmission is caused by an internal logic of FL. This indicates the FL controller can be easily congested only with its

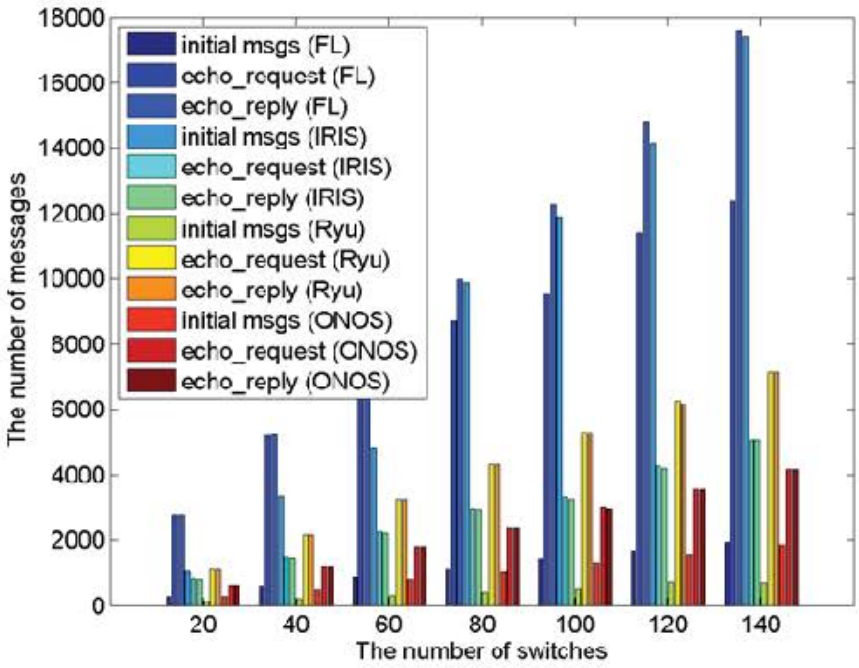


Figure 35: Initial control message analysis

own control messages. This phenomenon doesn't happen for other controllers. IRIS [14] and ONOS [1] have relatively small number of echo messages. However, IRIS creates abnormally high number of initial messages. As shown in Figure 36, FL generates the smallest number of control messages. ONOS relatively has small number of control messages. The number of control messages of ONOS linearly increases as the number of switches increases.

By analyzing the measured control messages, we can infer a few interesting SDN controller design approaches. The results indicate that the SDN controllers interpret the same OpenFlow specification differently. Some controllers use far more initial control

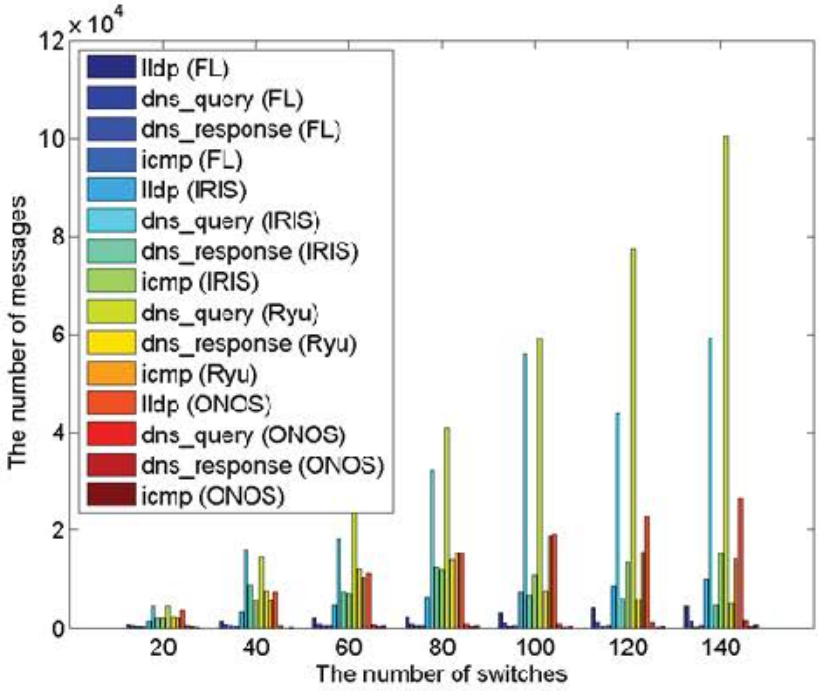


Figure 36: On-going control message analysis

messages. Also it should be noticed that there are many redundant control messages such as echo, LLDP, and ICMP for the same discovery purpose. By using the MCVO system, we will be able to further find software bugs, redundancies, and misinterpretations cases.

4.9 Summary

We have addressed various issues of HA in SDN. We first verified critical problems of SDN control path HA with the current OpenFlow specification and the existing HA solutions using a real network setup and experiments. We then proposed the HA

management framework that incorporates practical strategies towards building SDN *control path* HA including ensuring logical path redundancy aligning with physical network diversity, virtualizing a controller cluster, and exploiting topology awareness and link signals for fast and accurate failure detection and failover. We validated the efficacy of the proposed schemes with real network experiments. For future work, we further investigate a different approach to SDN high availability and scalability by validating the SDN controller software in Section 4.8. We propose a novel control Message Classification, Verification, and Optimization (MCVO) system that serves as a fundamental approach to improve the scalability and then network reliability for SDN. Unlike traditional solutions, the proposed solution will analyze, verify, and reduce the detrimental impact of an overloaded system by incorporating vitality of individual control messages. By using the MCVO algorithms and protocols in future work, we will further investigate the SDN control messages to verify the correctness of the SDN controller.

CHAPTER 5

SDN SCALABLE NETWORK MANAGEMENT

In this chapter, we discuss the problems of remote network management approaches that may lead to critical scalability issues in a network. Remote approaches are common in traditional network management and current SDN architecture to cope with the ossified network infrastructure and the underlying forwarding devices, respectively. However, since abnormal network events that occur within the network should be inferred by the remote management system on the network edge, as explained in Table 12, problems are often accumulated and enlarged, and diagnosis is delayed, inaccurate, unreliable, and not scalable. This tends to extend the legacy network's inaccurate and unreliable management problems into the control plane. In addition, the abstractions towards the remote and centralized control tend to impose excessive control traffic overhead in that a network controller needs to acquire global network visibility. All the events that occur within a network should be inferred by a centralized remote SDN controller and this requires that a network event monitoring and management system on the controller should be able to acquire global visibility of the network status as well as effectively analyze the network events to make an accurate control in time. However, as the underlying network is an inter-related complex system, it is not straightforward to identify the root cause of a problem. For example, a single problem may issue a huge amount of related syslog events as well as some faults may induce a failure which is seemingly not directly

related to the original source of the problem. These insignificant event reports may result in false negative or false positive decisions which may cause yet another network service problem. As the network system becomes more complex, it is not sufficient to rely on a single type of data to cope with network problems. The network health management system should have a data fusion facility that can collect various types of data and efficiently correlate the data in real-time. In this work, we specially focus on the SDN architecture and investigate and verify abnormal network events that can practically threaten scalability of SDN and elaborate our approaches to the current scalability issues. First of all, we explain our observations on scalability issues of SDN and elaborate experimental environments from where we can observe abnormal network events that caused those scalability issues. Then we describe our solutions for the identified scalability problems. We suggest two different approaches to solving scalability issues by implementing our proposed schemes in two different platforms; an embedded approach in the OpenFlow switch and an agent-based approach that is located near the OpenFlow switch. For the embedded approach, we propose a globally deployable Online Health management framework for SDNs (OHSDN). OHSDN addresses important network management issues including agility, accuracy, reliability, and scalability. We elaborate our implementation of OHSDN in the OpenWrt [54] based OpenFlow switch and Mininet [58]. For the agent-based approach, we propose and elaborate several scalability schemes: *Detect and Mitigate Abnormality* (*DMA*), *Modify and Annotate Control* (*MAC*), and *Message Prioritization and Classification* (*MPC*).

Table 12: Ineffectiveness of the remote management

Reason	Explanation
Agility	Since abnormal network events that occur within the network sould be inferred by the remote management system on the network edge via polling, notification, and logging, the diagnosis is delayed and the problems are often accumulated and enlarged.
Accuracy	Due to the system and performance limitations, the remote management cannot detect detailed abnormal network events within the network devices.
Reliability	Since the management message can be lost in the presence of link failure and router crashes, it is not reliable.
Scalability	Since the remote event polling consumes both system and network resources, it is not scalable when the network size or the number of monitored components increases.

5.1 Network Management Issues: Scalability

In this section, we investigate what causes the scalability issues in SDN and describe our observations. In order to identify scalability issues of SDN, we set up an SDN network with an OpenFlow switch using OpenWrt [54], controllers such as Floodlight [13], Baecon [11], and NOX [16], and hosts connected to this SDN network as illustrated in Figure 37. We also used Mininet [58] to create a large network with OpenFlow switches. We ran packETH [67] to generate data plane traffic that causes control plane traffic as well. We kept track of these control traffic flows throughout the experiments. We were able to observe that a network with even a small number of OpenFlow switches can generate enough control traffic on the network that induces congestion on the network, specially on an SDN controller.

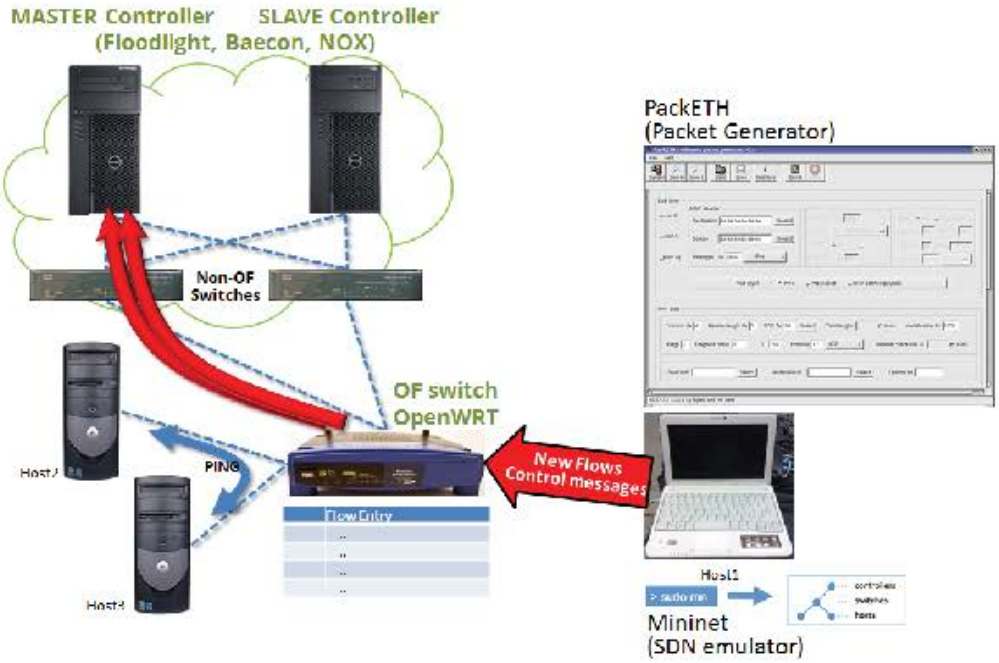


Figure 37: Overview of experimental system setting for observation of scalability issues

5.1.1 Interface Flapping

Unlike the traditional networks, a simple switch status change may cause various cascading actions in an OpenFlow network. For example, as shown in Figure 38, when an OpenFlow switch detects a port failure, it will send a port-status message to the SDN controller. Then, the SDN controller checks its network policy, currently available network topology, and routing information to find the flows that use the failed port. The controller recalculates the alternative flows and sends flow modification messages to the OpenFlow switches in order to update the flow tables. In practice, hundreds of logical interfaces can be configured for one physical port. If a port fails, it will also cause failures to all the

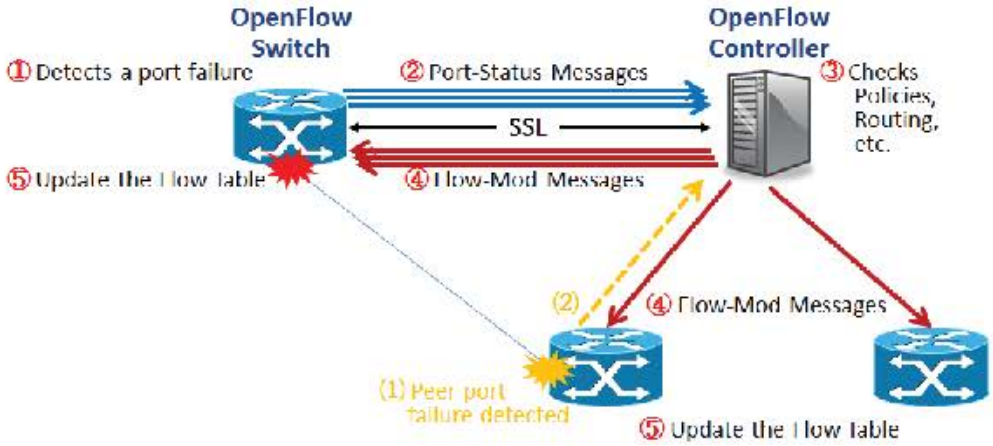


Figure 38: OpenFlow status change scenarios

logical interfaces configured on the port. The OpenFlow switch will create and send hundreds of port-status messages to an SDN controller. A port connected to the failed port on the other OpenFlow switch may also send hundreds of port-status messages. Being the single place of holding the network meta knowledge, the SDN controller needs to handle all the received port-status messages in a short time period. It checks its network policy, currently available network topology, and routing information for each request as well as updates all the information at the same time. Considering many related flows need to be updated, it may also send out thousands of flow modification messages to the switches. As all the exchanged messages are encrypted, the overhead on the SDN controller, switch, and network cannot be trivial. Furthermore, if the status keeps on changing, it may cause a significant problem on the OpenFlow network. Indeed, it is not uncommon to see that the status of an interface object keeps changing between up and down due to certain transient problems such as misconfiguration or partial physical failures. It is also possible

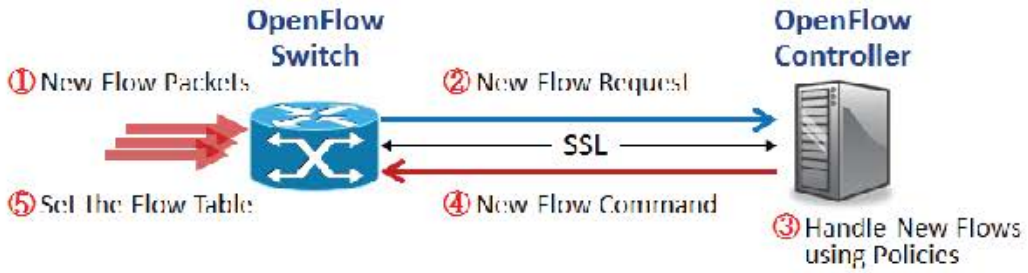


Figure 39: New flow attack

that an adversary can keep on causing up and down situations intentionally. This is called interface flapping. This unstable condition should be detected rapidly and should get the operator's attention in time. Otherwise, it may cause critical network malfunction especially in an OpenFlow network due to the related significant overheads.

5.1.2 New Flow Attack

One of the greatest advantages of SDN is that the control plane of the network is centralized and has a global view of the network. This global view enables the SDN controller to effectively and efficiently deal with traffic in the network and manage the network. However, due to this remote location apart from the data plane, an OpenFlow switch needs more effort to handle unknown incoming packets. As illustrated in Figure 39, when a packet arrives on an OpenFlow switch, it checks the flow table. If the packet does not match any flow entries in the flow table, the OpenFlow switch should send a new flow request to the SDN controller via a secure channel (SSL). The SDN controller handles the new flow request using the network policies and routing information. It makes a decision

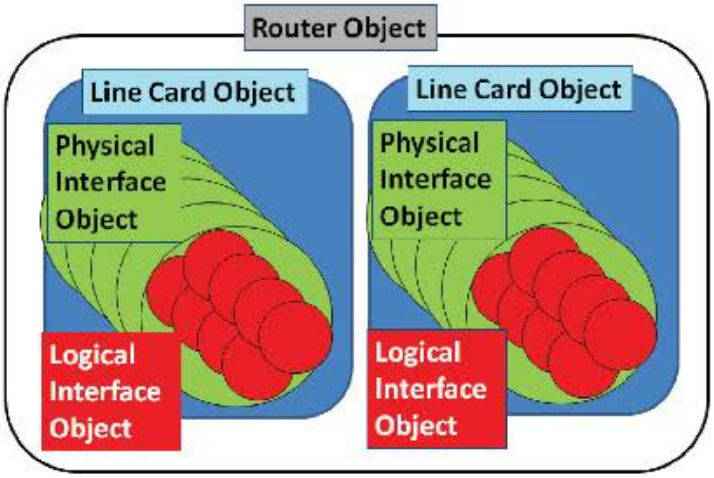


Figure 40: Object hierarchical relationships

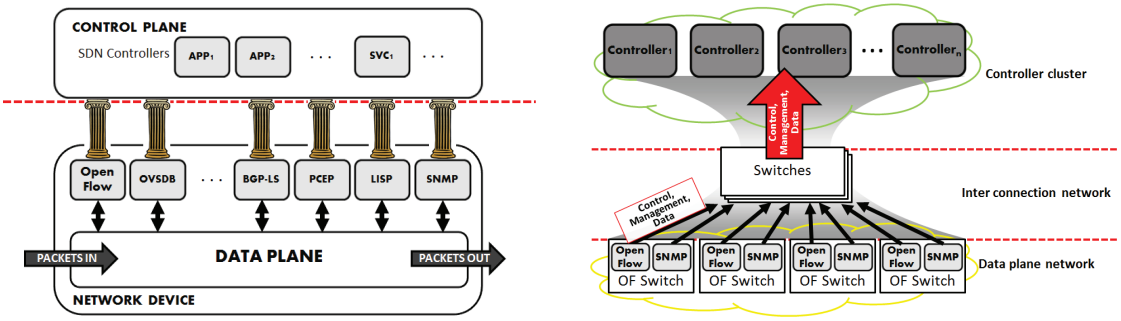
and sends a flow entry to the OpenFlow switch via a secure channel (SSL). According to the new flow decision, the OpenFlow switch adds the new flow entry into its flow table or drops the new packet. In practice, adversaries can inject randomly generated *New Flow* packets into an OpenFlow switch port to *Attack* the OpenFlow switch (named new flow attack). The OpenFlow switch needs to communicate with the SDN controller for each unknown flow packet via a secure channel. However, it may cause control and data traffic overhead to saturate CPU usage in the OpenFlow switch instantaneously. A FlowVisor is designed to act as a proxy between OpenFlow switches and multiple SDN controllers and to ensure the resource isolation within each SDN controller's domain. However, we have identified several new flow attack scenarios that the remote FlowVisor cannot handle properly.

5.1.3 Event Storm

There are hierarchical relationships among network objects. For example, as illustrated in Figure 40, a router device contains many line card objects. Each line card object also contains many physical interfaces (i.e., ports). In turn, each physical interface contains many logical or virtual interfaces. If objects are in a hierarchical relationship, a status change in an object causes status changes in all the objects it contains. This, in turn, may produce intensive status change notifications to cause an event storm. For example, a line card failure may trigger thousands of logical interface failure events as well as multiple physical interface failure events. With thousands of event notifications, it may cause tremendous overhead on the switch itself as well as the network and management services. If the event storm is not handled properly, it may cause significant problems in an OpenFlow network due to the related overhead.

5.1.4 Various Applications on the SDN Controller

SDN supports the programmable control interfaces by separating and abstracting the control plane from the data plane. SDN enables new applications, such as traffic engineering and network virtualization and further allows for rapid and simplified network exploration that improves network reliability, manageability, and security. Despite SDN's promises of flexibility and simplicity, the abstractions towards the remote and centralized control tend to extend the legacy network management's inaccurate and unreliable problems into the control plane. Although SDN's management plane is a relatively unexplored area, either the SDN controller agnostic application of the incumbent management



(a) Multiple management pillars (b) Traffic shares a physical network

Figure 41: Fundamental issues causing scalability of SDN

protocols or the full integration of the management plane into the controller's protocols such as OpenFlow cannot be a viable approach for highly dynamic SDN management. As illustrated in Figure 41(a), many recent SDN approaches evidence that SDN facilitates multiple management pillars such as customized interfaces and protocols so that the customer applications can directly communicate to the data plane to measure and monitor specific information.

SDN opens up control messages between the controllers and the forwarding devices to the communication networks. As studied in [9, 45, 73], SDN imposes excessive control traffic overheads in order for the controller to acquire global network visibility. More significantly, as shown in Figure 41(b), the overhead will be further increased by traditional network management events as well as application specific control traffic, as they may use the same physical network paths, buffers, and I/O channels at the same time. The overhead will be even worsened if the control plane uses an in-band network sharing with the data plane. If overhead is not controlled properly, it can cause various

scalability problems on networking devices, controllers, and the network itself including slow message processing, potential message drops, delayed root cause analysis, and late responses against urgent problems. Some control packets are vital to the basic functioning of SDN and should not be subject to random dropping by the controller. However, the decoupled SDN controllers do not support any packet prioritization and classification facility. According to the most recent OpenFlow [61] specification, the SDN controllers drop packets randomly regardless of the importance and urgency of the packets. The situation can be even worse, if the network and the controllers are in competition with other protocol and application traffics. In this work, we intend to facilitate a comprehensive prioritization and classification system for creating, handling, and managing network control messages in SDN. Unlike the existing scalability solutions, the proposed solution reduces the detrimental impact of an overloaded system by incorporating vitality of individual control messages. We believe that the facility services a fundamental approach to improve SDN scalability because the controllers and interconnection networks can drop fewer impacting and non-vital packets when the resource becomes limited.

5.2 SDN Scalability Management Framework: Overview

From the architectural point of view, our proposed scheme is based on a two-tier framework. As shown in Figures 42 and 43, it consists of two functional segments, the *abnormal network event detection and filtering* segment and the *abnormal network event correlation and detector management* segment. In the *abnormal network event detection and filtering* segment, our scheme is embedded in an OpenFlow Switch as a light-weight

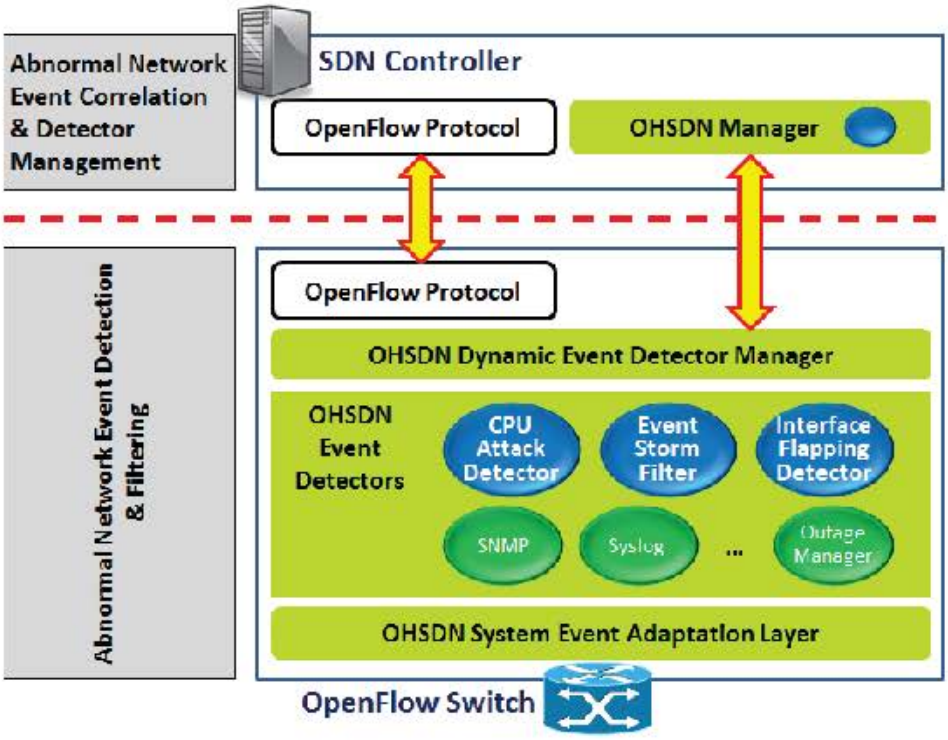


Figure 42: OHSDN management framework architecture

OpenFlow firmware extension and also implemented in the SUMA middlebox [8] as a user-defined monitoring (UM) function. Figure 44 shows the SUMA board. SUMA is implemented in a couple of multi-core network processing cards powered by a Tile-Gx36 [74] processor. It supports 36 cores and each core has 1.2GHz clock speed. It also supports a 10 Gbps packet processing capacity. Each card supports four 10G ports. As illustrated in Figure 45, the smart packet and flow filter take the fast-path to process the incoming packets with the line-rate. Common processing, basic monitoring functions are all processed over the slow-path. Virtual monitoring and function manager are implemented in the host user space and interact with other functions via the virtual monitoring

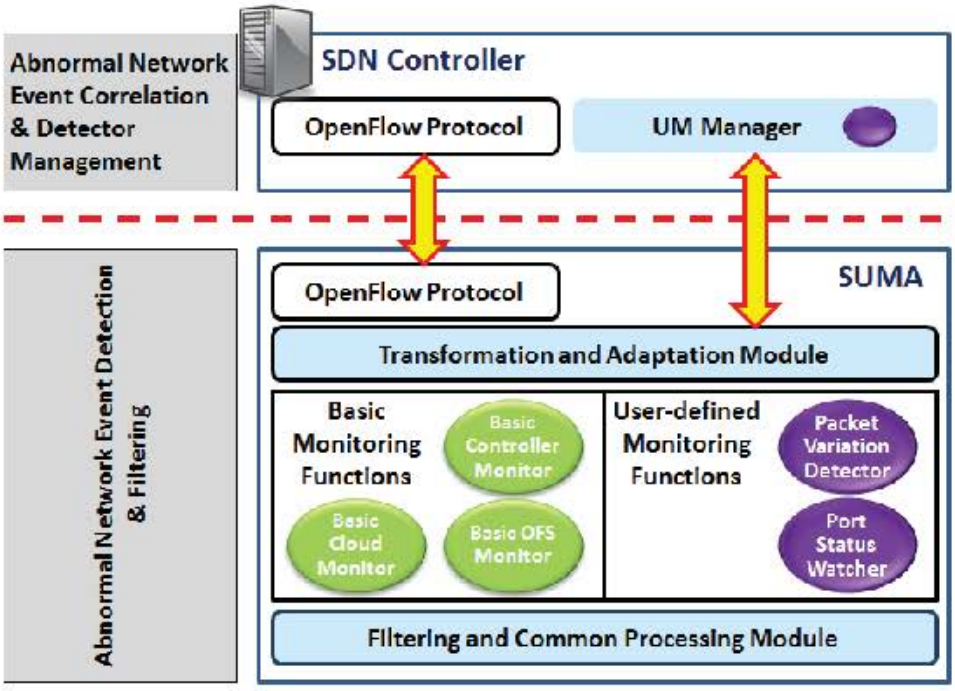


Figure 43: User-defined monitoring system architecture in SUMA

manager agent in the card. Our solutions for scalability issues are implemented as a User-defined Monitoring (UM) service and are realized on a Virtual Machine (VM). We talk about details of the *abnormal network event detection and filtering* segment for both of the architectures.

In the OHSDN architecture, it consists of *OHSDN Event Detectors*, *OHSDN Dynamic Event Detector Manager*, *OHSDN Event Publisher*, and *OHSDN System Event Adaptation Layer*. The *OHSDN Event Detectors* include basic event handlers such as SNMP, Syslog, and Outage manager as well as application specific handlers such as CPU Attack Detector, Event Storm Filter, and Interface Flapping Detector. The *OHSDN*

Figure 44: Software-defined Unified Monitoring Agent (SUMA) board (MDS-40G)

Event Detectors monitor the abnormal network events according to the configured policies. Raw data are persistently maintained within the switch, and are sent to or polled by the *OHSDN Event Publisher*. The *OHSDN Event Publisher* selects OpenFlow protocols, SNMP MIBs, or Syslogs to communicate with the remote *OHSDN manager*. The *OHSDN Dynamic Event Detector Manager* dynamically downloads network abnormality management policies from the *OHSDN Manager* in the controller and adaptively executes them in the OpenFlow switch using the *OHSDN Event Detectors*. This dynamic deployment capability reduces performance impact on the network devices and provides scalability to the management of the *OHSDN Event Detectors*. The *OHSDN System Event Adaptation Layer* facilitates a vendor independent environment to simplify the event detector deployment as well as a dynamic configuration to filter vendor critical information.

In the SUMA architecture, the *abnormal network event detection and filtering* segment consists of *Transformation and Adaptation Module*, *Basic Monitoring Functions*, *User-defined Monitoring (UM) Functions*, and *Filtering and Common Processing*

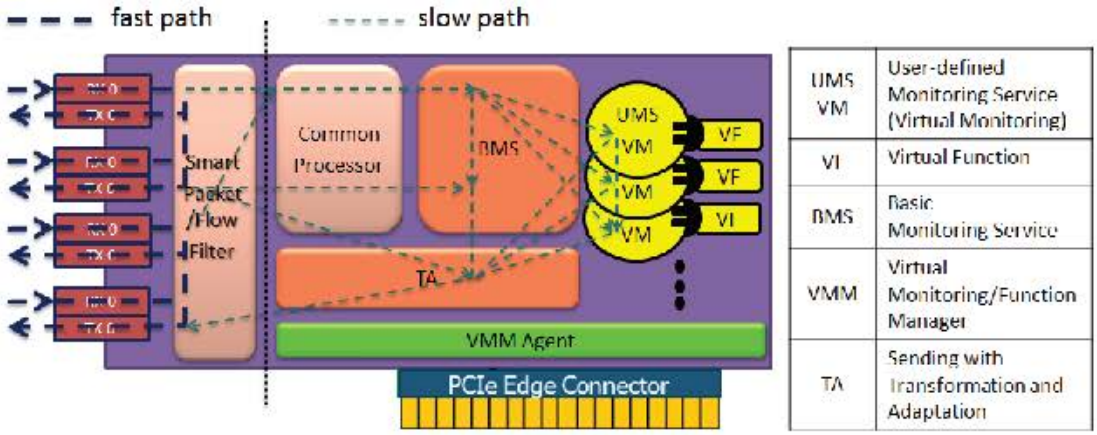


Figure 45: SUMA implementation structure

Module. The UM functions include Packet Variation Detector (i.e., MAC) and Port Status Watcher (i.e., DMA) which are equivalent to CPU Attack Detector and Interface Flapping Detector in the OHSDN architecture, respectively. Abnormal event notifications are sent to the *UM Manager* by the *Transformation and Adaptation Module*. The *Transformation and Adaptation Module* originally supports OpenFlow, SNMP, compression, and tunneling protocols to communicate with the remote managements and controllers. It requests runtime management policies and downloads dynamic event monitor modules from the SDN controllers. The *Filtering and Common Processing Module* is originally integrated with the SUMA middlebox and facilitates smart packet and flow filtering, control message/traffic aggregation, traffic classification and forwarding, and time-stamping.

Now, we go over the details of the *abnormal network event correlation and detector management* segment. The *OHSDN Manager* and the *UM Manager* can reside in

one or more controllers. Using the network topology, routing, and configuration information from the controllers as well as the abnormal network event information from the individual switches, these managers can perform network-wide abnormal network event correlation for the applications such as verification of customer's SLA and DoS attack detection. The OHSDN/UM management framework further enables an accurate root cause classification and a detailed event prediction that have been considered as not scalable or impossible to conduct. For example, facilitating a CPU utilization or packet variation measurements on each individual router can provide a potential indication of abnormal events such as a DoS attack. Traditional approaches are mainly based upon a watchdog to set a threshold (i.e., instantaneous CPU utilization is above 90%). However, this indication alone cannot be dependable information to predict abnormal network events such as a DoS attack. Hence, in practice, the ability to handle abrupt events in real-time is a very difficult issue. Instead of choosing a remote or embedded approach, our management framework harmonizes both approaches. While a light-weight embedded extension on the OpenFlow switch or the SUMA middlebox analyzes the trend of the network abnormality and rapidly responds to the network abnormality on the source of the problem, a remote system performs network-wide correlation.

5.3 Disaster Event Detectors in the OpenFlow Switch: Approach

In this section, we present three abnormal network event detectors in the OHSDN management framework such as *Interface Flapping Detector*, *CPU Attack Detector*, and *Event Storm Filter*. They are the primary components in the proposed framework.

Table 13: Notations for interface flapping detection

Notation	Explanation
FI_Th	Flapping Interval Threshold
FI	Flapping Interval to check the flapping condition
FC_Th	Flapping Event Count Threshold that is a minimum number of flapping events to start a flapping condition
FC	Flapping Event Count
Fl_Start	Flag indicating a start of a flapping condition
Fl_Notified	Flag indicating detection of a flapping condition

5.3.1 Proposed Solution Against Interface Flapping

To prevent the problems caused by the frequent status changes, we propose a light-weight, switch embedded interface flapping detection function. Table 13 explains the notations that are used for the OHSDN interface flapping detection algorithm. As described in Algorithm 5, when a set of down and up events is detected within *FI_Th*, the interface flapping detector starts to count the number of flapping events *FC*. If the *FC* exceeds the configured flapping count threshold *FC_Th* during the flapping interval *FI*, it is considered as a flapping condition. The interface flapping detector sends a *Flapping_Start* notification to the network management. It may take a follow-up action such as marking the flapping interface as a logically down status until the unstable condition is resolved. Once *FC* becomes less than *FC_Th*, a *Flapping_End* notification is sent to indicate that the flapping condition has been resolved.

5.3.2 Proposed Solution Against New Flow Attack

To ensure the CPU resource isolation against a New Flow attack, we propose a

Algorithm 5 OHSDN Interface Flapping Detection Algorithm

```
1: if a set of down and up event is detected within FI_Th then
2:    Fl_Start is Y;
3:    Fl_Notified is N;
4:    while Fl_Start is Y do
5:       reset FC;
6:       FC++ for each flapping event;
7:       wait for FI;
8:       if FC > FC_Th then
9:          if Fl_Notified ≡ N then
10:            send a Flapping_Start notification;
11:            Fl_Notified is Y;
12:         end if
13:      else
14:         if Fl_Notified ≡ Y then
15:            send a Flapping_End notification;
16:         end if
17:         Fl_Start is N;
18:      end if
19:   end while
20: end if
```

light-weight, switch embedded CPU usage detection function. Table 14 explains the notations that are used for the OHSDN CPU isolation algorithm. As shown in Algorithm 6, the CPU usage detector periodically (for a slot time interval (*STI*)) checks CPU usages of both the OpenFlow data path module (*OFdatapath*) and the OpenFlow controller connection module (*OFprotocol*). It saves the number of clock ticks *Jiffy* and the observed time *T* for the time slot *i*. A sliding window *WS* is used for a CPU usage calculation. For every *STI*, an average CPU usage is calculated using an accumulated *Jiffy* with a period time *WS* (Average CPU Usage = (*Jiffy* for *WS*) / *WS*). If the CPU usage of a port is over the threshold Th_p, the port drops incoming packets for the time interval *PDI*.

Table 14: Notations for new flow attack detection

Notation	Explanation
STI	Slot Time Interval
PDI	Packet Drop Interval
Th_p	CPU usage threshold for port *p*
Jiffy	The number of clock ticks since system boot (user mode (utime) + kernel mode (stime))
J[*i*]	Jiffy value on index *i*
T[*i*]	Time value on index *i*
WS	Window size which is the number of *STI*s
cur_index	The end index of the sliding window (mod by WS)
first_index	The first index of the sliding window (mod by WS)
OFdatapath	OpenFlow data path module (flow table)
OFprotocol	OpenFlow controller connection module (protocols)

Table 15: Notations for event storm filtering

Notation	Explanation
EDL	Event drop list
ESI	Event storm interval
SNL	Status notification list

5.3.3 Proposed Solution Against Event Storm

To prevent the event storm, we propose a light-weight, switch embedded event storm filtering function. Table 15 explains the notations that are used for the OHSDN event storm filtering algorithm. As described in Algorithm 7, when a status change event of an object is detected, the event storm filter checks the event drop list (*EDL*). If the object is already marked, the event will be ignored. Otherwise, it will check the hierarchical relationship. If the object has child objects, it will be added in *EDL* to ignore the same

Algorithm 6 OHSDN CPU Isolation Algorithm

for every *STI* for both *OFdatapath* and *OFprotocol* **do**
  read current *Jiffy* and *Time*;
  advance both *cur_index* and *first_index* by 1;
  write *J[cur_index]* and *T[cur_index]*;
  read *J[first_index]* and *T[first_index]*;
  CPUusage = (*J[cur_index]* - *J[first_index]*) / (*T[cur_index]* - *T[first_index]*);
  if *CPUusage* $\geq Th_p$ **then**
    drop the incoming packet from the port *p* for *PDI*;
  end if
end for

Algorithm 7 OHSDN Event Storm Filtering Algorithm

if a down or up event of an object *X* is detected **then**
  check *EDL* and drop if marked;
  check the *hierarchical relationship* of the object;
  if *X* has child objects **then**
    mark in *EDL* to ignore the same status change events from the child objects;
  end if
  check *SNL* to remove any child objects of *X*;
  add *X* into the notification list;
  send a notification for the objects within *SNL* after *ESI*;
end if

status change events from the child objects. The object will be further added into the status notification list (*SNL*), after removing all the child objects within *SNL*. After an interval (*ESI*), port-status messages for the objects within *SNL* will be sent to the controller. By sending a few representative object events, we can avoid the event storm.

Algorithm 8 Interface flapping detection and mitigation by DMA

```
1: initialize the value of EC to 0;
2: set Flap_Start to N;
3: set Flap_Notified to N;
4: if receive a port-status "down" message then
5:    if Start_Flag == 1 then
6:       if Flap_Interval is over then
7:          set EC to 0;
8:          set Start_Flag to 0;
9:       end if
10:      return 2;
11:   else if Start_Flag == 0 then
12:      // normal port-status down message
13:      set EC to 0;
14:   end if
15: else if receive a port-status "up" message then
16:   if Start_Flag == 1 then
17:      if Flap_Interval is over then
18:         set EC to 0;
19:         set Start_Flag to 0;
20:      else
21:         increase the value of EC by 1;
22:         if EC ≥ FC then
23:            notify the interface flapping has been detected;
24:            return 1;
25:         else
26:            warning that interface flapping may occur soon;
27:            return 2;
28:         end if
29:      end if
30:      return 2;
31:   else if Start_Flag == 0 then
32:      // new Flap_Interval starts
33:      set Start_Flag to 1;
34:      set EC to 0;
35:      increase the value of EC by 1;
36:   end if
37: end if
38: return 0; // Not in Flap_Interval
```

5.4 User-defined Monitoring Functions in the SUMA Middlebox: Approach

In this section, we continue to talk about the *abnormal network event detection and filtering* segment in a different platform. We present three UM functions in the SUMA middlebox such as *Detect and Mitigate Abnormality* (*DMA*), *Modify and Annotate Control* (*MAC*), and *Message Prioritization and Classification* (*MPC*).

5.4.1 Detect and Mitigate Abnormality (DMA)

The DMA module mainly detects interface flapping events from the network and notifies the SDN controller so that the controller can drop upcoming port-status messages. Figure 46 illustrates the algorithm of the DMA module. Algorithm 8 describes how the DMA module detects interface flapping events using port-status messages from OpenFlow switches. When a set of down and up events is detected, the interface flapping detector starts to count the number of flapping events in the event count (*EC*). If the *EC* exceeds the flapping count (*FC*) during the predefined flapping interval (*FI*), it is considered as a flapping condition. The interface flapping detector sends a *Flapping_Start* notification to the SDN controller. It may take follow up actions so as to mark the flapping interface as a logically down status until the unstable condition is resolved. Once *EC* becomes less than *FC*, a *Flapping_End* notification is sent to indicate that the flapping condition has been resolved. Upon receiving the *Flapping_End* notification, the SDN controller starts to handle upcoming port-status messages normally. A network administrator can adjust the parameters such as *EC*, *FC*, and *FI* according to their own environment.

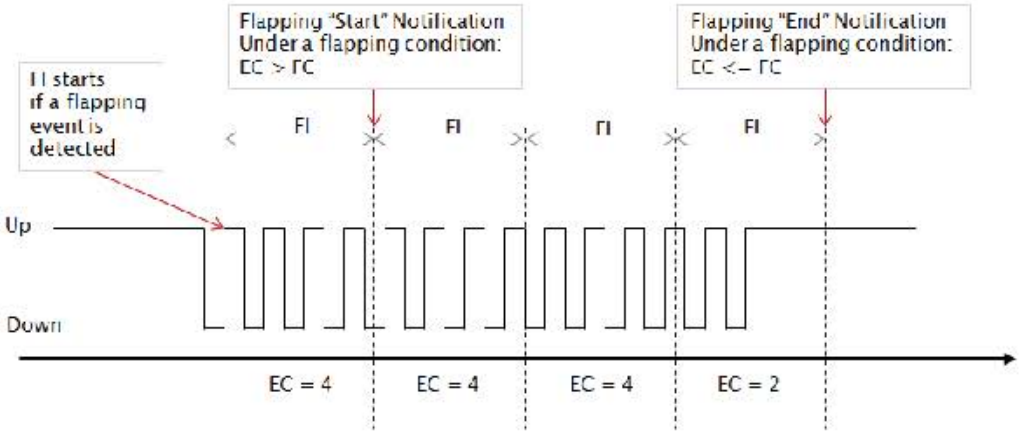


Figure 46: DMA operation during interface flapping events

5.4.2 Modify and Annotate Control (MAC)

We propose a scheme that utilizes the control message arrival patterns such as packet jitter and variation information as well as the packet count. Although the SDN controller may not be able to detect the remote switch problem by only counting the incoming packets, a different decision can be made by considering the incoming packet arrival patterns. As shown in Figure 47, the saturated switch presents very different the incoming control message patterns (packet jitter and variation information) for the same incoming packet count. In order to follow up data patterns from a specific port, we implemented a simple MAC (Modify and Annotate Control) facility that mainly detects the pattern variation.

In order to cope with problems such as following up the data patterns from a specific port, MAC provides algorithms, protocols, and facilities to modify and annotate control messages (e.g., adding sequence numbers in the control message) to assist remote

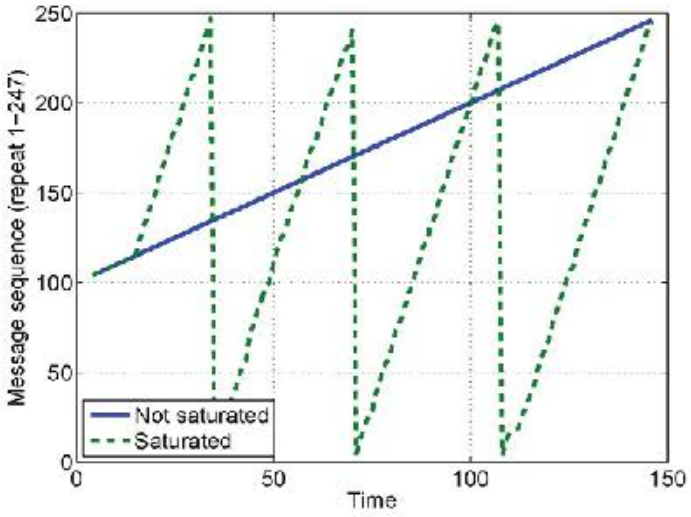


Figure 47: Different incoming packet variations

network monitoring, control message differentiation, control message differentiation, and resource isolation. According to the latest OpenFlow specification 1.4.0 [64], there are three different types of messages such as asynchronous, synchronous, and controller-to-switch messages. Especially, the synchronous messages sent by an SDN controller include a transaction identification (xID). This will be used to match the request-reply pair during operation. An SDN controller uses the "Stats-Request" message, which is one of the synchronous messages, to get a report of the statistics from an OpenFlow switch. The OpenFlow switch will respond to the request using the "Stats-Reply" message. Each request message has a unique xID. In addition, "Echo-Request" and "Echo-Reply" messages are used to check aliveness and various applications running on the SDN controller will send various synchronous messages to get global network visibility. Therefore, we can utilize these xIDs to track the incoming packets' arrival pattern variation.

5.4.3 Message Prioritization and Classification (MPC)

We perform an initial overhead analysis to understand the impact of control message drop under the priority-based suppression and packet drop without priority. We classify the control message types with varied frequencies. Control messages from the highest to the lowest priority include (11) messages for network events such as failure (10) messages for flow entry installation, (01) messages for system status gathering such as CPU utilization and capacity utilization, and (00) messages for flow statistics gathering. We found that priority-based suppression can effectively restrain additional traffic overhead and system resource use, while packet drops without a priority scheme can abruptly peak causing significant additional data packet drops, especially under network failure events. Instead of building a sophisticated classification mechanism, we develop a system to use 2 bits of the type of service (ToS) field in the IPv4 header according to the classified control message importance. This enables the controllers and switches to differentiate the processing sequence as well as to selectively drop received control messages. As illustrated in Figure 48, the implementation is based upon the agent where the initial messages from the switches can be annotated and further filtered. The agent also forwards packets to the different controllers according to the importance of the packets. If it is implemented in a controller, two different queues will be used. We also envisions providing the following three facilities. A selective process facility reduces control message processing overheads on the controller by facilitating a selective message processing mechanism. It can identify the essential messages to be processed among the received

packets. The following control messages can be ignored, if the related decisions are already made or the similar information has been seen before. A delegation and registration facility expedites the response time against the urgent issues by delegating actions to the immediate controllers, servers, or switches. The delegated system performs a resolution first and reports to the controller later according to the requested delegation level. The system also saves network bandwidth by reducing the amount of control message traffic. A controller can specifically register control message type, level, and schedule to receive. A correlation facility enhances the root cause analysis capability of the controller by providing intelligence related to the classified and prioritized control messages. It correlates the control messages with incoming traffic patterns and relationships among objects. The system also provides mechanisms to correlate control messages with other intelligences to expedite the decision process.

5.5 Experiment/Emulation Setup and Evaluation: OHSDN

In this section, we describe our experimental testbed and emulation setting and evaluate the effectiveness and efficiency of our proposed framework through extensive system experiments and simulations. We used the OpenWrt-based Linksys WRT54GL router [55] as well as used Mininet simulation for the scalable network abnormality tests.

5.5.1 The Case Against New Flow Attacks

As shown in Figure 49, three hosts and one Beacon controller are connected to the four port OpenWrt router. Host 1 uses a packETH traffic generator [67] to inject packets into the OpenFlow switch. Hosts 2 and 3 exchange ICMP messages. We create a new

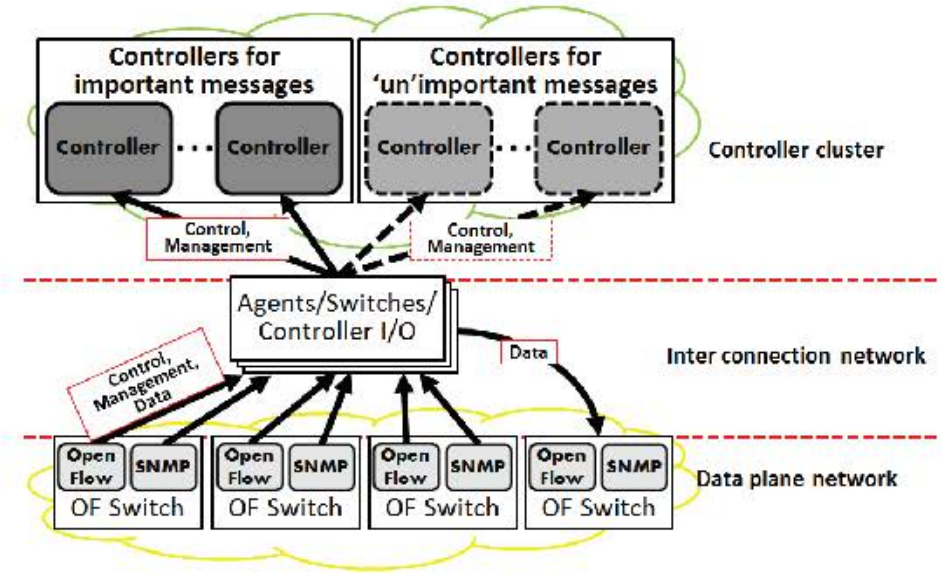


Figure 48: Proposed prioritization and classification architecture

flow attack by injecting garbage traffic into the OpenFlow switch port 1 from the traffic generator in host 1. As shown in Figure 50(a), when new flow packets are inserted into an OpenFlow switch, they cause both control and data overhead to saturate CPU usage in the switch. The CPU utilization of the OpenFlow protocol increases proportionally to the packet injection rate until the CPU is saturated. It is mainly due to the secure channel (i.e., encryption and decryption) overheads for sending new flow requests. However, it should be clearly observed that the OpenFlow protocol overhead does not exist in the traditional switches. It makes the OpenFlow switch more vulnerable to the new flow attack. Since there is no CPU isolation mechanism, the new flow attack on port 1 directly impacts the existing regular traffic between ports 2 and 3. For example, the average RTT in Figure 50(b) clearly shows that the ICMP messages between ports 2 and 3 are greatly

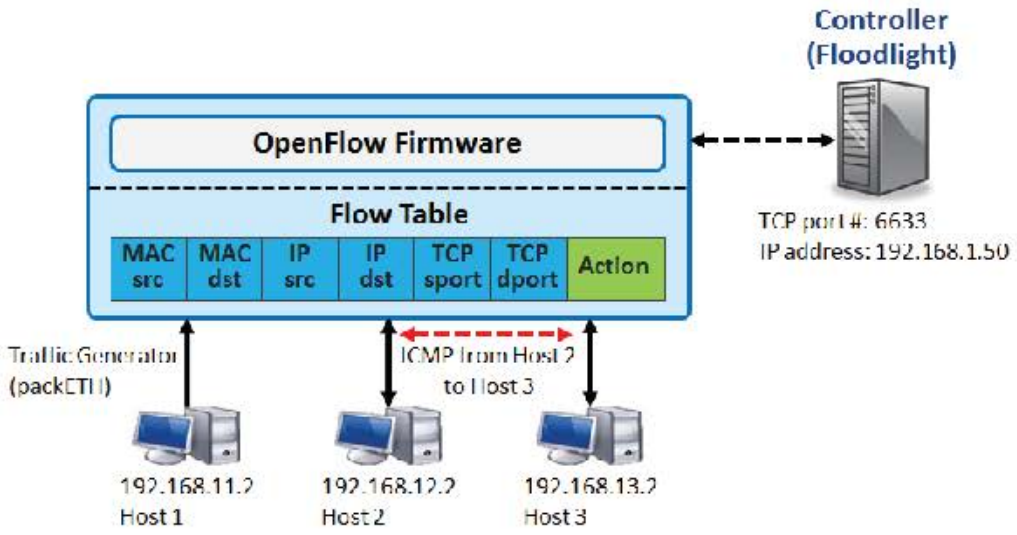
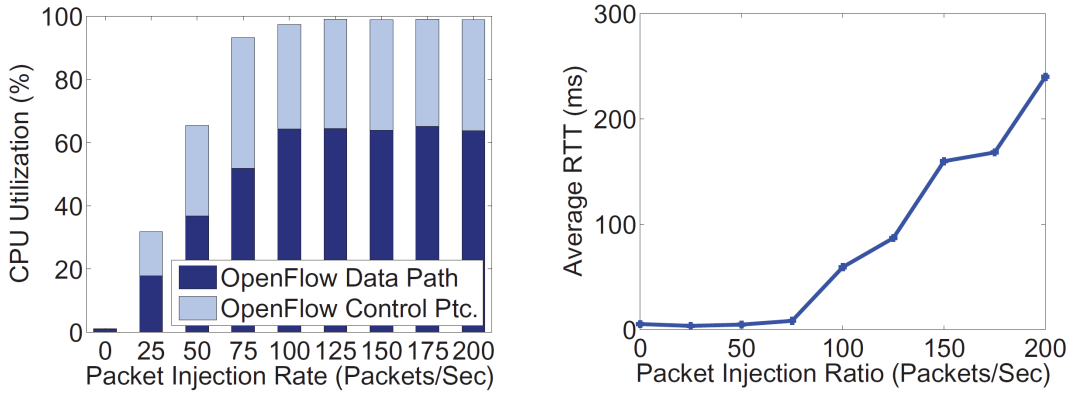


Figure 49: Experimental network setup for new flow attack

impacted by the new flow attack on port 1. These results indicate that a new flow attack is real and easy to be created. Although OpenFlow itself does not provide CPU isolation, FlowVisor [71] remotely monitors the OpenFlow switch's new flow packet count to ensure CPU isolation among the virtual slices. However, as presented in Figure 51, when the CPU is already saturated due to the abrupt injection of new flow packets, the actual number of new flow packets sent to the FlowVisor (i.e., 100 pps) can be far less than the real incoming new flow packet counts (i.e., 4000 pps). Figure 50(a) also shows that the protocol CPU utilization stays the same after the CPU saturation where the packet injection rate is over 100 pps regardless of the incoming packet rate. It indicates that only a few packets are actually sent to the FlowVisor. In this case, the FlowVisor fails to detect the critical CPU problem in the remote switch as shown in Figure 52(a). Figure 52(b) also



(a) CPU utilization

(b) Average round-trip time

Figure 50: Observation on performance under abnormal network events

presents that the FlowVisor cannot accurately ensure CPU isolation for the remote switch. Although an average CPU utilization may meet the CPU isolation target (i.e., 80%), it frequently violates the resource limitation. It is mainly due to the delayed response. These experimental results confirm that remote resource control has intrinsic difficulties. However, as presented in Figure 52(c), the proposed embedded CPU usage detection function can control the CPU resource very accurately. It also shows that the average CPU usage is better than the FlowVisor results in Figure 52(c).

5.5.2 The Case Against Interface Flapping

As illustrated in Figure 53, Mininet creates an OpenFlow network with multiple OpenFlow switches on a single virtual machine and Beacon is used as the main controller of the network. The Beacon controller runs on an Eclipse debug mode with applications including learning switch (self learning from the new flow messages), link topology

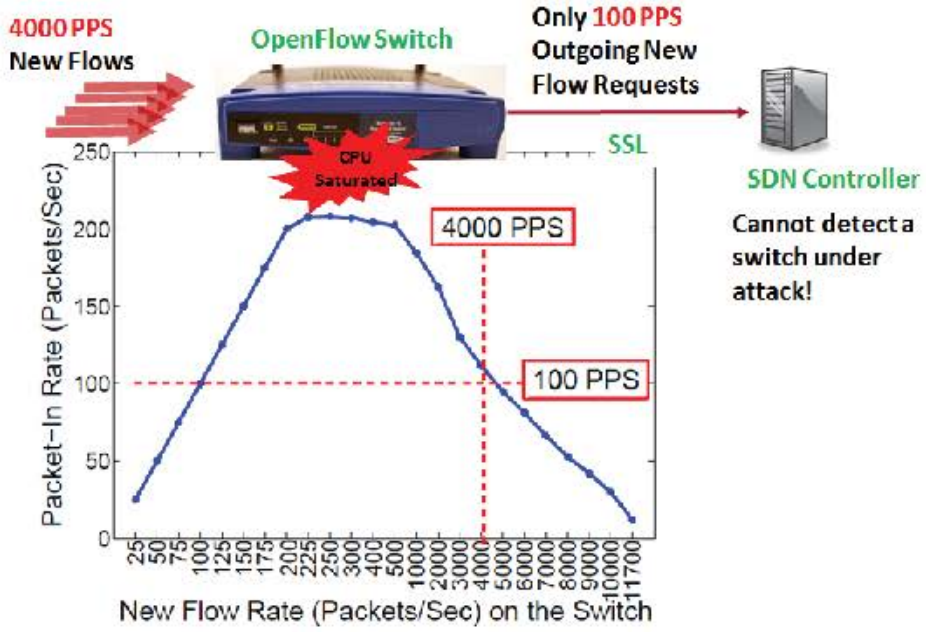
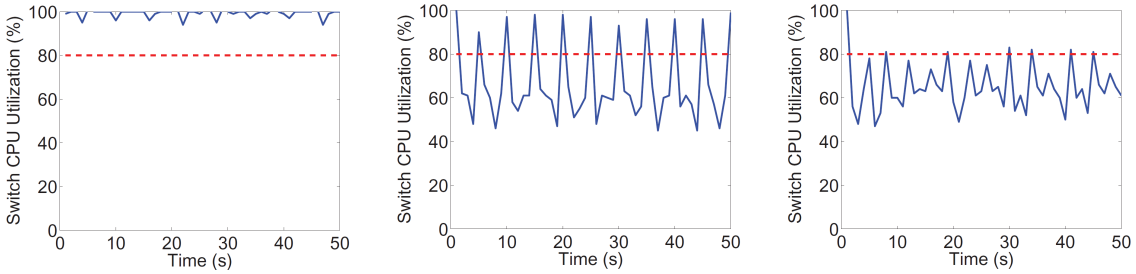


Figure 51: OpenFlow switch congestion that can not be recognized by a remote controller

discovery (links to a switch aliveness check with LLDP), and routing (APSP: All Pairs Shortest Path). We implemented up to 255 logical interfaces (configurable) for a port. We created an interface flapping by changing a port status on an emulated OpenFlow switch that also causes the status changes on the contained logical interfaces. First, we checked the controller CPU utilization by varying the number of logical interfaces. We used both NOX and Beacon controllers. As the controller needs to recalculate the existing flows and sends flow modification messages to the switches, we also changed the number of switches to see how the network size impacts the controller performance. As shown in Figure 54, CPU utilization on a controller increased proportionally to the number of logical interfaces. The result shows that the controller CPU utilization became around 35%



(a) CPU isolation failure (b) FlowVisor CPU isolation (c) OHSDN CPU isolation

Figure 52: OHSDN efficiently isolates switch's CPU while FlowVisor cannot fully control switch's CPU utilization

in a small network (three switches) with 255 logical interfaces for a port. It indicates that a simple port status change can cause great overhead in an OpenFlow network. We tested both the CPU utilization and the number of messages on a controller with an interface flapping detection algorithm by varying the number of switches. We used a Beacon controller with 255 logical interfaces for each switch port. As the controller recalculated the flows and sent flow modification messages, the network size impacted the controller performance. As presented in Figures 55(a) and 55(b), without the interface flapping detection algorithm, the controller CPU utilization was increased about 30% and the number of received messages was increased around 80K messages. Considering a relatively small network (six switches) was used in the experiment, it caused a significant performance overhead. However, with the interface flapping detection algorithm, the controller's CPU utilization was kept at less than 5% and the number of messages was far less than 5K messages. The event storm filtering algorithm was applied in this experiment along with the interface flapping detection algorithm. It indicates that a simple embedded algorithm

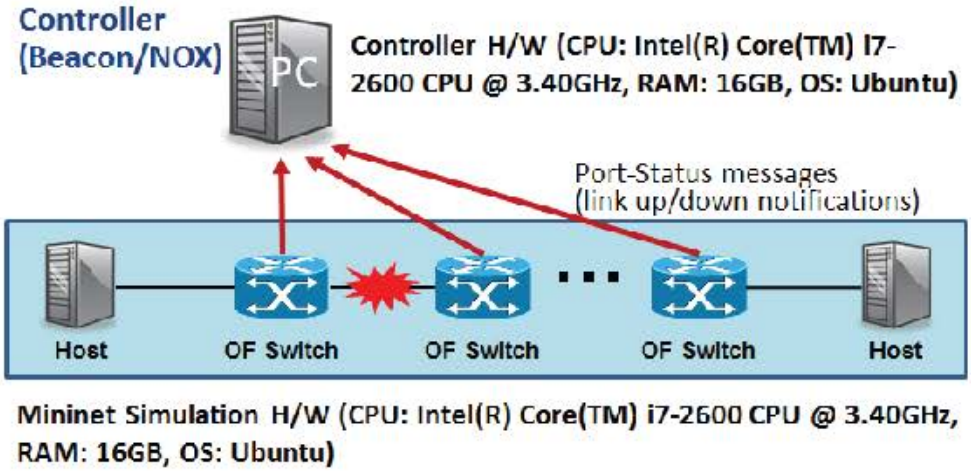


Figure 53: OpenFlow switch congestion that can not be recognized by a remote controller

can achieve significant performance improvement especially in an OpenFlow network.

5.5.3 The Case Against Event Storm

We created an event storm by changing the port status on a switch that also caused the status changes on the contained logical interfaces. We checked the average CPU utilization of a switch by varying the number of logical interfaces up to 200 ports. As shown in Figure 56, the average CPU utilization becomes around 30% with 200 logical interfaces for a port without the using event storm filtering while it is only 3% by using the event filtering algorithm. Considering typical configurations on the switches, it indicates that a simple port status change can cause great overhead in an OpenFlow network.

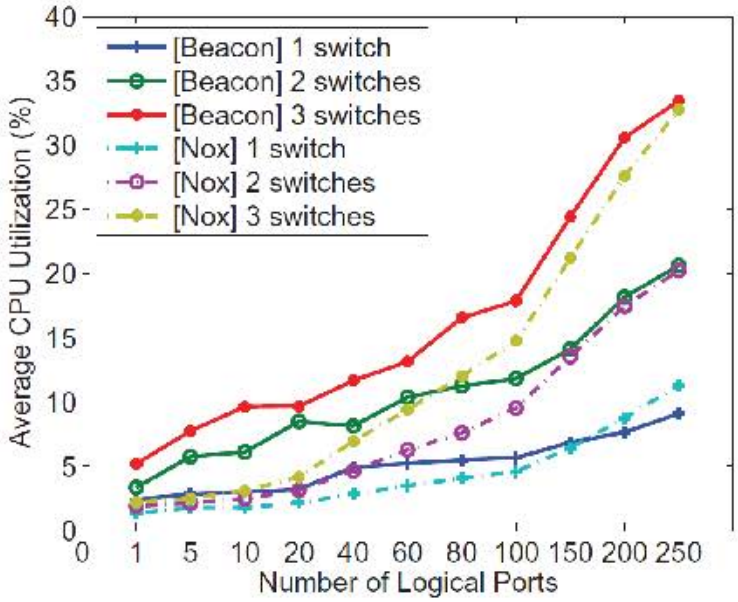
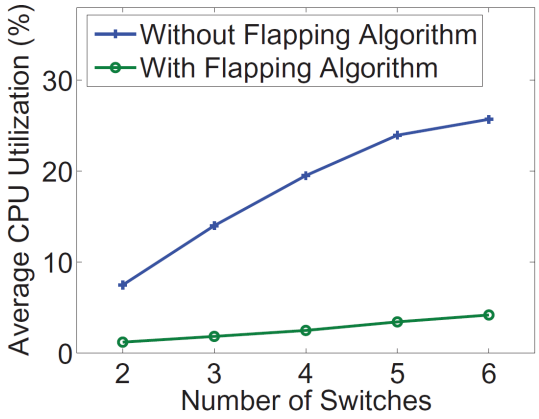


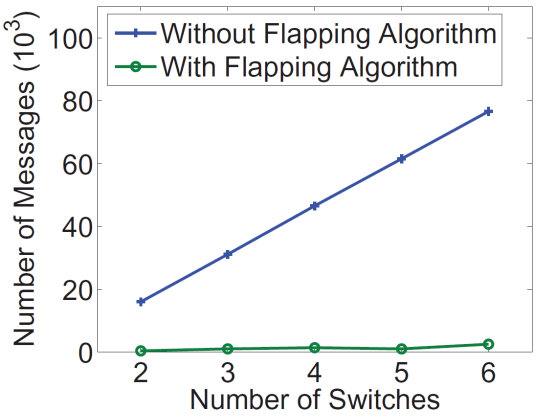
Figure 54: CPU utilization of Beacon/NOX as the number of logical interfaces changes

5.6 Experiment and Evaluation: UM Functions

In this section, we describe our experimental testbed and validate the effectiveness and efficiency of our proposed framework through extensive system experiments. We used the ETRI's network testbed as shown in Figure 57 in order to validate the proposed scalability solutions implemented in the SUMA middlebox. This network testbed is also used for the industry demonstrations at GLOBECOM 2014 [5]. As shown in Figure 58, the proposed UM monitoring modules are integrated into the ETRI's SUMA middlebox that taps the control messages from the network and interprets raw network messages into meaningful network events.



(a) Controller's average CPU utilization as the number of switches changes



(b) Port-status messages from switches as the number of switches changes

Figure 55: Comparison of average CPU utilization and port-status messages with and without flapping detection algorithm when the network scale increases

5.6.1 Detect and Mitigate Abnormality (DMA)

We validate the DMA function running in the SUMA middlebox. As explained in Section 5.4.1, DMA is designed to detect the interface flapping and notify the controller about abnormal network events. Figure 59 shows log messages obtained from the controller before and after loading the DMA module. As we can see, before we load the DMA module into the SUMA middlebox, DeviceManager and LinkDiscoveryManager modules of the controller are activated in order to respond to and cope with the interface flapping events (Case #1). After we load and run the DMA feature with the SUMA middlebox (Case #2), we can see the controller is not responding to the interface flapping events any more (Case #3). The controller simply ignores the interface flapping events from the specific port and calculates the route based on the rest of available ports or other

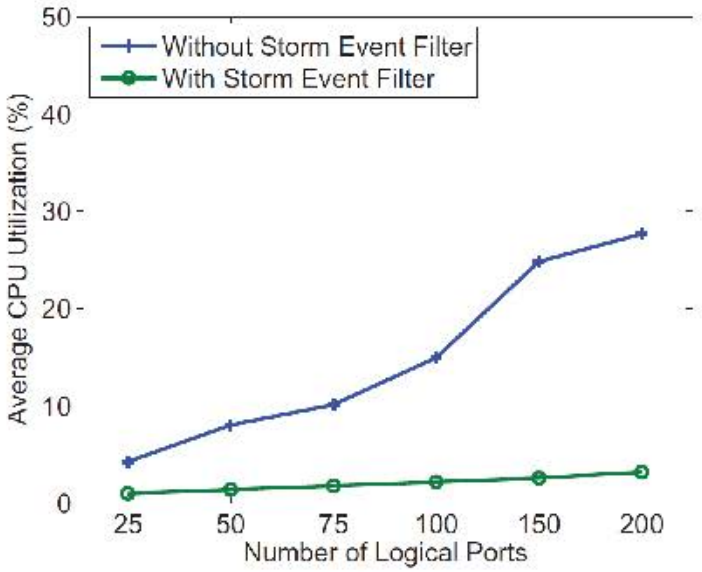


Figure 56: Event storm impacts CPU utilization

OpenFlow switches. Therefore, we can alleviate the detrimental effect of the interface flapping events.

5.6.2 Modify and Annotate Control (MAC)

We also validate the MAC function running in the SUMA middlebox. Due to the limited permission to the ETRI's network system, we couldn't capture the screen shots of the MAC operation. However, as we mentioned, the DMA and MAC functions have been demonstrated in the industry demonstrations at GLOBECOM 2014 [5] and its effectiveness and efficiency has been shown in Section 5.5.1.

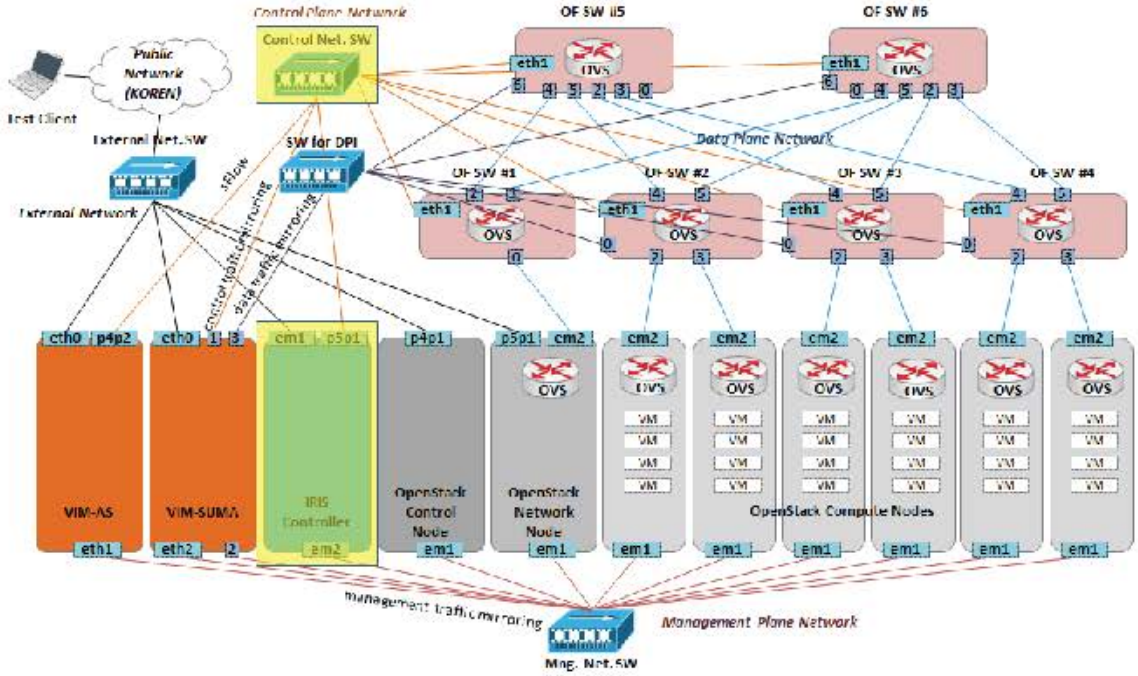


Figure 57: ETRI's experimental network architecture

5.7 SDN Scalability Framework Implementation

In this section, we describe the implementation of OHSDN abnormal network event detectors and UM monitoring services in the SUMA middlebox [8] that can be easily integrated with the current SDN systems.

For the implementation of the *abnormal network event detection and filtering* segment, we used the C, Python, and C++ languages, which are compatible with OpenWrt [54], Mininet [58], and software implementation of the SUMA board, respectively. In addition, we also used RESTful APIs to communicate between the SDN controller and the SUMA middlebox. We have implemented the CPU resource isolation function

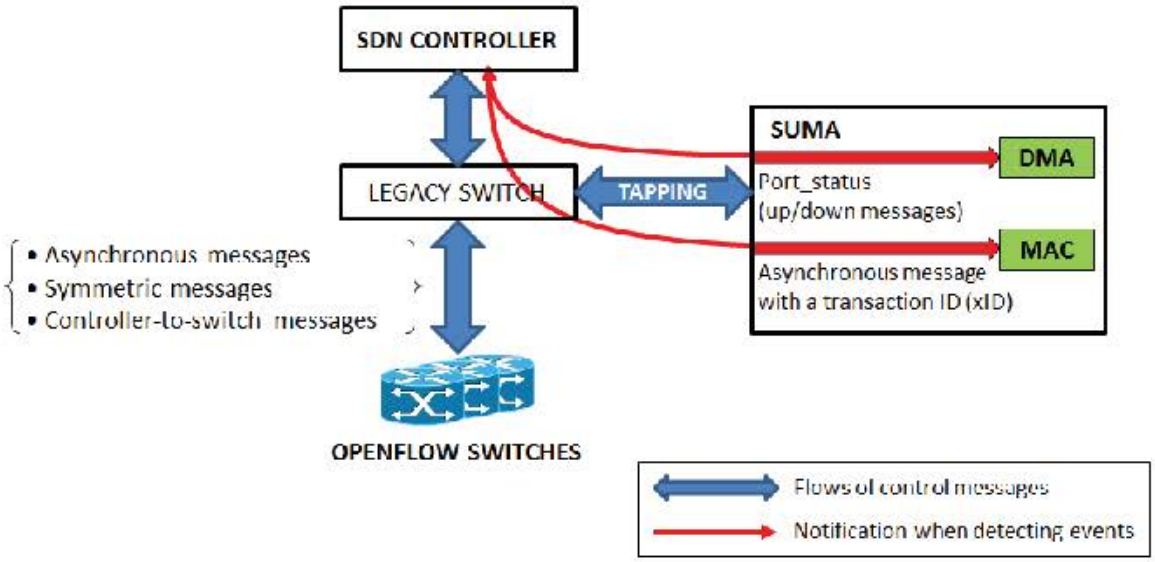


Figure 58: System architecture with DMA and MAC in the SUMA middlebox

and the event storm filtering algorithm in the OpenWrt switch. The interface flapping detection function is implemented in the Mininet simulation environment that enables the creation of a virtual OpenFlow network on a single machine. For the implementation of the *abnormal network event correlation and detector management* segment, we used Floodlight [13] and IRIS [14] controllers which are written in Java.

For the direct communication between the UM functions and the SDN controller, we implemented a communication extension using *curl* commands [21] in the *Transformation and Adaptation Module* (Figure 43). It structures the notification from the UM functions in the JSON format and sends them to the *UM Manager* in the SDN controller. Table 16 shows the available path to access the functions in the *UM Manager*. The DMA and MAC modules in the SUMA middlebox use RESTful APIs to communicate with the

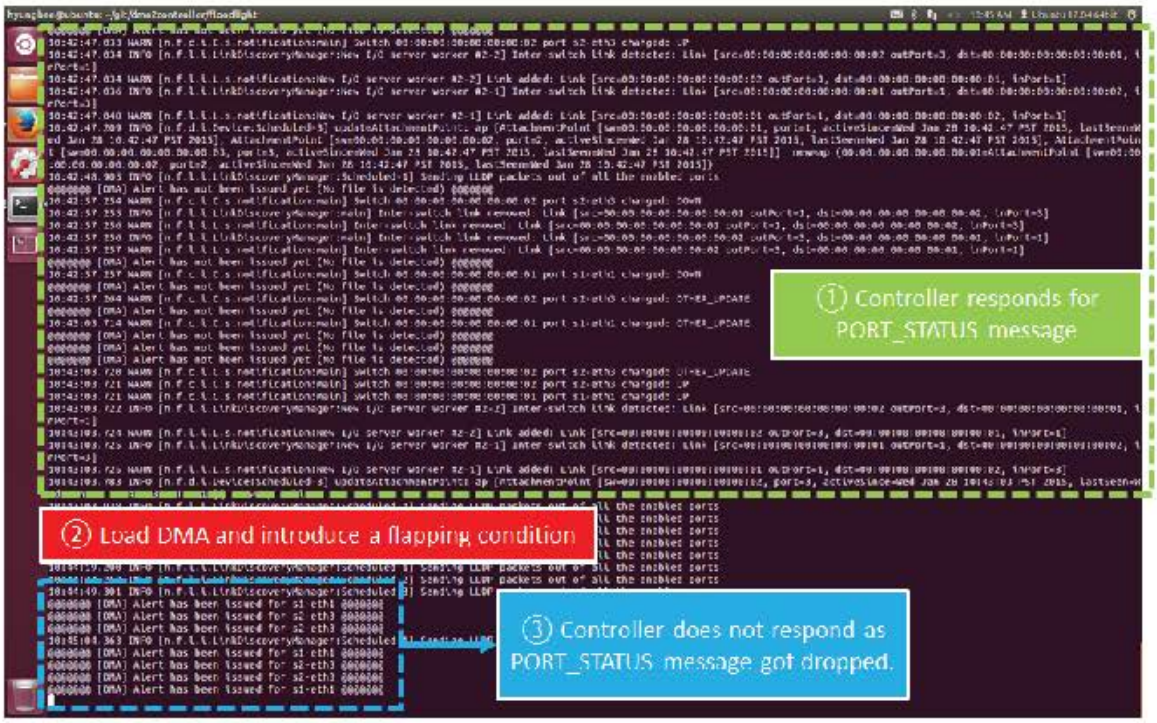


Figure 59: Controller's log messages before and after loading the DMA module into the system

SDN controller when abnormal network events are detected. As listed in Table 16, the addresses used by the DMA and MAC modules are "http://localhost:8080/wm/dmastatus/dmaalert/json" and "http://localhost:8080/wm/mac/alert/json", respectively. The detected network abnormality report will be integrated with these requests and the controller will parse the report from the request. Whenever there is a request using the above address, the SDN controller executes the module that is correlated with the requested address. The implementation on the IRIS and Floodlight controllers can be found in "Torpedo/src/etri/sdn/controller/module/ statemanager/" and "floodlight/src/main/java/it/park/", respectively.

Table 16: RESTful API URIs for the DMA and MAC modules

URI	Method	Description
/wm/dmastatus/dmaalert/json	GET	Get all the status information of ports of OpenFlow switches that currently experience interface flapping
/wm/dmastatus/dmaalert/json	POST	Send notification of interface flapping events on a specific port of a specific OpenFlow switch
/wm/mac/alert/json	POST	Send notification of congestion on a specific OpenFlow switch

5.8 Summary

We've proposed a two-tier network management framework that includes the *abnormal network event correlation and detector management* segment and the *abnormal network event detection and filtering* segment. As we discussed, in order to assess the health of a network, maintain a reliable network, and handle abnormal network events, traditional networks and current SDN architecture mainly take a remote approach to network management where raw network events or measured data are sent to a remote monitor or controller. This remote approach makes the real-time network monitoring unreliable and difficult and may cause various scalability issues as it delays root cause analysis and slows down response against urgent problems. We have shown that they are ineffective and vulnerable to various abnormal network events using concrete examples including new flow attacks, interface flapping, and event storm. We've implemented the two-tier network management framework in two different platforms. The OHSDN (Online Health

Management Framework for SDNs) is implemented in an OpenFlow switch as an embedded light-weight detector/analyzer. It is a practical abnormal network event management system. It works on SDN architecture (i.e., vendor-agnostic) and succinctly addresses the issues of agility, accuracy, reliability, and scalability. In addition, the proposed framework is implemented as an agent-based network management in the SUMA middlebox. We have shown the effectiveness of these approaches, especially compared to the plain OpenFlow environment.

CHAPTER 6

SUMMARY AND FUTURE WORK

As our daily life gets more dependent on essential and important services connected to the Internet, network reliability has never been more important. To deal with network reliability, we have mainly focused and studied network high availability and scalability. This dissertation specifically focused on Software-Defined Networks (SDN), identified new issues of network high availability and scalability of SDN, and solved the problems using various schemes and algorithms.

First of all, we addressed the various issues of network high availability in SDN. We verified the critical issues of *control path* high availability with the current OpenFlow specification and the existing high availability solutions using a real network setup and experiments. We then proposed the practical strategies towards building *control path* HA including ensuring logical path redundancy aligning with physical network diversity, virtualizing a controller cluster, and exploiting topology awareness and link signals for fast and accurate failure detection and failover. We validated the functionalities of the proposed schemes with real network experiments. Secondly, we addressed the various issues of scalability in SDN. We proposed various schemes and algorithms using event filtering, annotation, prioritization and classification techniques to alleviate the workloads of SDN controllers. Our proposed schemes and algorithms improved scalability of the SDN networks without sacrificing global view of SDN controllers or performance of OpenFlow

switches.

As future work, further investigation and development can be continued by applying the proposed HA scenarios and algorithms on the *control path* for efficient and reliable mechanisms to achieve HA for the carrier-grade SDN networks where we consider a large scale network deployment. In addition, a new approach to improve network high availability and scalability of the SDN controller can be further investigated by validating the SDN controller software. For scalability work, by comparing the performance of the proposed scalability solutions deployed in the embedded approach and the agent-based approach, the best deployment location of the scalability functions can be verified and determined in the context of SDN.

REFERENCE LIST

[1] Berde, P., Gerola, M., Hart, J., Higuchi, Y., Kobayashi, M., Koide, T., Lantz, B., O'Connor, B., Radoslavov, P., Snow, W., and Parulkar, G. ONOS: Towards an Open, Distributed SDN OS. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking* (2014), HotSDN '14.

[2] Linux Bonding Driver How-To. https://www.kernel.org/doc/Documentation/networking/bonding.txt.

[3] Cai, Z., Cox, A. L., and Ng, T. E. Maestro: A System for Scalable OpenFlow Control. Tech. Rep. TR10-11, Rice University, Dec. 2010.

[4] Casado, M., Freedman, M., Pettit, J., Luo, J., McKeown, N., and Shenker, S. Ethane: Taking Control of the Enterprise. *Proceedings of ACM SIGCOMM Computer Communication Review 37* (2007), 1–12.

[5] Choi, T., Cho, C., Yoon, S., Yang, S., Park, H., and Song, S. DEMO: Unified Virtual Monitoring & Analysis Function over Multi-core Whitebox. In *Proceedings of IEEE Global Telecommunications Conference (GLOBECOM)* (Dec. 2014).

[6] Choi, T., Kang, S., Yoon, S., Yang, S., Song, S., and Park, H. SuVMF: Software-defined Unified Virtual Monitoring Function for SDN-based Large-scale Networks. In *Proceedings of The 9th International Conference on Future Internet Technologies (CFI)* (Jun. 2014).

[7] Choi, T., Lee, B., Kang, S., Song, S., Park, H., Yoon, S., and Yang, S. IRIS-CoMan: Scalable and Reliable Control and Management Architecture for SDN-enabled Large-scale Networks. *Journal of the Network and Systems Management 23* (2015), 252–279.

[8] Choi, T., Song, S., Park, H., Yoon, S., and Yang, S. SUMA: Software-defined Unified Monitoring Agent for SDN. In *Proceedings of IEEE Network Operations and Management Symposium (NOMS)* (May 2014).

[9] The art of Application-Centric Networking. http://www.cisco.com/en/US/solutions/collateral/ns1015/ns175/ns348/ns1126/cisco_td_030513_fin.pdf.

[10] Designing a Campus Network for High Availability. http://www.cisco.com/application/pdf/en/us/guest/netsol/ns432/c649/cdccont_0900aecd801a8a2d.pdf.

[11] Beacon. http://www.beaconcontroller.net/.

[12] Big Network Controller. http://bigswitch.com/products/SDN-Controller.

[13] Project Floodlight. http://www.projectfloodlight.org/floodlight/.

[14] IRIS: The Recursive SDN OpenFlow Controller by ETRI. http://openiris.etri.re.kr/.

[15] Open Mul: High performance SDN. http://www.openmul.org/.

[16] NOX. http://www.noxrepo.org/nox/about-nox/.

[17] OpenDaylight. http://www.opendaylight.org/.

[18] POX. https://OpenFlow.stanford.edu/display/ONL/POX+Wiki.

[19] Ryu SDN framework. http://osrg.github.io/ryu/.

[20] Trema. http://trema.github.com/trema/.

[21] curl Man page. http://curl.haxx.se/docs/manpage.html.

[22] Curtis, A., Mogul, J., Tourrilhes, J., Yalagandula, P., Sharma, P., and Banerjee, S. DevoFlow: Scaling Flow Management for High-performance Networks. In *Proceedings of ACM SIGCOMM* (Aug. 2011), pp. 254–265.

[23] Desai, M., and Nandagopal, T. Coping with Link Failures in Centralized Control Plane Architecture. In *Proceedings of IEEE COMmunication Systems and NETworks (COMSNET)* (2010), pp. 79–88.

[24] Dixit, A., Hao, F., Mukherjee, S., Lakshman, T., and Kompella, R. Towards an Elastic Distributed SDN Controller. In *Proceedings of ACM SIGCOMM Workshop on HotSDN* (2013).

[25] Dixit, A., Hao, F., Mukherjee, S., Lakshman, T., and Kompella, R. ElastiCon: An Elastic Distributed SDN Controller. In *Proceedings of ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)* (2014).

[26] Element Management System (EMS) network manager. http://www.sonus.net/node/96.

[27] Project Floodlight: Module Applications. https://floodlight.atlassian.net/wiki/display/floodlightcontroller/Module+Applications.

[28] Floodlight RESTful API. https://floodlight.atlassian.net/wiki/display/floodlight controller/Floodlight+REST+API.

[29] Gude, N., Koponen, T., Pettit, J., Pfaff, B., Casado, M., McKeown, N., and Shenker, S. NOX: Towards an Operating System for Networks. In *SIGCOMM Computer Communication Review* (Jul. 2008), vol. 38, pp. 105–110.

[30] Ethernet Automatic Protection Switching (EAPS). https://tools.ietf.org/html/rfc3619, Oct. 2003.

[31] IP Multicast Load Splitting - Equal Cost Multipath (ECMP). http://www.cisco.com/c/en/us/td/docs/ios/12_2sr/12_2srb/feature/guide/srbmpath.html.

[32] Ethernet Ring Protection Switching (ERPS). http://www.cisco.com/c/en/us/td/docs/ios-xml/ios/cether/configuration/xe-3s/ce-xe-3s-book/ce-g8032-ering-pro.html.

[33] EtherChannels. http://www.cisco.com/c/en/us/td/docs/switches/lan/catalyst3550/software/release/12-1_13_ea1/configuration/guide/3550scg/swethchl.html.

[34] Fast Re-Routing (FRR). http://tools.ietf.org/html/rfc4090, May 2005.

[35] Cisco Hot Standby Router Protocol (HSRP). https:// www.ietf.org/rfc/rfc2281.txt, Mar. 1998.

[36] Link Aggregation Control Protocol (LACP). http://www.cisco.com/c/en/us/td/docs/ios/12_2sb/feature/guide/gigeth.html, Mar. 2007.

[37] Graceful OSPF Restart: Non-Stop Forwarding (NSF). http://tools.ietf.org/html/rfc3623, Nov. 2003.

[38] Non-Stop Routing (NSR). http://www.cisco.com/c/en/us/td/docs/ios-xml/ios/iproute_ospf/configuration/15-e/iro-15-e-book/iro-nsr-ospf.html.

[39] Resilient Packet Ring (RPR). http://www.ieee802.org/17/documents.htm.

[40] Stateful Switch-Over (SSO). http://www.cisco.com/c/en/us/td/docs/ios/12_0s/feature/guide/sso120s.html.

[41] Virtual Router Redundancy Protocol (VRRP) Version 3 for IPv4 and IPv6. http://tools.ietf.org/html/rfc5798, Mar. 2010.

[42] Heller, B., Sherwood, R., and McKeown, N. The Controller Placement Problem. In *Proceedings of the ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN)* (2012), pp. 7–12.

[43] JSON Data Interchange Standard. http://json.org/.

[44] GitHub: Open source parser Jsoncpp. https://github.com/open-source-parsers/jsoncpp.

[45] Kandula, S., Sengupta, S., Greenberg, A., and Patel, P. The Nature of Datacenter Traffic: Measurements and Analysis. In *Proceedings of ACM IMC* (2009), pp. 202–208.

[46] Keepalived: Load Balancing and High-Availability. http://www.keepalived.org/.

[47] Kempf, J., Bellagamba, E., Kern, A., Jocha, D., Takacs, A., and Skoldstrom, P. Scalable Fault Management for OpenFlow. In *Proceedings of IEEE International Conference on Communications (ICC)* (2012), pp. 6606–6610.

[48] Kim, D., Park, J.-W., Song, S., Choi, B.-Y., Park, H., Paik, E.-K., Jeong, K.-T., and Hong, S. Method and Apparatus for Processing a Control Message in Software-Defined Network. *Korean Patent*, 1020130143244 (Nov. 2013).

[49] Kim, H., Santos, J., Turner, Y., Schlansker, M., Tourrilhes, J., and Feamster, N. CORONET: Fault Tolerance for Software Defined Networks. In *Proceedings of IEEE International Conference on Network Protocols (ICNP)* (2012).

[50] Koponen, T., Casado, M., Gude, N., Stribling, J., Poutievski, L., Zhu, M., Ramanathan, R., Iwata, Y., Inoue, H., Hama, T., and Shenker, S. ONIX: A Distributed Control Platform for Large-scale Production Networks. In *Proceedings of USENIX conference on Operating Systems Design and Implementation (OSDI)* (2010).

[51] Krishnamurthy, A., Chandrabose, S. P., and Gember-Jacobson, A. Pratyaastha: An Efficient Elastic Distributed SDN Control Plane. In *Proceedings of the Workshop on Hot Topics in Software Defined Networking (HotSDN)* (2014), pp. 133–138.

[52] Kuźniar, M., Perešíni, P., Vasić, N., Canini, M., and Kostić, D. Automatic Failure Recovery for Software-defined Networks. In *Proceedings of the ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN)* (2013), pp. 159–160.

[53] Lee, K., Jang, I., Shin, M., and Baek, S. Design of Super Controller for Large Scale Software-Defined Networks. In *OSIA Standards & Technology Review* (Sep. 2012).

[54] Pantou: OpenFlow 1.0 implementation for OpenWRT. http://archive.openflow.org/wk/index.php/Pantou_:_OpenFlow_1.0_for_OpenWRT.

[55] Linksys WRT54GL. http://support.linksys.com/en-us/support/routers/wrt54gl.

[56] Luo, T., Tan, H.-P., Quan, P., Law, Y. W., and Jin, J. Enhancing Responsiveness and Scalability for OpenFlow Networks via Control-Message Quenching. In *Proceedings of International Conference on ICT Convergence (ICTC)* (2012), pp. 348–353.

[57] MaKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. OpenFlow: Enabling Innovation in Campus Networks. *Proceedings of ACM SIGCOMM Computer Communication Review 38* (2008), 69–74.

[58] Mininet: An Instant Virtual Network on your Laptop (or other PC). http://mininet.org/.

[59] Network Functions Virtualization (NFV). http://www.etsi.org/technologies-clusters/technologies/nfv.

[60] Network Management System (NMS). http://www.cisco.com/c/en/us/support/docs/availability/high-availability/15114-NMS-bestpractice.pdf.

[61] Open Networking Foundataion (ONF). https://www.opennetworking.org/.

[62] Production Quality, Multilayer Open Virtual Switch (Open vSwitch). http://openvswitch.org/.

[63] OpenFlow Switch Specification. https://www.opennetworking.org/index.php?option=com_content&view=category&layout=blog&id=57&Itemid=175&lang=en.

[64] OpenFlow Switch Specification Version 1.4.0. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf, Oct. 2013.

[65] Operations Support System (OSS) / Business Support System (BSS). http://www.ericsson.com/res/thecompany/docs/publications/business-review/2012/issue2/oss-bss_explained.pdf.

[66] Open vSwitch Database. http://openvswitch.org/ovs-vswitchd.conf.db.5.pdf.

[67] packETH. http://packeth.sourceforge.net/packeth/Home.html.

[68] Park, H., Song, S., Choi, B.-Y., and Choi, T. Toward Control Path High Availability for Software-Defined Networks. In *Proceedings of The 11th International Conference on Design of Reliable Communication Networks (DRCN)* (Mar. 2015).

[69] Park, J.-W., Kim, D., Song, S., Choi, B.-Y., Park, H., Paik, E.-K., Jeong, K.-T., and Hong, S. Method for Establishing Connection between Switches and Controllers in Software-Defined Network. *Korean Patent*, 1020130143289 (Nov. 2013).

[70] Park, S. H., Lee, B., You, J., Shin, J., Kim, T., and Yang, S. RAON: Recursive Abstraction of OpenFlow Networks. In *Proceedings of the Third European Workshop on Software Defined Networks (EWSDN)* (Sep. 2014), pp. 115–116.

[71] Sherwood, R., Gibb, G., Yap, K.-K., Appenzeller, G., Casado, M., McKeown, N., and Parulkar, G. Flowvisor: A network virtualization layer. Tech. Rep. OPENFLOW-TR-2009-1, Deutsche Telekom Inc. R&D Lab, Stanford University, Nicira Networks, Oct. 2009.

[72] A Simple Network Management Protocol (SNMP). http://www.ietf.org/rfc/rfc1157.txt.

[73] Tavakoli, A., Casado, M., Koponen, T., and Shenker, S. Applying NOX to the Datacenter. In *Proceedings of HotNets* (2009).

[74] TILE-Gx36 Multicore Processor. http://www.tilera.com/products/?ezchip=585&spage=621.

[75] Tootoonchian, A., and Ganjali, Y. Hyperflow: a distributed control plane for openflow. In *Proceedings of Internet Network Management Conference on Research on Enterprise Networking (INM/WREN)* (2010), USENIX Association.

[76] Unified QoS - Locally generated packet prioritization. http://www.cisco.com/c/en/us/td/docs/ios/12_0s/feature/guide/unif_qos.pdf.

[77] Williams, D., and Jamjoom, H. Cementing High Availability in OpenFlow with RuleBricks. In *Proceedings of ACM SIGCOMM Workshop on HotSDN* (2013), pp. 139–144.

[78] Yan, H., Maltz, D., Ng, T., Gogineni, H., Zhang, H., and Cai, Z. A 4d Network Control Plane. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2007).

[79] Yeganeh, S. H., and Ganjali, Y. Kandoo: a framework for efficient and scalable offloading of control applications. In *Proceedings of HotSDN* (Aug. 2012).

[80] Yu, M., Rexford, J., Freedman, M., and Wang, J. Scalable Flow-Based Networking with DIFANE. In *Proceedings of ACM SIGCOMM* (Aug. 2010), pp. 351–362.

[81] Yun, W.-D., Koo, T.-H., Song, S., Choi, B.-Y., Park, H., kyoung Paik, E., Jeong, K.-T., and Hong, S. Method and System for detecting network failure in Software Defined Network. *Korean Patent*, 1020130143238 (Nov. 2013).

VITA

Hyungbae Park was born in Seoul, South Korea, on August 31, 1980. Mr. Park entered the Kwangwoon University in South Korea in March 1999 and graduated in December 2004 with a Bachelor of Engineering degree in Computer Engineering. Between 2000 and 2002, Mr. Park performed his military service in the Korean army. After graduation, Mr. Park came to the USA to study and gained a Master of Science degree in Computer Science from South Dakota State University at Brookings in South Dakota in December 2007. His thesis topic was implementing a fault tolerance facility for Linux Kernel 2.6.

After his Master degree, Mr. Park joined interdisciplinary Ph.D. curriculum at the University of Missouri-Kansas City in 2008. His coordinating discipline is Computer Science and his co-discipline is Telecommunication and Computer Networking. His main research interest and dissertation topic is highly availability and scalability of Software-Defined Networking (SDN). Mr. Park gained Outstanding Ph.D. student award from Telecommunication and Computer Networking discipline in 2013. Mr. Park also performed several distinct projects in various areas such as location privacy in wireless sensor networks, campus network outage traffic analysis and modeling, and smartphone-based collision avoidance system, localization and authentication.