Fall 12-2015

Using Software-Defined Networking to Improve Campus, Transport and Future Internet Architectures

Adrian Lara
University of Nebraska-Lincoln, adrianlara@gmail.com

USING SOFTWARE-DEFINED NETWORKING TO IMPROVE CAMPUS, TRANSPORT AND FUTURE INTERNET ARCHITECTURES

by

Adrian Lara

A DISSERTATION

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Doctor of Philosophy

Major: Computer Science

Under the Supervision of Professor Byrav Ramamurthy

Lincoln, Nebraska

December, 2015

USING SOFTWARE-DEFINED NETWORKING TO IMPROVE CAMPUS, TRANSPORT AND FUTURE INTERNET ARCHITECTURES

Adrian Lara, Ph.D.

University of Nebraska, 2015

Adviser: Byrav Ramamurthy

Software-defined Networking (SDN) promises to redefine the future of networking. Indeed, SDN-based networks have unique capabilities such as centralized control, flow abstraction, dynamic updating of forwarding rules and software-based traffic analysis. SDN-based networks decouple the data plane from the control plane, migrating the latter to a software controller. By adding a software layer between network devices and applications, features such as network virtualization and automated management are simpler to achieve.

In this dissertation, we show how SDN-based deployments simplify network management at multiple scales such as campus and transport networks, as well as future Internet architectures. First, we propose OpenSec, an SDN-based security framework that allows network operators to implement security policies in campus networks. Second, we propose the eXtensible Traffic Engineering Framework (XTEF) to enable application-driven traffic engineering and provision transport network resources using on-demand Wavelength Division Multiplexing (WDM) tunnels. Third, we demonstrate how SDN can be used to dynamically create intra-domain cut-through switching tunnels to bypass the routing layer in MobilityFirst. Finally, we propose how to extend the cut-through capabilities to inter-domain routing in MobilityFirst.

In our work, we run experiments on the GENI testbed (Global Environment

for Network Innovations), the ORBIT (Open-Access Research Testbed for Next-Generation Wireless Networks) and Mininet. The results show that SDN can be used to simplify policy-based network management, virtualize an entire WAN as a single switch, create Wavelength Division Multiplexing (WDM) tunnels on demand and create inter-domain tunnels using techniques that scale better than traditional distributed methods.

ACKNOWLEDGMENTS

I would first like to thank my advisor, Prof. Byrav Ramamurthy, for his infinite patience to guide me through my Ph. D. He provided a working environment where I was never afraid of asking questions and felt motivated to work. Likewise, I would like to thank my committee members, Prof. Witawas Srisa-an, Prof. Mark Walker and Prof. Lisong Xu, for their valuable feedback and suggestions to improve my work. Other professors have also contributed significantly to improve the quality of my research. In particular, I would like to thank Dr. Nagaraja at Ericsson, Prof. Raychaudhuri at Rutgers University and Prof. Ramakrishnan at UC Riverside, for their immense help to understand and investigate the MobilityFirst project. Next, I would like to thank Mr. Inder Monga and Mr. Eric Pouyoul, for their feedback and cutting-edge research ideas during my time at ESnet and afterwards. Finally, I would like to thank Ms. Anisha Kolasani, Mr. Aravind Krishnamurthy and Ms. Shreyasee Mukherjee for co-authoring papers with me.

Next, I would like to thank the National Science Foundation for partially funding this work through grants CNS-1040765 and CNS-1345277. Likewise, I would like to thank BBN Technologies for continuously funding my travels to the GENI Engineering Conference meetings. Likewise, I would like to thank the US Dept. of Energy (DOE) for partially funding this work through Award Number DE-SC0001277.

I would also like to thank all my friends at the Netgroup Lab throughout these years. Mr. Santiago Gimenez, Mr. Saichand Palusa, Ms. Pan Yi, Ms. Sara El Alaoui and many others listened to my presentations repeatedly and always provided suggestions to improve my work.

I would have never finished my Ph. D. without the support of my parents and

sister. They supported this adventure even before it started and pushed me to pursue my dreams and make them come true. I owe them for teaching me to face life with a smile on my face and to never look back. Many thanks to them for always being there for me.

Finally, my biggest gratitude goes to Mariana, my fiancee, for being strong during these years, for believing in me and for finding the perfect words to cheer me up every time. The end of my Ph. D. journey also means a new beginning with her back home, and I look forward to many years together.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Networks of the future need to be flexible and dynamic to accommodate the expected demand. For example, global mobile data reached 2.5 exabytes per month at the end of 2014 [5] and a growth of more than 50% is expected till 2019, where the data load could reach 24.3 exabytes. By the same token, the Large Hadron Collider (LHC, the world's largest particle accelerator), is capable of sending data at a rate of 10 gigabytes per second for processing at facilities worldwide [6]. Clearly, networks need to provide sufficient bandwidth to transfer such huge amounts of data. However, they also need to be dynamic and flexible so that data transfers can be initiated when needed. To illustrate this, imagine that the LHC generates 10 gigabytes but it takes two days to create a circuit in order for a scientist in New York to receive the data. By then, it is already faster to ship two DVDs by mail.

Networks of the future also need to be manageable. As the complexity of network deployments increases, the number of devices deployed increases as well. Manual configuration of such devices is error-prone and time-consuming. As a consequence, one expectation of future networks is to provide mechanisms to the operators to simplify network management, reduce the amount of time needed to configure devices and minimize configuration errors.

Software Defined Networking (SDN) [7] is an increasingly popular paradigm that decouples the data plane from the control plane, migrating the latter to a software controller. The motivation is to move the complexity away from the hardware and to allow for flexibility and innovation in the software. As a result, hardware tends to become simpler and focus only on forwarding traffic. In contrast, the software becomes responsible for managing the network and how devices forward traffic. SDN deployments have inherent capabilities that allow for innovative ways to manage networks. First, control of the network is centralized, since a software controller is responsible for managing all forwarding devices. Second, all traffic is abstracted as a flow, independently of whether it carries data over IP, Ethernet or lower layers. Third, in SDN forwarding rules can be updated dynamically, thus providing great flexibility to network management. Fourth, SDN facilitates software-based analysis of traffic traversing the forwarding devices. This allows the centralized controller to use multiple software techniques for detection and pattern identification. All these capabilities have enabled a significant body of research focusing on how SDN can redifine the way networking is done today.

OpenFlow [8] is the most commonly used SDN protocol. It standardizes how a software-based controller and an OpenFlow-compliant Ethernet switch should communicate. The protocol contains an extensive list of messages that are used by the controller and the switches to exchange information. For instance, it allows a switch to notify the controller of an incoming packet for which no forwarding rule matched. Similarly, it specifies how the controller can send a message to the switch requesting to add a new forwarding rule in the flow table that will match the incoming packet. These are only two examples of messages that can be exchanged through the OpenFlow channel, a Transmission Control Protocol (TCP) connection established between the controller and the switch to exchange data.

1.1 Motivation

The motivation of this dissertation is to analyze how SDN can be used to simplify network operation and create flexible and manageable deployments. By adding a software layer between network devices and applications, a great deal of simplification can be achieved. However, it is also true that anything that can be done using SDN can also be achieved without it. The challenge, thus, is to justify how SDN-based deployments are more flexible and manageable compared to traditional networks.

When OpenFlow was proposed, it was considered a tool to enable network innovation in campus networks. The benefit of using OpenFlow was to allow researchers to run novel experiments on real hardware without disrupting production traffic. As a consequence, an effort to simplify network management at campus scale resulted in OpenSec [9], the first product of this dissertation. OpenSec is an SDN-based security framework that allows network managers to implement security policies in the network.

When the popularity of SDN increased, so did the scope of SDN-based deployments. In particular, the deployment of an entire backbone network at Google using OpenFlow [10] was a demonstration that an SDN-based wide area network (WAN) was possible. Our motivation to shift to the WAN was driven by our interest in the optical layer. As a consequence, we focused on how SDN could simplify transport networks and we proposed the the eXtensible Traffic Engineering Framework (XTEF) [11] to provide application-driven traffic provisioning using on-demand Wavelength Division Multiplexing (WDM) tunneling in SDN transport networks.

Finally, our collaboration with the MobilityFirst team motivated us to look into

the role of SDN in the Future Internet. MobilityFirst is a Future Internet architecture project funded by the National Science Foundation that decouples network addresses and identifiers to better support mobility. To achieve mobility-awareness, the MobilityFirst network must be capable of fine-grained, per-flow routing to provide mobility support while staying efficient for large flows that traverse the core network. Consequently, we designed an SDN-based routing framework for MobilityFirst that benefits from the traffic analysis and centralized control capabilities of SDN to ensure mobility-aware, efficient routing in MobilityFirst.

1.2 Background

1.2.1 Software Defined Networking and OpenFlow

Since this work focuses heavily on SDN and OpenFlow, an entire chapter (Chapter 2) is devoted to provide a detailed explanation of the topic. This chapter includes a brief description of the historical background of programmable networks and the reasons why OpenFlow was more successful than previous attempts. The chapter also provides a detailed description of the OpenFlow protocol, starting at the specification 1.0. Finally, we survey innovative work using OpenFlow, as well as current large scale deployments based on SDN.

1.2.2 Future Internet Architectures and MobilityFirst

Future Internet Architectures (FIAs) are research projects aimed at re-designing the Internet. The common thread of all projects is the identification of limitations of the current Internet, such as the lack of support for mobile devices or the lack of security. Also, they highlight that the Internet was designed to enable communication between one client and one server, whereas nowadays Internet traffic

is content-oriented independently of the location of the required content. These projects were born in 2010 when the National Science Foundation (NSF) created the FIA project and funded novel initiatives to propose clean slate architectures for the Future Internet. Projects such as Named Data Networking (NDN) [12], MobilityFirst [13], Nebula[14] and ChoiceNet [15] are currently funded by the NSF as a result of the initiative. We are part of the MobilityFirst team and we collaborate on investigating how MobilityFirst can benefit from SDN and optical networks. Detailed review on MobilityFirst mechanisms such as globally unique identifiers (GUID), global name resolution system (GNRS) and hop-by-hop transmission is given in Chapter 5.

1.3 Contributions

The first contribution of this dissertation is a survey on network innovation using OpenFlow, published in the IEEE Communications Surveys & Tutorials online journal [7]. To the best of our knowledge, this work was the first comprehensive survey to describe the difference between SDN and OpenFlow, as well as the capabilities, applications, challenges and deployments of SDN-based networks using OpenFlow.

The second contribution of this dissertation is OpenSec, an SDN-based framework that allows network operators to write human-readable security policies and implement them automatically in the network. Through this work, we reinforced the idea of moving middleboxes away from the main datapath of a LAN to improve scalability and reliability. Furthermore, we proposed a new policy specification language that is simpler than the ones proposed by the related work. Our work was published in the proceedings of IEEE GLOBECOM 2014 [9] and a journal

submission is currently under review.

The third contribution of this dissertation is the eXtensible Traffic Engineering Framework (XTEF), a WAN framework capable of creating dynamic, on-demand, end-to-end circuits in transport networks. XTEF uses OneSwitch, a WAN abstraction model, to expose the WAN as a single switch to external tenants. Using XTEF, the tenants can request bandwidth or latency guarantees and XTEF uses dynamic WDM tunnels to provision the network and allocate such demands. An implementation and evaluation of OneSwitch was published in the proceedings of IEEE/OSA OFC 2015 [16]. A description of the main challenges in WAN for scientific flows was published in the proceedings of the SDN Workshop for Scientific Computing at Super Computing 2015 [17]. The XTEF work was submitted to a conference and is currently under review [11].

The fourth contribution of this dissertation is the implementation of an OpenFlow-based control plane for MobilityFirst with intra-domain cut-through switching. The motivation for this framework is to perform efficient data transfer in MobilityFirst for flows with static end-points. To avoid the overheads of the routing layer, the proposed framework creates Ethernet-layer tunnels for improved efficiency and flow aggregation. This work was published in the proceedings of IEEE ANTS 2013. An extended version of the paper was also published in the Photonic Communication Networks Journal special issue consisting of selected papers from IEEE ANTS 2013.

Finally, the fifth contribution of this dissertation is to extend the MobilityFirst SDN framework to an inter-domain scale. To this end, we first model and solve an optimization problem to minimize the total transfer time of flows across domains. Next, we propose a routing framework that enables domain controllers to exchange topology information and communicate to create inter-domain tunnels. This is an

important contribution to the SDN field because it proposes a novel mechanism to create inter-domain tunnels that is scalable and uses fewer messages than current protocols such as the label distribution protocol (LDP). This work was submitted to a conference and is currently under review.

1.4 Dissertation organization

The rest of this dissertation is organized as follows. First, Chapter 2 provides detailed background on SDN and OpenFlow. Next, Chapter 3 introduces OpenSec, a framework that enables network operators to implement security policies in a campus network. After that, Chapter 4 describes XTEF, a WAN provisioning framework. After that, Chapters 5 and 6 address the topic of cut-through switching in MobilityFirst using OpenFlow at intra-domain and inter-domain scale. Finally, we draw our conclusions and describe future work in Chapter 7.

Chapter 2

Software Defined Networks and OpenFlow

2.1 Introduction

A recent approach to programmable networks is the Software Defined Networking (SDN) architecture. SDN consists of decoupling the control and data planes of a network. It relies on the fact that the simplest function of a switch is to forward packets according to a set of rules. However, the rules followed by the switch to forward packets are managed by a software-based controller. One motivation of SDN is to perform network tasks that could not be done without additional software for each of the switching elements. Developed applications can control the switches by running on top of a network operating system, which works as an intermediate layer between the switch and the application. Another motivation is to move part of the complexity of the network to the software-based controller instead of relying only on the hardware network devices.

OpenFlow [8] was proposed to standardize the communication between the switches and the software-based controller in an SDN architecture. The authors identify that it is difficult for the networking research community to test new ideas in current hardware. This happens because the source code of the software running on the switches cannot be modified and the network infrastructure has been "ossified" [8], as new network ideas cannot be tested in realistic traffic

settings. By identifying common features in the flow tables of the Ethernet switches, the authors provide a standardized protocol to control the flow table of a switch through software. OpenFlow provides a means to control a switch without requiring the vendors to expose the code of their devices.

Table 2.1: Example OpenFlow-compliant switches.

Switch Company	**S**eries
Arista	Arista extensible modular operating system (EOS), Arista 7124FX application switch
Ciena	Ciena Coredirector running firmware version 6.1.1
Cisco	Cisco cat6k, catalyst 3750, 6500 series
Juniper	Juniper MX-240, T-640
HP	HP procurve series- 5400 zl, 8200 zl, 6200 yl, 3500 yl, 6600
NEC	NEC IP8800
Pronto	Pronto 3240, 3290
Toroki	Toroki Lightswitch 4810
Dell	Dell Z9000 and S4810
Quanta	Quanta LB4G
Open vSwitch	Software switch. Latest version: 1.10.0

OpenFlow was initially deployed in academic campus networks [8]. Today, at least nine universities in the US have deployed this technology [18]. The goal of OpenFlow was to provide a platform that would allow researchers to run experiments in production networks. However, industry has also embraced SDN and OpenFlow as a strategy to increase the functionality of the network while reducing costs and hardware complexity. Table 2.1 shows a list of several OpenFlow-compliant switches available in the market. The Open Networking Foundation (ONF) [19] was founded in 2011 by Deutsche Telekom, Facebook,

Google, Microsoft, Verizon, and Yahoo to promote the implementation of SDN and OpenFlow-based networks. Currently, ONF has more than 95 members including several major vendors.

OpenFlow networks have specific capabilities. For example, it is possible to control multiple switches from a single controller. It is also feasible to analyze traffic statistics using software. Forwarding information can be updated dynamically as well and different types of traffic can be abstracted and managed as flows. These capabilities have been exploited by the research community to experiment with innovative ideas and propose new applications. Ease of configuration, network management, security, availability, network and data center virtualization and wireless applications are those that have been investigated the most using OpenFlow. They have been implemented in different environments, including virtual or real hardware networks and simulations. Researchers have also focused on evaluating the performance of OpenFlow networks and on proposing methods to improve their performance.

OpenFlow offers great opportunities for network innovation but it also faces challenges. The fact that the availability of the network depends on a single controller at a given time, creates scalability and availability problems. There are security concerns regarding the fact that all the network information is contained in one single server. Compatibility issues must also be taken into consideration. Questions remain about future directions of OpenFlow research as well. We discuss the extension of this technology to network-layer devices such as IP routers, as well as the deployment of OpenFlow in wide area networks (WAN).

In this chapter we describe the capabilities, applications, deployments and challenges of OpenFlow networks in local and wide area environments. We also describe SDN and alternative standards such as ForCES [20]. We explain how

OpenFlow has received major attention among SDN technologies but we also point out the difference between SDN and OpenFlow.

We begin by giving a background of programmable networks and describing SDN in Section 2.2. We explain the OpenFlow specification in Section 2.3. Then we present the capabilities of OpenFlow networks in Section 2.4 and we survey how they have been exploited in different applications in Section 2.5. We describe deployments of OpenFlow-based networks in Section 2.6. Next we discuss studies that have evaluated the performance of OpenFlow in Section 2.7. Then we discuss the challenges faced by OpenFlow in Section 2.8. We conclude by proposing future research directions in Section 2.9.

2.2 Background of programmable networks

In this section we present several contributions to programmable networks prior to SDN and OpenFlow. One of the first approaches was SOFTNET [21], an experimental multihop packet radio network that introduced the idea of adding commands to the contents of each packet. The goal was to modify a network node during operation time, using commands written in the SOFTNET language. The motivation of the authors in creating this network was to enable experiments with different network protocols. SOFTNET was deployed as a proof of concept. There were no further large scale deployments, but the idea behind it was the motivation for Active Networks [22, 23].

The main idea of Active Networks (AN) was to allow packets to contain programs that could be executed by the network devices that they traversed. The concept of active network is due to the fact that switches perform computations on the data of the packets flowing through them and the users can inject programs

into the network [22]. A survey on AN research is available in [24]. Although AN became an active field of research, it ultimately failed at being widely used. Recently, NetServ [25] was proposed as ActiveNetworks 2.0. The authors argue that NetServ contains all the necessary elements to be deployed.

SOFTNET and ActiveNetworks did not use software components to control the network devices. The programmability of the network was achieved by adding source code to the payload of the packets. More recent approaches proposed separating the control plane from the data plane by moving the first one to general purpose servers. We describe SoftRouter [26], ForCES [20] and finally we focus on OpenFlow [8]. They are all based on software defined networking architectures, where the network devices are controlled by software components.

2.2.1 Software Defined Networking

The difference between SDN and the previous approaches is that a software component running on a server or a CPU is added to the architecture of the network. In SDN, the software component is responsible for the control plane of the network. This is why we say that SDN decouples the control and data planes, as this distinction was not as clear in previous approaches.

One important feature of SDN is its ability to provide a network wide abstraction. Keller et al. [27] discuss the idea of the "platform as a service" model for networking. According to the authors, it is a common trend to decouple the infrastructure management from the service management. In this model, the underlying physical network and the topology are hidden to the user. Instead, the abstraction presented to the user is a single router. According to them, the customer is mostly interested in being able to configure policies and defining how packets are handled. We will see during the rest of this chapter that a large number of publications aim

at hiding the complexity of the network and providing an easier way to configure a service. Using names instead of IP addresses, or high level policies instead of access control configuration files are examples of this abstraction.

Network operating system is a key concept in SDN. It comes from the idea of abstracting the complexity of the underlying network. Lazar [28] explains how an early approach to programmable networks introduced the term of kernel in terms of networking. The idea was precisely to draw a parallel between the network operating system and the typical operating system. In an operating system, the abstraction includes the hardware components of the CPU. In a network, the abstraction hides the topology and the network devices. Therefore, the network operating system is responsible for the abstraction provided by SDN to its users.

Another important advantage of SDN is that it enables innovation and flexibility. If the control and data plane are managed by a hardware network devices, there is little room for innovating and experiment, as the software or firmware of those devices cannot be easily modified. Instead, by having access to a software component to manage the control plane, many ideas can be explored.

2.2.2 Standardizing the communication between the control plane and the data plane

SDN provides network-wide abstraction to the user and any software-based technique can be used to manage the control plane. However, we have not discussed how is the communication between the control and data plane standardized. Next we describe how several researchers have proposed to standardize this communication.

One early proposal is the IEEE P1520 Standards Initiative for Programmable Networks Interfaces [29]. The authors identify the need of abstracting the complex-

ity of the network to the user as well as the necessity of a programming interface to define the network. They also discuss the need of having a protocol to access the network elements.

The SoftRouter architecture [26] allows dynamic binding between the network element running the data plane and the control element (software-based). This architecture was proposed for network-layer devices that can be controlled by standard purpose servers. The software component does not need to be wired to the network device and a network element can have more than one control element across the network.

ForCES (Forwarding and Control Element Separation) [20] was created by the Internet Engineering Task Force (IETF). ForCES was proposed to standardize the way that controlling elements communicate with network elements. However, this standard did not experience widespread adoption by the vendor community. The Internet Research Task Force (IRTF) has also undertaken efforts regarding SDN. The Software Defined Networking Research Group (SDNRG) [30] aims to identify SDN approaches that can be used in the nearby future, as well as to identify future challenges. It also aims at providing a forum to SDN researchers [31].

OpenFlow [8] came next and was based on the same motivation: how to standardize the communication between the control plane and the data plane. It describes how software applications can program the flow table of different switches. OpenFlow quickly became an active research topic and we describe it in detail in the next section. Before, we briefly compare ForCES and OpenFlow.

The IETF documented the differences between ForCES and OpenFlow [32]. According to this document, both standards decouple the control and data planes and they both standardize the communication between the two planes. Regarding the architecture of the network, one difference can be found between ForCES and

OpenFlow. ForCES defines networking and forwarding elements and how they can communicate with each other. The architecture of the network remains unchanged. On the other hand, OpenFlow modifies the architecture in the sense that data plane elements become simple devices that forward packets according to rules given by the control element. ForCES allows multiple control and data elements within the same network and the logic can be spread through all the elements. OpenFlow aims at having a centralized control plane.

Due to the emergence of OpenFlow as the SDN architecture that has received major attention, we focus this chapter on network innovation using OpenFlow.

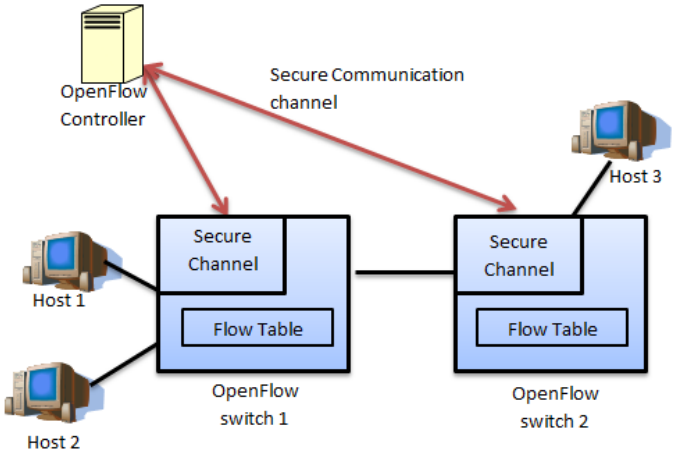

Figure 2.1: OpenFlow components.

2.3 OpenFlow specification

The OpenFlow specification describes an open protocol to allow software applications to program the flow table of different switches. An OpenFlow architecture consists of three main components: an OpenFlow-compliant switch, a secure channel and a controller, as shown in Fig 2.1. Switches use flow tables to forward packets. A flow table is a list of flow entries. Each entry has match fields, counters

and instructions. Incoming packets are compared with the match fields of each entry and if there is a match, the packet is processed according to the action contained by that entry. Counters are used to keep statistics about packets. The packet can also be encapsulated and sent to the controller.

The controller is a software program responsible for manipulating the switch's flow table, using the OpenFlow protocol. The secure channel is the interface that connects the controller to all switches. Through this channel, the controller manages the switches, receives packets from the switches and sends packets to the switches. An OpenFlow-compliant switch must be capable of forwarding packets according to the rules defined in the flow table. Figure 2.2 shows a high level description of how a network device processes a packet. First, the communication between the switch and the controller is possible through flow table rules. Internally, a switch uses Ternary Content Addressable Memory (TCAM) and Random Access Memory (RAM) to process each packet.

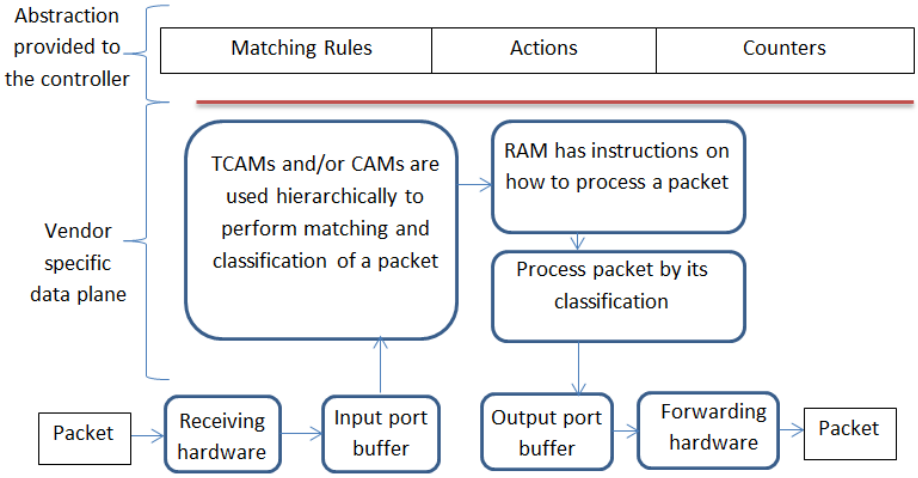

Figure 2.2: Elements of an OpenFlow-compliant switch.

Different versions of the OpenFlow protocol specification are available. The first version was the OpenFlow version 0.2.0 released in March, 2008. Versions 0.8.0

and 0.8.1 came next in May, 2008. Version 0.8.2, released in October, 2008, added the Echo Request and Echo Reply messages. Then, version 0.8.9 was released in December, 2008. It included IP netmasks, additional statistic information and several other updates. OpenFlow 0.9 was released in July, 2009. Finally, OpenFlow version 1.0, the most widely deployed version, was released in December, 2009. Next, we focus on versions 1.0.0 [33], 1.1.0 [1], 1.2 [34] and 1.3.0 [35], as previous versions are now deprecated. A detailed list of changes included in every version is available in the OpenFlow 1.3.0 specification document [35].

Table 2.2: Match fields of a flow table entry in an OpenFlow 1.0.0 switch.

Ingress Port
Ether src
Ether dst
Ether type
VLAN id
VLAN priority CoS
IP src
IP dst
IP Proto
IP ToS bits
TCP/UDP src port
TCP/UDP dst port

2.3.1 OpenFlow 1.0.0

Currently, the most widely used specification is the version 1.0.0. A switch supporting OpenFlow specification 1.0.0 uses 12 header fields present in the header and payload of the Ethernet packets coming into the switch. Table 2.2 shows all the header fields.

A packet can be matched to a particular flow entry in the flow table by using one or more header fields of the packet. A field in the flow table can have the

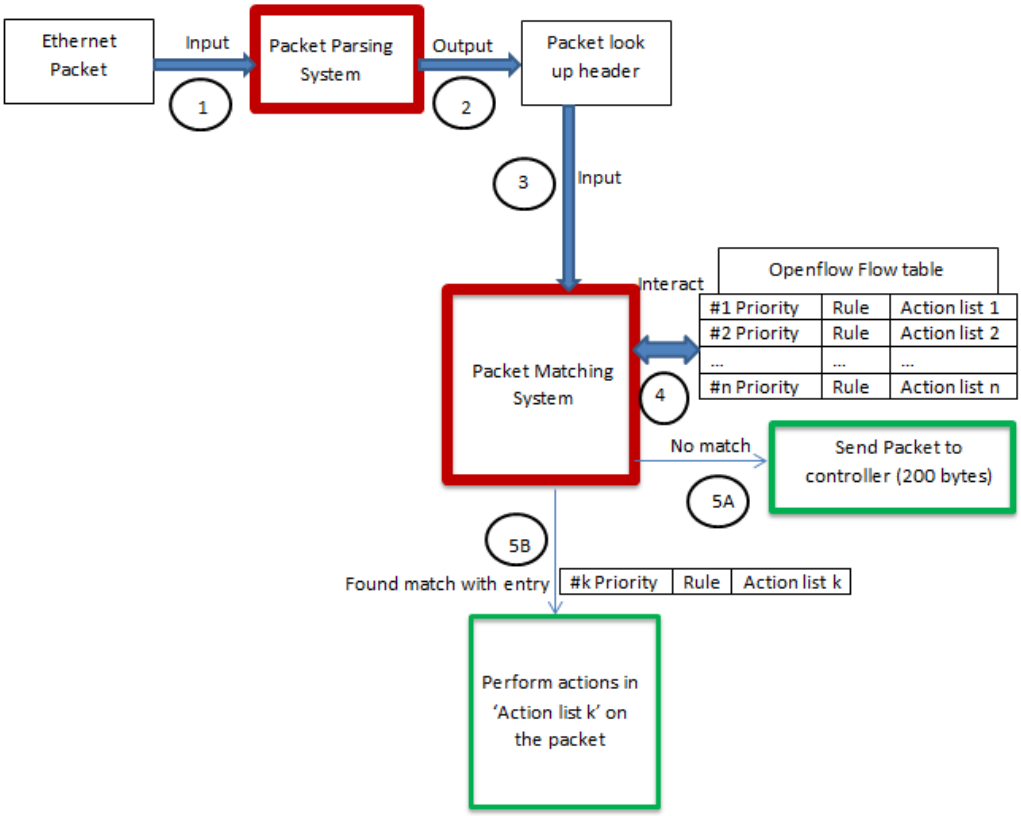

Figure 2.3: How a packet is processed and forwarded in an OpenFlow 1.0.0 switch.

value of ANY and it will match all packets. If the forwarding table is implemented using Ternary Content Addressable Memory (TCAM), ANY can be implemented in the switch hardware using the third masking state of the TCAM.

In Fig. 2.2 we showed the main elements of an OpenFlow switch. Figure 2.3 shows the details of the data plane in an OpenFlow 1.0.0 switch. In step 1, the Ethernet packet entering the switch goes to a packet parsing system. In step 2, the header fields are extracted and placed in a packet lookup header, as they are used for matching purposes. In step 3, the packet lookup header generated is sent to the packet matching system. In step 4, the packet lookup header is compared to the rules defined for each flow entry in the OpenFlow flow table. Note that the

flow entries in the table are present in the descending order of priority. Therefore, the comparison of the packet lookup header is done starting from the first flow entry on the flow table. If a match is found, the actions in the matched flow entry are performed on the packet (step 5B). Otherwise, the first 200 bytes of the packet are sent to the controller for processing (step 5A).

Table 2.3: Match fields of a flow table entry in an OpenFlow 1.1.0 switch.

Ingress port
Metadata
Ether src
Ether dst
Ether type
VLAN id
VLAN priority
MPLS label
MPLS EXP traffic class
IPv4 src
IPv4 dst
IPv4 proto / ARP opcode
IPv4 ToS bits
TCP/UDP/SCTP src port. ICMP Type
TCP/UDP/SCTP dst port. ICMP Code

2.3.2 OpenFlow 1.1.0

In the OpenFlow 1.1.0 specification, a switch contains several flow tables and a group table, instead of just one single flow table, as in OpenFlow 1.0.0. Figure 2.4 shows the main components of the OpenFlow 1.1.0 switch. The match fields are also different, as shown in Table 2.3. We have highlighted in bold the added cells. The metadata field is used to pass information between the tables as the

packet traverses through them. It is a register used to carry information between the tables. The Multiprotocol Label Switching (MPLS) fields are used to support MPLS tagging.

The processing of a packet entering the switch has changed as there are multiple flow tables available in the switch. The flow tables in the switch are linked to each other through a process termed as pipeline processing.

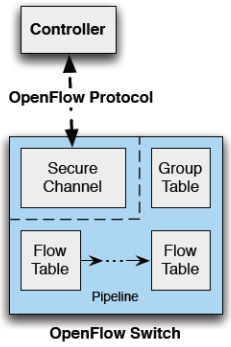

Figure 2.4: Components of an OpenFlow 1.1.0 switch. Source: [1].

Pipeline processing involves a set of flow tables linked together to process the packet coming in. When the packet first enters the switch, it is sent to the first table to look for the flow entry to be matched. If there is a match, the packet gets processed there and if there is another table that the particular flow entry points to, the packet is then sent to that flow table. This happens until a particular flow entry does not point to any other flow table.

The flow entries in the flow tables can also point to the group table. The group table is a special kind of table designed to perform operations that are common across multiple flows. This means that actions belonging to a set of flows are grouped together. Also, the set of flows is controlled to perform various actions collectively under a single group. Complex forwarding actions such as multipath

and link aggregation are enabled through the group table.

Finally, specification 1.1.0 introduces instructions instead of actions. Previously, an action was associated to each flow table entry. That action could be to forward the packet or to drop it, as well as processing it normally as it would be in a regular switch. Instructions are more complex and they include modifying a packet, updating an action set or updating the metadata.

2.3.3 OpenFlow 1.2

The OpenFlow specification version 1.2, was released in December 2011 and it includes a few major features. First of all, support to IPv6 addressing is added. Matching could be done using the IPv6 source and destination addresses. Another important feature supported is the possibility of connecting a switch to multiple controllers concurrently. The switch maintains connections with all the controllers and these can communicate with each other to do hand overs. Having multiple controllers provides faster recovery during failure and it is also possible to achieve load balancing.

2.3.3.1 OpenFlow 1.3.0

The OpenFlow specification version 1.3 was released in June 2012. Some of the improvements over version 1.2 are listed next. It is possible to control the rate of packets through per flow meters. Also, auxiliary connections between the switch and the controller have been enabled. Another improvement is that cookies can be added to the packets sent from the switch to the controller and specific durations field have been added to most statistics. A complete list of changes is available in the specification's document [35].

Table 2.4 compares specifications 1.0.0, 1.1.0, 1.2 and 1.3.0.

Table 2.4: Comparison of OpenFlow specifications.

Specification	**1**.0.0	**1**.1.0	**1**.2	**1**.3.0
Widely deployed	Yes	No	No	No
Flow table	Single flow table	Multiple flow tables	Multiple flow tables	Multiple flow tables
MPLS matching	No	Yes	Yes	Yes, bottom of stack bit added
Group table	No	Yes	Yes	Yes, more flexible table miss support
IPv6 support	No	No	Yes	Yes, new header field added
Simultaneous communication with multiple controllers	No	No	Yes	Yes, auxiliary connections enabled

2.3.4 Implementing applications using OpenFlow

In order to run applications on top of a single controller to manipulate the flow table of a switch, a network operating system is required (see Fig. 2.1). It acts as an intermediate layer between the OpenFlow switch and the user application. The network operating system communicates with the switch using the OpenFlow protocol and notifies the application of network events. Nox [36], Beacon [37] and Maestro [38] are examples of network operating systems. Recently, Big Switch released Floodlight [39], an open source Java based controller. Foster et al. [40] proposed Frenetic, a network programming language that simplifies the development of applications on top of network operating systems. NEC proposed Trema [41] to develop OpenFlow applications using Ruby and C. Finally,

DreamersLab developed Node.flow [42], a package to build a JavaScript based flow controller using Node.js [43]. Table 2.5 summarizes comparative data for the OpenFlow controllers that we have mentioned.

There are at least four possibilities to implement OpenFlow-based applications. First, an OpenFlow-compliant hardware switch can be used. We have provided a list in Table I. It is also possible to implement an OpenFlow-compliant software-based switch using Open vSwitch [44, 45]. A third option is to deploy virtual networks using Mininet [46], a virtual environment developed by the Stanford University that can be used to simulate multiple hosts in virtual network within one single host machine. Finally, a NetFPGA platform can be used. It consists of a PCI card that provides four 1G Ethernet ports, static RAM and other network functionalities [47]. The NetFPGA is also available with four 10G Ethernet ports.

Since physical and virtual switches can be used to deploy an OpenFlow network, it is important to note some similarities and differences between them. The advantage of a virtual switch is definitely the cost. Open vSwitch can be downloaded for free and it can be installed using commonly used virtual machine tools. A virtual switch performs the operations shown in Fig. 2.2 and Fig. 2.3 in software. Therefore, its main drawback is the performance. Hardware based switches perform data plane operations faster.

It is worth mentioning that debugging network applications is not a common technique yet. However, a first prototype of a debugger has recently been proposed by Handigol et al. [48].

Using OpenFlow, experimental and production traffic can share the same OpenFlow switch. The action of a flow table entry of an OpenFlow switch can be to send the packet to the switch data path. On the other hand, a different flow entry can be defined for experimental traffic. This way, experimental traffic can

Table 2.5: OpenFlow controllers.

Controller	**L**anguage	**C**reated by	**C**omments
NOX	C++	Nicira Networks	NOX was donated to the research community in 2008. It has several branches at Stanford University, such as classic NOX, new NOX and POX. New NOX is the version that will be further developed. POX supports Python and it is used for educational or research applications [49].
Beacon	Java	Stanford University	Supports both event-based and threaded operation. Mostly used for research and experimentation [37].
Maestro	Java	Rice University	Licensed under licensed under LGPL v2.1. Not as common as other controllers such as NOX [50].
Floodlight	Java	Big Switch Networks	Forked from Beacon and extended for enterprise usage. Apache-licensed [39].
Trema	Ruby and C	NEC	Supports Linux applications only [41].
Node.Flow	JavaScript	DreamersLab	Works on top of Node.js, a platform built on Chrome's JavaScript runtime [42, 43].

be tested without interfering with the production traffic [8]. In order to further enhance this, Sherwood et al. proposed FlowVisor [51]. Using this technique, it is possible for several single controllers to share the control of a switch. A centralized OpenFlow- based controller "slices" the network and acts as an intermediate layer between the switch and all the OpenFlow controllers that manipulate the switch.

2.3.5 OpenFlow: a specification, a protocol or an architecture?

OpenFlow can be viewed as a specification when it is in the context of an OpenFlow switch. An OpenFlow switch is achieved by implementing the requirements specified in the OpenFlow specification, in the device. For instance, in the OpenFlow specification, it is required that the switch has to support the flood action on the packets belonging to a particular flow. The flood action floods the packet using the normal pipeline of the switch [1]. Whether or not to implement this feature is a decision made by the vendor, but an OpenFlow switch must provide this functionality.

The OpenFlow protocol deals with defining the format of the messages passed between the control plane and the OpenFlow switch through the secure channel. The format of the messages has to be understood as well as generated by both the entities. This standard format of message passing is defined in the OpenFlow protocol. In fact, the OpenFlow protocol is part of the OpenFlow specification and it applies to the OpenFlow control plane as well as to the OpenFlow switch.

Finally, OpenFlow is viewed as architecture in the context of an entire network. In an OpenFlow network, OpenFlow switches are being controlled by one or more OpenFlow controllers. Such a network can be viewed as supporting the OpenFlow architecture.

It is important to keep in mind that the data plane implementation of the switch is vendor specific. As long as a switch can communicate with an OpenFlow controller, the data plane can be implemented differently by each vendor. Therefore, the fact that two switches are OpenFlow-compliant does not make them equal. Actually, not all switches implement all the features of the OpenFlow specification. It is possible that an OpenFlow-based application works using one switch but does

not work using a different switch.

2.3.6 OpenFlow and SDN

Since OpenFlow has become the most popular SDN technology, some consider these terms as synonyms. However, it is important to note the difference between them. SDN consists of decoupling the control plane from the data plane, whereas OpenFlow describes how a software controller and a switch should communicate in an SDN architecture. SDN gives the user an abstraction of the network-wide state and OpenFlow abstracts a network component. As an analogy, an operating system provides a system-wide abstraction, just like SDN provides a network-wide abstraction. On the other hand, just like the operating system communicates with hardware through drivers, OpenFlow can be considered a driver to communicate a single controller and a network component.

As an SDN technology, OpenFlow networks have specific capabilities that we describe next.

2.3.7 Capabilities of OpenFlow

OpenFlow architectures allow centralized control of the network, software-based traffic analysis, dynamic updating of forwarding rules and flow abstraction. In this section we describe these capabilities and we give examples that illustrate how they can be exploited.

2.3.8 Centralized control of the network

One important capability of an OpenFlow network is that the controller has network-wide knowledge of the system. Several OpenFlow switches can be connected to a single controller and it is then possible to make decisions in a

centralized manner. Instead of having several network devices with a limited knowledge of the network, a single controller can take decisions based on its knowledge of a broader part of the network.

One example of this is Ethane [52], an architecture proposed for managing the network of an enterprise. The key idea is to create a centralized policy that is managed by the controller. The switches become simple machines that forward and drop packets according to the rules defined by the controller. Using this architecture, it is possible to manage the network policies using high-end names. Routing decisions are also considered by the policy and finally, it becomes easier to bind a packet to its origin.

Another example of this capability deals with link failure recovery. In a traditional network, each switch has a limited knowledge of the network. When a link fails, then routes get adjusted at each switch until new routes are found. In an OpenFlow network, a centralized controller can find new paths in a much faster and easier way.

A comparison between the Path Computation Element (PCE) [53] architecture and OpenFlow is worth being mentioned when discussing this capability. Path computation in large and complex networks may require cooperation between different domains. The PCE architecture was proposed to address these challenges. A PCE is an entity that is capable of computing a network path or route based on a network graph [53]. A PCE architecture is not fully centralized. However, a cooperation between different entities does exist. Nevertheless, it can also occur that an entity does not have visibility over another element. Therefore, the knowledge of the network is not full. In OpenFlow-based networks, the controller usually has a broader knowledge of the network and therefore the control of the network is centralized. On the other hand, OpenFlow controllers do not cooperate

together as it happens in a PCE architecture. Giorgetti et al. [54] propose OpenFlow and PCE architectures to control wavelength switched optical networks.

To illustrate the difference between PCE and OpenFlow architectures, we describe how the OSCARS [55] (On-Demand Secure Circuits and Advance Reservation System) project provides a PCE module [56]. Through this module, researchers can deploy PCE elements in the network in a distributed manner. Therefore, it is possible to perform path computation without using a single centralized point. If we compare this to an OpenFlow testbed, we will find that researchers deploy the code on top of an OpenFlow controller and all computations are performed from there.

Another centralized approach towards network management is the Bandwidth Broker (BB) architecture [57]. A BB consists of one or more servers that perform network functionalities such as quality of service (QoS), policy enforcement or admission control. The data plane communicates with the BB modules. The advantage of this architecture is that part of the complexity is assumed by the BB and minimal configuration is required in the network device. This architecture can be used at the edge of a network to control bandwidth allocation.

2.3.9 Software-based traffic analysis

Software-based traffic analysis is a powerful capability of OpenFlow networks. This capability greatly enables innovation, as it is possible to improve the capabilities of a switch using any software-based technique. Traffic analysis can be performed in real time using machine learning algorithms, databases and any other software tool.

As an example, a distributed denial of service attack (DDoS) detection method is proposed in [58] and it heavily relies in traffic analysis. The method is based

on retrieving traffic data on periodic intervals and using self organizing maps to classify traffic as normal or malicious. Because the traffic analysis is done by software, there are more possibilities of using advanced features to perform the analysis, such as neural networks.

Another application of this capability is source address validation. Yao et al. [59] proposed checking the source address of each new flow. When a switch forwards a packet to the controller because it does not match any rule in the flow table, the controller can validate whether or not that source address corresponds to a valid flow.

2.3.10 Dynamic updating of forwarding rules

Another capability of OpenFlow networks is that they allow dynamic updates of forwarding rules. All kinds of changes in the topology can be performed in real time, based on the decisions taken by a software controller. No human interaction is required. This is possible because the controller can modify the flow table entries at any time.

In [60], the controller is notified of a link failure and it modifies the entries of the flow table to re-route the traffic. By doing this, the network can react to link failures without requiring any action by the network administrator. The authors also suggest that the controller can automatically allocate more or less bandwidth according to the traffic load, to save energy.

Another application of this capability is load balancing. The controller can assess the load of several servers and dynamically change the forwarding rules to make sure that the load is properly balanced. Handigol et al. [61] proposed Plug-n-Serve, a load balancer that can dynamically add new servers to the cluster without interrupting the service.

2.3.11 Flow abstraction

Finally, networks using OpenFlow abstract all traffic as flows. For each flow there is an entry in the flow table. For each entry, different rules can be defined. One flow could be all traffic using one specific TCP protocol. Another could be all packets travelling between two defined MAC addresses or all data with one IP address destination. One could also define a non standard header to identify traffic of a specific entry. This allows managing different kinds of flows using the same control element.

Merging packet and circuit networks in a single infrastructure has been studied by several authors and it relies on this capability. Packet and circuit networks are treated as two different flows but they can be managed by the same controller.

In the next section we survey how the capabilities described above have been exploited in OpenFlow-based applications.

2.4 OpenFlow-based applications

In this section we survey studies that use OpenFlow for different kinds of applications. Ease of configuration, network management, security and availability are examples of these applications. OpenFlow has also been used to achieve network and data center virtualization, as we describe next.

2.4.1 Ease of configuration

OpenFlow-based applications can simplify the configuration of the network. Common approaches include access control lists and configuration files whose administration is time consuming and can lead to errors. By using SDN, it is possible to use software to take care of this. Yamasaki et al. [67] proposed using OpenFlow

Table 2.6: Comparison of security applications using OpenFlow.

Publication	**P**roblem approached	**D**escription of the solution	**I**mplementation	**S**DN capabilities exploited
Suh et al. (CONA) [62]	DDoS attack detection	Frequency and pattern of requests are analyzed to detect DDoS attacks.	NetFPGA-OpenFlow switches	Traffic analysis and dynamic rules updating
Braga et al. [58]	DDoS attack detection	Statistic information in the flow table is used to classify traffic as normal or malicious.	Simulation of a NOX based network.	Traffic analysis and centralized control.
Chu et al. [63]	DDoS attack detection	Locator/ID separation protocol (LISP) is used to identify authorized and malicious sources.	Small network with one controller and two OpenFlow switches. Specialized hardware simulates DDoS attacks.	Traffic analysis and dynamic rules updating
Liu et al. [64]	Covert channel protection	The controller uses a second software node that filters authorized communication.	Simulation of a network using a virtual OpenFlow switch.	Dynamic rules updating and centralized control.
Yao et al. (VAVE) [59]	Source address validation	The controller analyzes traffic and calculates the flow path to decide if the source address is valid.	Simulation of a network using a virtual OpenFlow switch.	Traffic analysis and dynamic rules updating.
Jafarian et al. [65]	Moving target defense	The controller periodically assigns different virtual IP addresses to hosts to hide the real IP addresses to an intruder.	Simulation using Mininet.	Centralized control, dynamic rules updating.
Gutz et al. [66]	Traffic isolation	Network slices are defined through a programming language instead of using network-level techniques.	A tool was developed to test whether traffic isolation was correct	Centralized control

to manage the VLANs of a campus network. They describe how the number of VLAN ids is limited and how the configuration tasks are time consuming. In their approach, the controller analyzes incoming traffic and detects if the communication should be allowed or not, based on virtual group ids (GID) instead of VLANs. Using this approach, the number of VLANs limitation is overcome and the configuration of the network is simplified.

Several authors have addressed how to ensure consistent network updates using SDN. Reitblatt et al. [68] describe how to provide abstract operations that allow updating rules across the entire network in one fell swoop. In another paper, Reitblatt et al. [69] describe how updating network policies can lead to inconsistencies when packets are processed by both the old and the new policy. The authors note that achieving per-packet and per-flow consistency is critical to avoid inconsistencies and they describe techniques to implement both features. Also, Katta et al. [70] introduce algorithms that trade time against TCAM space in order to do the updates in an efficient manner. McGeer [71] proposes a network update protocol as well. His method uses boolean formulas and it ensures that flows are treated consistently. As an example, if a ruleset 1 is updated to a ruleset 2, the protocol ensures that the packets that were being processed using ruleset 1 are conserved, then the update takes place in all routers and finally the packets are released and processed by ruleset 2. Finally, Ghorbani et al. [72] propose a method to migrate virtual machines in a consistent manner and respecting bandwidth requirements. The authors have implemented an algorithm that outputs the order in which virtual machines must be migrated in order to ensure that no inconsistencies occur.

As we described earlier, Casado et al. [52] proposed Ethane, an SDN architecture explicitly designed to simplify the management of the network in an

enterprise. Ethane relies on the idea that the network policy should be known by the controller and enforced in all switches. The main requirement is that all communications between two hosts require explicit permission. Instead of creating configuration files for all the switches in the network, these devices are kept simple and the rules are managed by the controller. An implementation of an Ethane switch in hardware is described in [73].

Some common points can be extracted from these studies. We mentioned in Section II that a user is interested in defining policies and configuring how their packets are forwarded. Here we notice that the studies by Reitblatt et al. [69] and by Casado et al. [52] focus on simplifying the creation of policies and hiding how these policies are implemented underneath. The study by Yamasaki et al. [67] provides another way of creating VLANs in such a way that the user must not deal with troublesome configuration files.

2.4.2 Network management

Deploying OpenFlow-based networks has also motivated research on OpenFlow management infrastructures. These studies aim at simplifying network management through OpenFlow. Mattos et al. [74] implemented a user friendly interface that allows the user to manage the network. Their implementation is based on NOX. Several applications are developed on top of that network operating system and a web based interface is provided to the user. Also, a multiagent system is capable of autonomously perform management.

Gibb et al. [75] propose an architecture in which network appliances (middleboxes) are not located at points of the topology that are traversed by plenty of traffic. They argue that these chokepoints are not suitable for middleboxes, as performance and correctness issues arise. Instead, they suggest using processing

units in waypoints of the network. An OpenFlow switch, located at the choke-point, is capable of routing to the processing units only the traffic that needs to be processed by the middlebox. By doing this, less traffic traverses the network appliances and a much simpler hardware is used at the chokepoint of the network.

Defining and implementing network policies has also been addressed using OpenFlow. Voellmy et al. [76] propose Procera, a controller architecture and a high level network control language that can be used to reactively define network policies. Regarding implementation, Fergusson et al. [77] propose an OpenFlow-based method to perform policies delegation in SDN networks. Their idea consists of creating delegation trees, where each path can be managed by different network administrators. The authors create hierarchical flow tables that can be used to delegate policies. An incoming packet is matched to these policies and processed accordingly.

Finally, an innovative way of managing IP multicast in overlay networks was proposed by Nakagawa et al. [78]. The authors propose using OpenFlow instead of a more common approach such as Internet Group Management Protocol (IGMP). Two important contribution of their approach are eliminating periodical join/leave messages and making use of multipath in the layer-2 network.

Outsourcing network functionality is another interesting innovation to simplify the network management. Gibb et al. [79] propose Jingling, an architecture that allows adding functionality to a network in an outsourced manner. Feature providers can be located anywhere outside the network. Policies defined how feature providers must be used and a network controller maps the policies to the feature providers. Following the idea of having services outside the network, the idea of Networking-as-a-Service (NaaS) has emerged. Raghavendra et al. [80] propose using OpenFlow to manage networks in such a way that they are ready to

user services provided as NaaS.

In this section, we notice that the common trend is to exploit how OpenFlow can dynamically update the forwarding rules. Having a network-aware controller allows the network manager to dynamically forward traffic according to specific needs. Once again, we also note how several studies simplify the creation of network policies.

2.4.3 Security

OpenFlow has also been used to create applications that provide security to the network. Table 2.6 compares the problems approached, the solutions proposed and the infrastructures used to test the implementations.

Methods to detect DDoS using OpenFlow have been proposed recently [62, 58, 63]. Suh et al. [62] proposed a content oriented networking architecture. This approach relies on creating flows based on the identity of the client and the type of content requested. A DDoS attack is detected when the server that provides a given content type receives more requests than expected, based on a pre-defined range. Chu et al. [63] proposed a method that analyzes the frequency of traffic. If a threshold is exceeded, then the controller considers that a DDoS attack is happening and it starts dropping packets. Finally, as we mentioned earlier, Braga et al. [58] proposed a method that gathers traffic information and uses self organizing maps to classify the traffic as normal or malicious.

Liu et al. [64] proposed an SDN architecture where nodes with different levels of security clearance can exchange communication. The OpenFlow controller sets up the rules so that traffic is authorized only when the requester has a higher security clearance than the receiver.

Yao et al. proposed VAVE [59], an OpenFlow-based architecture designed to

validate the address of all incoming packets. When the switch receives a packet that does not match any rule, the packet is sent to the controller and the source address is validated. If spoofing is detected, then a rule is created to stop that traffic.

Jafarian et al. [65] propose a moving target defense (MTD) technique using OpenFlow. The proposed defense assigns virtual IP addresses to hosts and the controller maps virtual addresses to physical addresses. This is performed once and again, in an unpredictable way such that the attacker cannot identify which host is behind each IP address.

Finally, traffic isolation has been studied by Gutz et al. [66]. The authors argue that current traffic isolation techniques such as VLANs increase the complexity of the network configuration. They propose creating network slices at a higher level. Under their approach, a network programming language should be able to create this slices to isolate traffic. This way, slices are defined at a high level and then forwarding rules are automatically added to the switches.

When it comes to security, we notice how the researchers heavily rely on the ability of processing data in the controller. In all these publications, some kind of intelligence is added to the switch through the controller. For example, Braga et al. [58] use self organizing maps, which could not be implemented on regular switches. Also, Yao et al. [59] exploit the idea that, since a given packet must be analyzed by the controller, then a more rigorous address validation can be performed. Once again, in the study by Gutz et al. [66], we note how more capability is given to a higher layer. In this case, it is about isolating network traffic using a programming language. This is a common trend in SDN: how to allow a user to perform network tasks without needing full access to the network topology.

Table 2.7: Comparison of network virtualization applications using OpenFlow.

Publication	Problem approached	Description of the solution	Implementation
Simeonidou et al. [81]	Packet and circuit network integration	An OpenFlow controller is integrated with a GMPLS controller	No implementation provided
Das et al. [82]	Packet and circuit network integration	An OpenFlow controller is integrated with a GMPLS controller	Prototype network using NetFPGA switches that emulates a WAN
Das et al. [83]	Packet and circuit network integration	An OpenFlow controller is integrated with a GMPLS controller	Fully functional hardware based network. Used as a proof of concept for a demonstration.
Das et al. [84]	Application aware aggregation and traffic engineering in a circuit-packet network	The capabilities of SDN are exploited in a circuit-packet network to provide application aware routing.	Hardware based network used to emulate a WAN
Das et al. [85]	Complexity of IP/MPLS control plane	The MPLS data plane is controlled by OpenFlow instead of the traditional IP/MPLS control plane.	Open vSwitch and Mininet are used to emulate a WAN
Ferkouss et al. [86]	Flexibility of MPLS nodes	An OpenFlow controller is used to dynamically modify MPLS nodes	Hardware implementation that exploits the pipelining of OpenFlow 1.1.0.
Kempf et al. [87]	Supporting MPLS forwarding in OpenFlow 1.0.0	Additional match fields are added to the flow entry format and MPLS actions are added to the OpenFlow 1.0 specification	NetFPGA-OpenFlow switches
Sharafat et al. [88]	MPLS implementation complexity	The centralized control capability is exploited to implement MPLS-TE and MPLS-VPN in a simpler way than the traditional approach	Physical and virtual switches supporting the MPLS section of OpenFlow 1.1 and simulation using Mininet

2.4.4 Availability

OpenFlow-based applications have focused on providing availability to the network as well, including load balancing and fault tolerance. Load balancing is

a commonly used technique to distribute a working load between two or more nodes. This improves the availability of a network since the system can support one or several single failures. Fault tolerance refers to the property of a system to continue operating when a failure occurs.

Load balancing: Handigol et al. proposed Plug-n-Serve [61], a load balancer for unstructured networks that attempts to reduce the response time by taking into consideration the load of the servers and the congestion of the network. The proposed method displays the load of the network in real time. The software running on the controller takes the load of the network and servers into consideration and decides where to direct the traffic. Using this solution, it is also possible to add new servers to the cluster and the software will dynamically detect them and add them to the load balancing. An improved version of Plug-n-Serve, Aster*x was also proposed in [89]. Aster*x runs on the Global Environment for Network Innovations (GENI) infrastructure and it is used at a much larger scale than Plug-n-Serve.

Wang et al. [90] argue that Plug-n-Serve works by reactively creating forwarding rules for incoming requests. They proposed a proactive approach, based on wild cards. They divide the entire client address space into different rules. These rules forward the traffic to specific servers. The controller knows what percentage of traffic should be handled by each server and it creates the rules so that the expected loads are respected. We can see that the approach by Wang proactively creates the rules to make sure that each server handles the required percentage of connections. This requires a smaller number of rules than the approach used by Plug-n-Serve, which improves its scalability. On the other hand, Plug-n-Serve takes into consideration the load of the server and the network and does not require a specific percentage of traffic for each server and it is more flexible, since each client can be handled individually.

Fault tolerance: Sharma et al. [91] and Staessens et al. [60] have explored fault tolerance using OpenFlow. In [91], the authors describe how failure recovery can be implemented using OpenFlow. They explain how the controller can dynamically change the routing rules when a failure is detected in a link. In [60], experiments are designed to analyze if an OpenFlow based network can recover from a link failure. The authors argue that carrier grade networks must be able to recover in less than 50 ms. The experiments show that restoration is successful but that the dependency on the centralized controller makes the goal of 50 ms challenging to achieve.

Another way of ensuring availability is to verify that there are no configuration errors that might cause a disruption. Khurshid et al. [92] propose VeriFlow to check network invariants in real time. This includes loops in the routing tables, unavailable paths and other problems that can be identified before deploying the network. Moreover, the authors are interested in doing this in real time. VeriFlow sits between the controller and the switch and monitors the communication between these two parts. By modelling the network as a graph, network invariants are checked in the order of hundreds of microseconds.

Porras et al. [93] propose a policy enforcement mechanism that is also based in analyzing the forwarding rules that are added to or deleted from the flow table. The author introduce FortNOX and they aim at performing role based authentication and security constraint enforcement. The application checks for conflicting rules after every update of the flow table. When two rules incur in a contradiction, then the rule defined by the user with the highest security clearance is kept.

These studies have some common trends. First, the capability of dynamically updating forwarding rules is heavily exploited. Load balancing is performed

based on the ability of the controller to alter the forwarding rules. The fact that the controller is network-aware is also helpful. In the studies by Sharma [91] and by Staessens [60], finding new paths after a failure occurred is easily done in a centralized manner, since the topology is known. Traditionally, this kind of recovery is done by decisions taken by switches that are not network-aware and a centralized method simplifies this task.

Network virtualization using MPLS and GMPLS: Network virtualization is another research area where OpenFlow has been applied. Circuit and packet switched networks are typically managed using separate infrastructure and this is costly. Several authors have proposed OpenFlow-based architectures that could be used to manage both packet and optical circuit networks using the same infrastructure [81, 82, 83, 84, 94]. Azodolmolky et al. [94] provide a good explanation on how OpenFlow and GMPLS can be used together as an integrated control plane. This approach relies on the fact that packet and optical circuit networks can be managed as different flows in the switch's flow table. In order to manage both flows, a GMPLS controller is integrated to the standard OpenFlow controller. The OpenFlow controller is responsible for managing the flow table. However, when a flow corresponds to traffic over an optical circuit, then the GMPLS controller takes care of the routing decisions and a flow entry containing the forwarding action and the required wavelength is added to the flow table. This way, switches can handle two kind of flows, one for circuit networks and one for packet networks.

MPLS and GMPLS have also been used in other applications. Kempf et al. [87] add an extension to OpenFlow 1.0 that allows a switch to forward MPLS on the data plane. Das et al. [85] proposed using MPLS in the data plane but OpenFlow in the control plane instead of the traditional IP/MPLS control plane. El Ferkouss et al. [86] argue that OpenFlow can be used to "deossify" an MPLS

architecture. They show how an MPLS node can play multiple roles for different MPLS domains, which provides greater flexibility to the nodes. Sharafat et al. [88] implement MPLS-TE and MPLS-VPN using an OpenFlow controller to show that centralized control makes the implementation easier. Table 2.7 compares the different applications that use OpenFlow to virtualize networks using MPLS and GMPLS. Centralized control, dynamic rules updating and flow abstraction are the most commonly exploited capabilities for these applications.

The studies that we have mentioned exploit the circuit switching capability of GMPLS and not the VLAN-switching capability. In summary, the research direction regarding GMPLS and OpenFlow is to simplify the creation of end-to-end circuits. Das et al. [95] discuss why GMPLS has not been as successful as expected in the control plane and how combining it with software defined networking is a more suitable approach.

Data center virtualization: Similar to network virtualization, virtualizing data centers using OpenFlow has also been an active research area. SDN architectures have been considered to meet the requirements of a data center: efficiency, agility, scalability and simplicity [96]. Al-Fares et al. [97] proposed Hedera, a dynamic flow scheduling method for data center networks. They proposed an OpenFlow-based architecture that can dynamically modify the flows according to the traffic load. The authors argue that this approach achieves a larger network utilization. Rotsos et al. [98] also use OpenFlow to dynamically virtualize the network. They argue that VLANs and MPLS can be used to create virtual networks in a static way. However, the network utilization can be optimised if the network virtualization is performed according to the traffic load.

Wide area network applications: A majority of studies have deployed their experiments in local area networks. However, some studies address the possibility

of deploying OpenFlow in a wide area network (WAN). First, in [85] the authors show that OpenFlow could be deployed in a WAN by emulating this kind of network. Studies such as [82, 84] show that OpenFlow could be used to control this type of network.

Bennesby et al. [99] propose an inter-domain routing solution using an OpenFlow architecture running on a NOX controller. The authors explain how the different autonomous systems (or domains) interact with each other through the Internet. They propose a routing scheme based on OpenFlow that would allow autonomous systems to communicate with each other.

Wireless applications: OpenRoads [100] was designed to enable research in mobile networks. It can be considered as the wireless version of OpenFlow. In this architecture, a flow visor [51] (as introduced in Section III) controls network devices through the SNMP protocol. Several controllers can be deployed on top of the flow visor. Details of the OpenRoads architecture are available in [101]. A deployment of OpenRoads in the campus of Stanford University is described in [102]. Other works using OpenRoads include [103, 104].

There are also other wireless applications that do not use OpenRoads. Huang et al. [105] proposed PhoneNet, an infrastructure which supports group communication among phones. A group of users can interact using their phones after a multicast address is created so that it can be accessed by all the members in the group.

Bansal et al. [106] propose OpenRadio, a design for a programmable wireless network data plane to automatize how devices' software is updated. They argue that software updates have become more frequent (there used to be a release every few years and now updates are available monthly). OpenRadio aims to providing an infrastructure to update base stations of wireless systems via software. Without

this approach, devices must be collected so that the software can be manually updated. This frequent hardware collection is expensive and network software updates are more adequate. As an example, they describe that in an urban area, there could be one device per block to provide adequate coverage. In this scenario, collecting the sensors every time an update must be installed would be prohibitively expensive. OpenRadio enables updating the devices without having to physically collect them.

Regarding wireless enterprise local area networks (WLAN), Suresh et al. [107] propose Odin, a prototype SDN architecture that simplifies client management in a WLAN. The network is given programmability and light virtual access points are introduced. These access points are managed from an OpenFlow controller.

Other applications: OpenFlow has also been used in other areas not listed above, such as routing and network congestion control. Liu et al. [108] proposed a method to control congestion using queuing systems and a centrally controlled network. Yap et al. [109] also consider network congestion, as well as bandwidth reservation and multicast. Nascimento et al. [110] proposed QuaqFlow, a Quagga implementation using OpenFlow. Quagga is a routing package that provides implementation of TCP/IP routing protocols. RouteFlow [111], an architecture that provides routing as a service, was proposed as an extended work of Quagga. Rothenberg et al. [112] proposed an OpenFlow-based approach that allows the introduction of advanced routing systems. This study was built by extending the earlier RouteFlow [111]. Egilmez et al. [113] proposed an architecture to provide routing for video streaming.

In the next section, we focus on larger-scale deployments rather than the applications themselves.

2.5 OpenFlow deployments

Deployments of OpenFlow-based networks mainly include campus networks and testbeds, as well as deployments undertaken by the industry.

Stanford University has deployed an OpenFlow-based network in one of its buildings. The network includes production, experimental and demonstration traffic. It connects approximately fifty switches and around 25 users, both wired and wireless. Details of the topology can be found at [114]. Other universities have also deployed OpenFlow-based networks. The full list is available at [18] and it includes Clemson University [115], Georgia Tech [116], Indiana University [117], Kansas State University [118], Rutgers University [119], University of Washington [120], University of Wisconsin [121] and Princeton University [18].

At a larger scale, the Global Environment for Network Innovations (GENI) [122] provides a research infrastructure where OpenFlow experiments can be conducted. The OpenFlow core of this network consists of several interconnected OpenFlow-compliant switches on both Internet2 [123] and National LambdaRail (NLR) [124] networks. The connection to the NLR network is achieved through HP6600 switches deployed at Sunnyvale, Seattle, Denver, Chicago, and Atlanta and through NetFPGA switches in Sunnyvale, Houston, Chicago, and New York [125]. Internet2 has OpenFlow-compliant switches installed in Los Angeles, New York, Washington DC, Atlanta [126]. Campus networks can connect to the GENI deployment to run larger scale experiments.

As of October 2012, Internet2 provides a nationwide 100G software defined network [127]. The network is currently operational for member institutions of Internet2. The deployment includes routers of the Brocade MLX family and related Brocade NetIron platforms, as well as Juniper Networks MX Series routers

Figure 2.5: Draft of the planned U.S. UCAN network using the Internet2 100G deployment. (Source: [2]).

[128]. It also provides a 100G Ethernet network and a 8.8 Terabit per seconds optical network. Internet2 will operate the U.S UCAN (United States Unified Community Anchor Network) program [2]. Their goal is to use this software defined network to provide a platform to interconnect research, educational and health care institutions. Figure 2.5 shows a draft of the expected deployment.

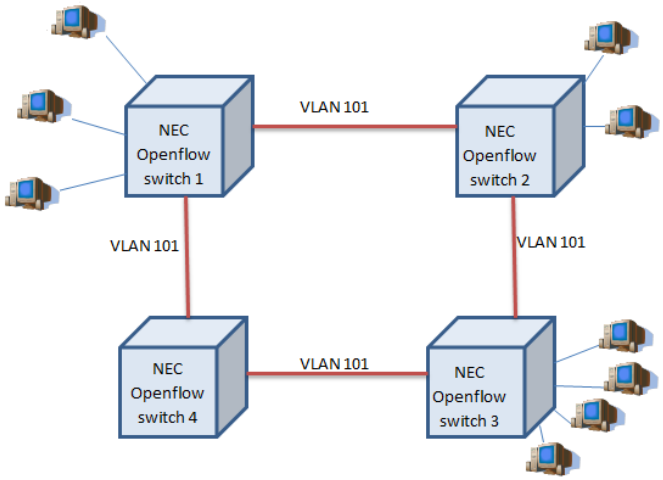

Figure 2.6: Topology of the ANI OpenFlow testbed.

The Energy Science Network (ESnet) [129] is funded by the Department of Energy (DOE) and operated at the Lawrence Berkeley National Laboratory. ESnet

has also deployed an OpenFlow testbed, originally funded by the Advanced Networking Initiative (ANI) [130]. ANI was an investment in next-generation technology infrastructure to speed of scientific discovery. ESnet operates two testbeds: the Long Island Metropolitan Area Network (LIMAN) and the 100G. The LIMAN is a 10G testbed. It includes four NEC IP8800 OpenFlow switches [131]. The OpenFlow network operates on the VLAN 101. There are two ways of running an experiment on the testbed. One option is to connect the controller directly to the OpenFlow switches through the management VLAN. The second option is to connect to the flow visor controller and getting a partition of the network to run the experiments. The first option requires the researches to reserve the testbed beforehand. The second option does not require any reservation of resources. The flow visor configuration file has to be sent to the administrator to get connected. The 100G testbed runs between the DOE Supercomputer centers in Argonne National Lab (Chicago) and NERSC (California) through a 100G dedicated network [132]. To deploy experiments using the 100G testbed, researchers must follow a proposal process that includes writing a 1-2 page proposal and demonstrating that the experiment is working in a small environment [133]. Figure 2.6 shows the topology of the ANI OpenFlow testbed.

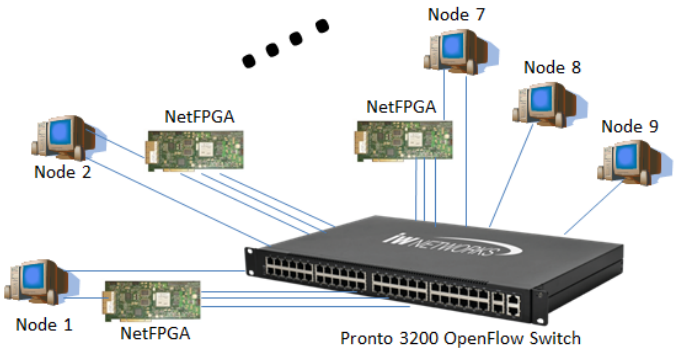

Figure 2.7: Topology of the ORBIT OpenFlow testbed.

Another smaller deployment is the Open Access Research Testbed for Next-Generation Wireless Networks (ORBIT) testbed [4], which is being developed and operated by WINLAB, Rutgers University. It is intended to be used to test and evaluate innovative protocols in real-world settings and it includes an OpenFlow-based network. The deployment consists of an OpenFlow-compliant switch Pronto 3290 connected to nine nodes. Out of the 9 nodes, 7 of them are connected to one NetFPGA each. Each of the NetFPGA is connected to the Pronto 3290 OpenFlow switch through four 3GbE connections. All of the 9 nodes are connected to the Pronto 3290 OpenFlow switch and they are connected to a control plane through which the nodes can be accessed through telnet/ssh sessions by the experimenter. Figure 2.7 shows the topology of the ORBIT OpenFlow testbed.

Similar testbeds have been deployed in Europe and Japan as well. Ofelia is a project funded by the European Union that provides an OpenFlow-based network with nodes in Belgium, Switzerland, UK, Spain, Germany, Italy and Brazil [134]. Also, the Dynamic Network System (DYNES) project [135], funded by the National Science Foundation (NSF), is exploring technologies such as OpenFlow to interconnect campus, regional and backbone networks. Other future deployments also include the Network Development and Deployment Initiative (NDDI) and the Open Science, Scholarship and Services Exchange (OS^3E) [135].

OpenFlow has also been deployed by several companies, as seen in the keynote lectures of the 2012 Open Networking Summit [136]. As an example, Google has deployed OpenFlow in the inter-datacenter backbone network that carries all the traffic between the different datacenters [10]. Currently, this network is completely OpenFlow based. According to the speaker, adopting OpenFlow has been the most significant change in networking in the company [137].

By surveying OpenFlow-based applications and deployments, we have iden-

tified some challenges faced by OpenFlow-based networks. We discuss these challenges next.

2.6 Performance of OpenFlow-based networks

We have surveyed different OpenFlow-based applications and deployments. Next we mention several studies that have designed experiments to evaluate the performance of OpenFlow architectures. We also discuss publications that propose alternatives to improve the performance of OpenFlow networks.

2.6.1 Measuring and modelling the performance of OpenFlow-based networks

Jarschel et al. [138] model an OpenFlow controller as a M/M/1 queuing system. This model allows obtaining results regarding the total sojourn time of a packet through the system. The model also captures the difference in terms of delay between a packet that is processed by the switch and a packet that must go to the controller. Also, the probability of dropping a packet because the controller is under high load is studied. The results show that the sojourn time depends largely on processing speed of the OpenFlow controller. Also, the authors are able to conclude that the processing time of the controller lies between 220 and 245 μs. Another interesting result shows that current controllers cannot handle a big number of flows in 10Gbps links.

Bianco et al. [139] compare the performance of OpenFlow switching, link layer Ethernet switching and network layer IP routing. Experiments include using packets of different sizes and comparing the results of single flows against multiple flows. In all the experiments, OpenFlow achieves good results in comparison to link layer Ethernet switching and network layer IP routing.

Levin et al. [140] address the following question: "How does distributed SDN state impact the performance of a logically centralized control application?" [140]. The authors argue that the SDN network control plane cannot be fully physically centralized because responsiveness, reliability and scalability issues arise. One possible solution is to have a distributed control plane where a logically centralized control plane operates. This design faces consistency challenges and the authors study how much inconsistencies in the global network view affect the performance of the network. The authors compare two applications: one is ignorant to possible inconsistencies and the other takes inconsistency into consideration when operating. This study concludes that optimality is significantly affected when inconsistencies are not considered and that the robustness of an application is increased when it is aware of the network state distribution.

Heller et al. [141] address two important questions regarding reliability, scalability and performance. First, they analyze how many controllers are needed in a network. Second, they discuss where in the topology should these controllers go. The authors introduce these questions as an important part of the controller placement problem. Regarding the number of controllers needed, the authors analyze the latency of different topologies and they observe that one controller is often enough to keep the latency at a reasonable rate. They also explain that, in general, adding k controllers reduces the latency by a factor of k. However, they also show examples where this is not the case and more controllers are required. Regarding the placement of controllers, they show how this decision can also affect the latency of the network. They also show that randomly selecting the location of the controller yields results that are far from optimal.

Finally, the performance of OpenFlow has also been evaluated in the optical networks domain. Liu et al. [142] evaluate the performance of an OpenFlow-

based wavelength path control in transparent optical networks. They study two different approaches for lightpath setup (sequential and delayed) and two ways of lightpath release (active and passive). The experimental setup includes four OF-PXCs connected in a mesh topology, with one OpenFlow switch and one client node attached to each OF-PXC. A photonic cross-connect (PXC) devices switches optical signals in an all-optical device. The results show that a path between two clients (thus traversing two switches) can be provisioned faster using the sequential approach. Also, releasing a path can be done faster if the active approach.

2.6.2 Improving the performance of OpenFlow-based networks

Several authors have also proposed modifications to OpenFlow or alternative ways of using it to increase the scalability, reliability or performance of the network.

Yeganeh et al. [143] propose Kandoo, a framework that aims at reducing the number of events that are received at the control plane of the network. To do this, two layers of controllers are used. The upper layer maintains the network-wide state. The bottom layer consists of several controllers that do not know the network-wide state and that are not interconnected. The bottom layer handles most of the events and reduces the overhead at the upper layer. This framework also increases the scalability of an OpenFlow network.

At least two studies have proposed additional ways to take profit of a CPU being connected to the switch. Mogul et al. [144] propose software defined counters. Recall that an OpenFlow switch collects statistic data for each flow. The authors explain that this data is stored in the switch using application specific integrated circuits (ASIC). The propose keeping and processing information in a CPU, where more variable and flexible statistics could be processed. The study does not include implementation or simulation results, but the feasibility of

software defined counters is analyzed theoretically.

Lu et al. [145] also propose combining ASIC and CPU processing. The authors point out two limitations of current switches: a limited size forwarding table and a limited size packet buffer. They argue that a their approach relaxes these limitations by using a CPU. A prototype is developed and a 3.9Gb/s software forwarding throughput is achieved. Also, large TCP traffic bursts are absorbed without packet losses. The experimental setup consists of sending 50k bidirectional TCP flows among four servers.

Vanbever et al. [146] propose HotSwap, a system that enables correct and efficient upgrades of SDN controllers. The goal of HotSwap is to be able to change from one controller to another (when upgrading the controller is needed) without disrupting the network. They argue that stopping the old controller and starting the new one introduces delays and can also create errors in the network. HotSwap records relevant messages between the switches and the controller and bootstraps the new controller by replicating previous network events. By the time the new controller starts operating, the network state is the same as when the previous controller was operating.

2.7 Challenges of OpenFlow-based networks

OpenFlow deployments face several challenges that must be taken into consideration, including security [147], availability [60], scalability [148], reliability [149], expenditure [150] and compatibility [151].

2.7.1 Security

One principal challenge of an OpenFlow-based network is the dependence on the controller. The controller becomes a component with a critical knowledge of the network and a very attractive target for an attacker. Security measures must be considered to ensure the availability of the controller. At the same time, since this component has access to all the network, it must be strongly protected from intruders.

The channel between the controller and the switches can also be vulnerable. According to the OpenFlow specification, Transport Layer Security (TLS) can be used to secure the communication. However, this feature is not a requirement and it is also acceptable to communicate the controller and the switches using plain text traffic. TLS can then provide security to the channel, but its usage depends on the design of the network since it is not required.

The flow table is a component that could also present security risks, although there are no published vulnerabilities yet. It is possible to manage a flow table from two different controllers, where one of them is a production hardware and the other one is just experimental. Since the latter one will be subject to lower security controls, it is important to make sure that the consistency of the flow table remains and that a malicious update coming from one controller will not tamper other flow entries. Currently, the flow visor takes care of those considerations but since OpenFlow is a recent protocol, this needs to be kept in mind.

A centralized software-based controller can also have security advantages. In a distributed network, many vulnerabilities must be addressed in different protocols and different devices. Having a software controller outside of the data plane can simplify how security is enforced, as there is plenty of expertise on securing

servers through hardening instead of securing network devices.

2.7.2 Availability

The dependence on the controller is also a challenge regarding availability. An OpenFlow-compliant switch is capable of forwarding packets using cached rules. However, the communication with the controller is eventually needed for any kind of modification of the rules. One advantage of a traditional, distributed network architecture is that if a switch fails, the availability of the network can be maintained. In an OpenFlow network, the communication with the controller must be ensured. As we mentioned in the previous subsection, the controller becomes a single point of failure.

How to handle the delay needed to create new flows is also a challenge. When an OpenFlow switch receives a packet that does not match any rule in the flow table, then the first 200 bytes of the packet are sent to the controller. After this, the controller can install a new forwarding rule. Therefore, the delay to process the first packet is larger. If this delay is too large, then the availability requirements of a network might not be met.

2.7.3 Scalability

The controller can also become a bottleneck. If too many packets must be forwarded to the controller, then performance issues can occur. A well designed network should ensure that the most part of the traffic can be handled by the switches without needing to forward data to the controller. It is also important to assess whether the controller will become a bottleneck when the number of nodes grows. As we discussed in Section VII, authors have addressed this challenge while evaluating the performance of OpenFlow. In particular, Heller et al. show how a

single controller is usually enough to keep an acceptable latency. They also show that introducing k controllers reduces the latency by k [141].

OpenFlow-based architectures also face two important scalability challenges: a limited flow table size and hardware constraints. First, the number of flows that can be contained in the flow table is limited. It is still a challenge to handle a very large number of flows using an OpenFlow-compliant switch. Manipulating packets at the control plane is slow as well. Therefore, end-to-end traffic control is hard to implement if many different flows must be manipulated. Second, there are hardware limitations on the speed at which flows can be added. For these two reasons, it is still unclear if OpenFlow deployments can be used to control the core of a network. Currently, OpenFlow is being used at the edge of a network instead.

2.7.4 Survivability

The dependency on the controller also creates reliability issues. One example can be found in [60]. In this OpenFlow-based network, a link failure is reported to the controller and a new path is found. According to the results, the network recovers successfully but not quickly enough. The authors explain that the expected recovery time is not met because of the time lost contacting the controller. A common requirement by carriers is to achieve a network recovery in less than 50 seconds. In the study by [60], this goal is not met.

On the other hand, a centralized control also has advantages regarding network recovery. In a distributed network, recovering from a broken path can be a slow process. However, an OpenFlow controller is network-aware and it can find the new path faster.

A multipath proposal for OpenFlow addresses how to recover faster from failures. This proposal includes a fast reroute support, where backup flows can be

installed in advance. If the switch detects that a specific port has lost connectivity, then the backup flow is installed. This is a proactive way of dealing with link failures and it has the advantage that the controller does not need to be contacted immediately after the failure.

2.7.5 CAPEX and OPEX

It has been debated whether OpenFlow can reduce the capital and operational expenses (CAPEX and OPEX) of an organization.

OpenFlow adopters argue that by moving the complexity to the software-based controller, network devices become simpler and therefore, cheaper. This would reduce the CAPEX. However, OpenFlow also has limitations and advanced hardware is still required to operate a network. It does not seem likely that network switches and routers will become simple commodities in the short term. Also, ensuring the availability of the control plane can increase the CAPEX. It is important that the controller remains reachable even in case of a failure in the data plane. Achieving this could increase the costs of a deployment.

A similar trade-off occurs for OPEX. We have discussed several studies that simplify the network configuration and management. Certainly, OpenFlow can be used to reduce the number of human based configuration tasks that are time consuming and error prone. This reduces the OPEX. On the other hand, moving the complexity of the network to the software control plane requires work. Project administrators, software developers, testers, debuggers and other costs are examples of expenses that must be incurred in an OpenFlow-based deployment. Therefore, it is not clear either whether OpenFlow greatly reduces the OPEX.

2.7.6 Compatibility

Another important challenge for OpenFlow deployments is that the network operating systems support specific versions of the OpenFlow specification. Currently, most of them support OpenFlow 1.0.0. Even though OpenFlow 1.1.0 has been available for several months, the network operating systems do not support specific features of the newer version. The challenge is then to upgrade both the OpenFlow specification and the software of each network operating system.

This compatibility issue also applies to the network devices, whose software must be updated to meet the requirements of new OpenFlow specifications. For instance, in the HP ProCurve switches series, modifying the packet header fields (for example: IPv4 destination address) in the switch hardware is not supported. But, it is possible to do the same in the switch software which is a slower path for processing. Therefore, it is likely that switch vendors would fine tune their hardware to support additional features in the switch hardware to improve efficiency. This updating process must be taken in consideration when new versions become available.

User developed applications face compatibility issues as well. We have shown how there are significant differences between specifications 1.0.0 and 1.1.0. Another example is that version 0.8.9 became deprecated when version 1.0.0 was available. Therefore, it is important to consider if applications running under version 1.0.0 will still work on version 1.1.0 or if all affected developments must also be updated. This scenario could occur again in further releases.

Finally, we believe that compatibility among controllers should also be taken into consideration. Currently, multiple network devices perform switching and routing in a standardized way. However, if the devices are controlled by software-

based controllers, then standardization should be achieved too. Controllers from different domains should use the same protocols to ensure that the communication is possible between hosts in different domains.

Next we conclude this chapter by discussing the future research directions in OpenFlow-based networks.

2.8 Conclusions and future directions

SDN is a promising technology for enabling advanced functionality in programmable networks. Our survey paper [7] is, in our opinion, the first one to discuss the capabilities, application, deployments and challenges of SDN/OpenFlow-based networks. We also explained and compared the OpenFlow specifications. Below, we identify future research directions in OpenFlow-based networks.

First of all, applications have been developed in areas such as security, ease of configuration, availability, network and data center virtualization, wireless applications and others. Currently, a majority of the surveyed applications consist of small, simple networks with some OpenFlow switches and hosts. Only a small number of studies demonstrate their work in a WAN. In [85], the authors emulated an OpenFlow-enabled WAN, but this is an exception to the majority of studies. Whether OpenFlow can be used in WAN deployments or not is still an open question. Studies show that OpenFlow could be used to control a WAN ([82, 84, 74]). However, scalability and performance experiments have not been conducted yet.

Second, we observe that OpenFlow switches have been used as a multi-layer network device. This technology was first proposed to control Ethernet switches. However, OpenFlow has also been used in routing ([90, 98, 109, 110]), IP address

validation ([59]) and MPLS control ([81, 82, 83, 84, 85, 86, 87, 88]). This shows that OpenFlow can be used at multiple layers. Future directions include tighter integration of OpenFlow features with routers and MPLS switches to reduce their complexity and cost.

Third, we find an open problem in the design of OpenFlow architectures. So far, mostly all applications and deployments use only one controller to manage all the switches. Distributed architectures with more than one controller could be used to address some of the challenges such as availability or reliability [141]. In fact, a vast majority of networks contain duplication as a means to ensure the availability of the system. We believe that the possibility of communicating controllers in the OpenFlow 1.2 specification ([34]) is an opportunity to deploy this kind of architecture. Coordinating tasks across multiple controllers and using them during normal and failover conditions are tasks for future investigations.

Fourth, we believe that most studies do not involve real hardware but use virtualization tools such as Mininet [46] and Open vSwitch [45]. Also, the number of hosts is small in most of the applications. Scenarios such as Ethane [52], where validation includes real hardware and up to 300 hosts are not very common. Realistic hardware simulations would also yield better results regarding the advantages and disadvantages of using OpenFlow in real networks. Using testbeds such as those described in this chapter is a good way to strengthen the validation of new applications.

Finally, it is important to mention that data center virtualization is one of the active areas that has received a lot of attention in the industry. The deployment of OpenFlow by Google [136] in one of their backbone networks and active participation of the Open Networking Foundation are good examples of the interest of industry in OpenFlow. Integrating OpenFlow into such large scale real-world

applications is an important future direction.

In conclusion, SDN is one of the transformational technologies to affect the networking vendor community in the last decade and exhibits tremendous scope for future research and deployment. Since the publication of our survey paper [7], other surveys on SDN have also appeared [152, 153, 154].

In the next Chapter, we show how SDN and OpenFlow can be used to simplify campus networks management by proposing OpenSec, a framework that enables policy-based management at campus scale.

Chapter 3

Campus scale: Policy-based security management using OpenSec

3.1 Introduction

With the advent of SDN, efforts to automate and simplify network operation have become popular [155, 52, 156]. In SDN, the complexity of the network shifts towards the controller and brings simplicity and abstraction to the network operator. As we move away from manual configuration at each device, we get closer to automated implementation of network policies and rules. SDN decouples the control plane from the data plane and migrates the former to a logically centralized software-based network controller. More complex network-control applications can thus be implemented at the controller and exploit the fact that they are network-aware due to the centralized nature of the control plane.

In this chapter we show how SDN can be used to implement policy-based security in small networks such as local area networks (LAN) or campus networks. To this end, we propose OpenSec, an OpenFlow-based network security framework that allows campus operators to implement security policies across the network. To motivate this work, suppose a campus operator needs to mirror incoming web traffic to an intrusion detection system (IDS) and e-mail traffic to a spyware detection device. Our goal is to leverage SDN to allow the operator to write a high-level policy to achieve this, instead of having to manually configure each device.

Furthermore, suppose the IDS detects malicious traffic and the sender needs to be blocked from accessing the network. Instead of having the operator configure the edge router to manually disable access to the source, we are interested in blocking the sender automatically.

Because OpenSec provides an abstraction of the network, the operators can focus on specifying simple and human-readable security policies, instead of on configuring all the devices to achieve the desired security. OpenSec consists of a software layer running on top of the network controller, as well as multiple external devices that perform security services (such as firewall, intrusion detection system (IDS), encryption, spam detection, deep packet inspection (DPI) and others) and report the results to the controller. The main goal of OpenSec is to allow network operators to describe security policies for specific flows. The policies include a description of the flow, a list of security services that apply to the flow and how to react in case malicious content is found. The reaction can be to alert only, or to quarantine traffic or even block all packets from a specific source.

We have built OpenSec taking three design requirements into consideration. First, policies should be human-readable. Simplicity is one of the main goals of our framework and although current work has focused on creating human-readable policies [76, 156, 157], we argue that there is still room for improvement to make the policy languages human readable. Second, data plane traffic should be processed by the processing units (network devices, middleboxes or any other hardware that provides security services to the network). When the controller is responsible for all tasks it becomes a bottleneck and the solution does not scale well. In OpenSec, the controller is subject to a low workload and is responsible for implementing policies and modifying forwarding rules based on the security alerts received from the processing units. Third, the framework should react to security alerts

automatically to reduce human intervention when suspicious traffic is detected.

To demonstrate the benefits of using OpenSec, we show two use cases. First, we demonstrate how OpenSec can be used in a campus network to implement network control for residential housing. To do this, we meet the requirements of network usage established by the Carnegie Mellon University campus. Second, we show how OpenSec can be used in a campus network to deploy a Science demilitarized zone (DMZ) to enable higher throughput for scientific data transfers.

Our evaluation shows that OpenSec scales well because the delay in reacting to alerts remains constant when the traffic rate increases. We also show the benefits of moving the middleboxes away from the data path in terms of achieved throughput. Finally, we compare OpenSec with existing solutions such as Procera [76], CloudWatcher [157] and Fresco [158] and show that the performance of the proposed framework is equal or better than similar works.

Our contributions in this chapter are as follows:

1. We create a simple, human-readable language to automatically implement network security policies.

2. We give a first step towards automated, policy-based reaction to security alerts using OpenFlow.

The rest of this chapter is organized as follows. In Section II survey related work and in Section III we motivate our work. Next, in Section IV we describe the OpenSec framework and its main components. After that, we describe the operation of OpenSec in Sections V and VI. In sections VII and VIII we describe the two use cases. In Section IX we evaluate the framework in terms of scalability and performance and we compare OpenSec against similar work in Section X. Finally, we conclude in Section XI.

3.2 Related work

3.2.1 Policy-based management without SDN

A significant amount of work has focused on policy specification [159], policy refinement [160, 161, 162, 163], conflict detection [164, 165] and policy analysis [166] in networks. Policy-based management (PBM) has also been applied to network management [167] and security [168].

Agrawal et al. [166] provide an overview of how policy-based management can be applied to networked systems. In particular, they explain how Policy Management for Autonomic Computing (PMAC) can be applied to network management. In a nutshell, PMAC is a generic policy middleware that supports extensive and flexible policy languages. Also, a Policy Definition Tool (PDT) should be provided to allow users to create and modify policies. Finally, an automated manager is responsible for collecting policies and implementing them.

Rubio-Loyola et al. [163] propose a a method to refine policies in policy-based management systems. Policy refinement allows to derive low-level enforceable policies from high-level guidelines. The authors provide a list of steps needed to convert high-level goals into low-level policies and describe a framework that supports all the required steps.

Charalambides et al. [164] address conflict resolution in PBM, a crucial aspect when managing a system using policies. Indeed, as the authors point out, when several policies coexist it is likely to encounter that two or more policies give a different output for the same input. This study addresses the problem of conflict resolution when using policies to provide Quality of Service (QoS).

OpenSec is similar to these methods in that it proposes a centralized system capable of receiving policies as input and analyzing them, checking for conflicts

and implementing them. In this work, however, we provide a detailed explanation of how policies can be converted into OpenFlow messages to update the forwarding rules dynamically. Also, the policies used in OpenSec are low-level specifications because they already include a list of OpenFlow matching fields that should be used. Thus, the main contribution of OpenSec is the automated administration of processing units and dynamic reaction to security alerts using SDN, as opposed to deriving low-level policies from high-level goals.

3.2.2 Policy-based network management using SDN

With the advent of SDN, the field of network management has evolved to become more dynamic [169, 170, 171]. Casado et al. proposed Ethane [52]. In Ethane, an operator creates a policy using the Flow-based Security Language (FSL) to create a high-level access control list. Ethane allows an operator to write an access control policy with good granularity while still using high-level language (for example, using "testing nodes" instead of a subnet mask). Although OpenSec does not focus on referring to network objects by name, we do provide a broader set of security services besides access control. Also, OpenSec includes a reactive component to security alerts. When anomalous traffic is detected, OpenSec can modify traffic rules as specified by the policy, which adds a reactive component missing in Ethane.

Foster et al. propose Frenetic [40], a programming language to program OpenFlow-based networks. Frenetic provides an interface to query traffic information. Frenetic can also be used to create a policy to react to network events. In our opinion, Frenetic focuses on simplifying how to program network events and how to retrieve traffic information. OpenSec focuses on hiding such complexity and allowing a security operator to work at a higher level, since it was designed to

implement and enforce security policies, rather than to provide another mechanism to handle events sent by the network switches.

Finally, Bari et al. [172] proposed PolicyCop, an autonomic QoS policy enforcement framework for SDN. This framework allows to specify service level agreements (SLAs) to implement and enforce QoS in an OpenFlow-based network. The step-by-step method used by PolicyCop to convert policies into flow rules is similar to that of OpenSec. However, our focus is on reacting to network security alerts instead of QoS violations.

3.2.3 Candidate frameworks for comparison against OpenSec

Among the proposed frameworks that convert policies to OpenFlow messages, three are candidates for comparison against OpenSec: CloudWatcher [157], Fresco [158] and Procera [76]. We describe these frameworks next and we compare them against OpenSec in Section 3.10.

Shin et al. proposed CloudWatcher [157], a security monitoring framework for the cloud that has several similarities with our work. Using CloudWatcher, a network operator can use a policy to describe a flow and describe which security services must be applied to it. For example, if traffic within a subnet must be subject to denial of service attack detection and intrusion detection, then a policy can be used to describe this. The authors focus on how the controller can find the optimal route to send the traffic to those processing units and the policy language is not described in detail. OpenSec goes one step further by allowing the operator to describe how to react in case malicious traffic is detected. Our focus is more on how to implement the policies instead of optimal routing decisions to find the processing units.

Fresco [158] is another OpenFlow-based security framework that exposes se-

curity modules to external users, who can in turn define security policies using such modules. To use Fresco, an operator must define the type, input, the output, the parameter, the action and the event. Fresco is also similar to Procera and Frenetic in the sense that it allows manipulating network events and handling them through pre-defined modules.

Voelli et al. proposed Procera [76], a "functional reactive programming" framework where a user can write a high-level policy to define how to handle network events. Just like Frenetic, Procera also aims to simplifying how to deal with network events. They have in common that they both seek a simpler interface to program the network and to react to network events. Also, Procera addresses an important topic: enforcing network policies. Our approach to enforce the implementation of security policies is based on the technique proposed in Procera. One thing that OpenSec does that is not considered in Procera and Frenetic is automated reaction to security alerts raised by external units. In OpenSec, the network operator defines a security level and the framework automatically modifies flow rules when malicious traffic is detected. In Frenetic and Procera, all reaction is defined by the code written by the operator. In a nutshell, Frenetic and Procera allow for a more granular control of the flow setup whereas OpenSec hides such events from the operator and reacts automatically.

Although other studies have addressed policy-based network administration using OpenFlow, as well as providing security through SDN, OpenSec's innovative approach allows operators to customize the security of the network using human-readable policies and to customize how the controller reacts automatically when malicious traffic is detected.

3.3 Motivation

The main goal of OpenSec is to be a human-friendly, dynamic and automated security framework. Next we describe three design requirements of our framework: moving middleboxes away from the main datapath, reacting automatically to security events and creating a simple policy specification language.

3.3.1 Moving middleboxes away from the main datapath

Sekar et al. show that, in a given enterprise, there are almost as many network appliances as there are routers [173]. Moreover, they point out how middleboxes do not favour network innovation, as they are closed system with no room for experimentation. Indeed, middleboxes are harder to update, upgrade or replace when compared to standard Ethernet switches.

The first goal of OpenSec is to move the middleboxes away from the choke points of the topology traversed by all traffic. Instead, these devices should be located outside of the main path between the LAN and the Internet and should act as security processing units that are visited only by the traffic that needs to be processed. Using a smarter OpenFlow-based control plane, the OpenSec should dynamically create rules to re-route traffic. This is important in terms of performance and reliability, since a L2 switch is easier to maintain, upgrade or replace in comparison to specialized hardware. When a specific flow is subject to deep packet inspection, for example, then the controller adds a rule that forces such traffic to visit the DPI processing unit.

To allow OpenSec to scale better, the processing units are responsible for analyzing the traffic and detecting malicious flows. Sampling traffic at the controller increases the chances of introducing a bottleneck and increases the complexity.

Also, the connectivity between switches and controller is usually of low bandwidth. In contrast, the data plane allows a faster bit rate and the processing units are optimized to handle big flows. In OpenSec, the controller remains listening to alerts and reacts to those alerts by deciding how to modify the traffic rules.

3.3.2 Reacting automatically to security events

The second goal of OpenSec is to enable automated reaction to security events. When a middlebox detects suspicious traffic, it issues a security alert. Traditionally, these alerts are received by a network operator who then decides how to react. With OpenSec we aim at automating this reaction so that the framework either blocks the traffic, or simply alerts the operator of the detected malicious traffic. The reason why this is feasible is because, in general, an operator can plan ahead of time how critical a flow is based on the service provided through that flow. In a production network, for example, an operator would want a denial of service attack to be stopped as soon as possible. In contrast, a testing environment could be less critical. Thus, we designed OpenSec to allow the operator to specify ahead of time what the automated reaction should be, so that in case of malicious traffic, the human participation is minimized.

3.3.3 Creating a simple policy specification language

The third goal of OpenSec is to provide a simple policy specification language to allow operators to redirect traffic to the middleboxes and to enable automated reaction. Among all related work, Procera (see Section 3.2.3) is probably the one that has focused most on designing human-readable policy definition. However, we argue that understanding a definition written in Procera is not straightforward. For example, the following instructions define a rule that allows all traffic:

Procera: proc *world* $\rightarrow$ **do**; **returnA:** λ *req* $\rightarrow$ *allow*

This statement still contains symbols that make it complicated to read. Instead, we aim at statements such as:

OpenSec: Flow: VLAN=192; **Service:** DPI; **React:** alert.

This OpenSec sample policy is described in Section 3.4.1. However, note that three components can easily be identified: the matching pattern, the security units that must be visited by this flow and the type of automated reaction.

Procera relies on reactive programming and its goal is different from OpenSec. In Procera, the goal is to program the network using policies and this includes handling events generated by the switches. In contrast, OpenSec does not communicate the network events to the end-user. Instead, they are automatically processed. This allows us to use a much simpler syntax to describe the flow, identify one or more services and specify how to react when malicious traffic is detected. Policies can be defined using the keywords shown in Table 3.1.

3.4 OpenSec components

OpenSec is an SDN framework capable of forwarding flows to security processing units based on policies and to automatically react to events raised by these middleboxes. Using this framework, security devices such as intrusion detection middleboxes, firewalls or encryption units can be removed from the main data path between the LAN and the Internet. OpenSec leverages a smart control plane

to allow end-users to direct only part of the traffic to these security units.

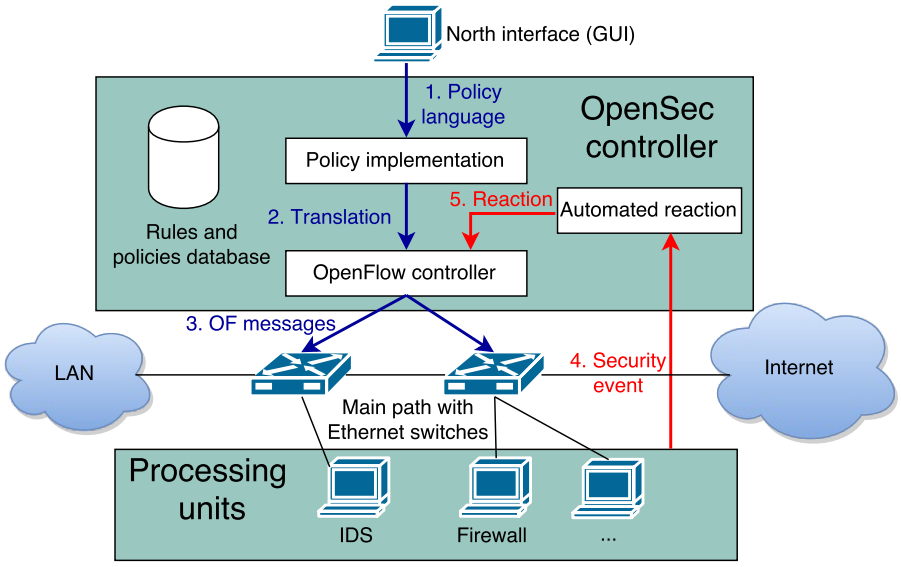

Figure 3.1: The OpenSec framework: Security functions are provided by the processing units; traffic is routed to each processing unit based on requirements given through security policies; the reaction to security alerts is automated.

In this section we describe the main components of the framework: a policy specification language, a northbound interface, a policy manager, a set of processing units, a security evento processor, an OpenFlow controller and a data repository. Then, in sections 3.5 and 3.6 we explain how these components interact with each other to implement the policies and to react to security alerts issued by the processing units.

3.4.1 Policy specification language

OpenSec's policy specification language allows to specify a matching pattern, a list of security units that should be traversed by such traffic and an automated reaction in case of receiving a notification from a unit (see Table 3.1). The matching fields correspond to those available in OpenFlow 1.0. The service corresponds to

any service ID registered by the processing units manager. Finally, the alert can be to alert only (via email), to block or to re-route traffic to a quarantine device.

Table 3.1: Syntax to create policies using OpenSec.

	Value	Description
Flow	inPort, VLAN, etherSrc, etherDst, ipSrc, ipDst, TCP-SrcPrt, TCP-DstPrt	Uses OpenFlow match fields to describe a flow
Service	Encrypt, IDS, DPI, spam, DDoS or any other service registered	Identifies a security service that should be applied to the flow
React	alert, quarantine, block	Determines how to react if the service reports malicious content

To illustrate how this language is used, we explain the example given in Section 3.3.3:

Flow: VLAN=192; **Service:** DPI; **React:** alert.

The policy above specifies that all traffic tagged with VLAN 192 should be re-routed to the DPI unit. Also, if the DPI middlebox informs of a suspicious sender, OpenSec must only alert the operator via e-mail. Several match fields and several units can be listed when specifying the policy.

3.4.2 Northbound interface

The current prototype of OpenSec includes a graphical user interface shown in Fig. 3.2. The list on the left shows all policies currently implemented and the

buttons on the left allow for adding or removing a policy. On the right side, the detailed policy is shown on top and the sources that have been blocked automatically using that policy are shown below. Finally, the operator can unblock a source. A similar GUI is provided to the user to show the information of each registered processing unit, similar to the data shown on table 3.2.

OpenSec should allow external applications to implement network security policies automatically. As a consequence, there is a need for a northbound Application Programming Interface (API) between OpenSec and the applications. The development of such an API is part of our future work.

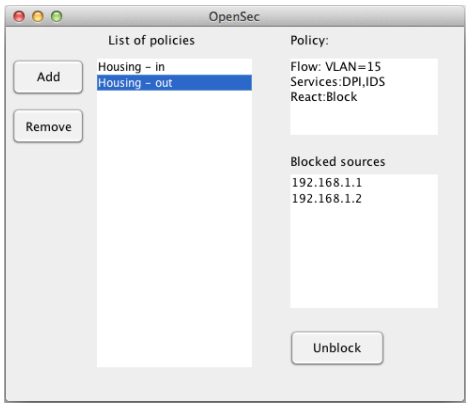

Figure 3.2: OpenSec's graphic user interface. This interface allows the network operator to add, remove and view policies. It can also be used to re-authorize blocked sources.

3.4.3 Policy manager

The policy manager is a core component of OpenSec. It is responsible for parsing new policies sent by the GUI and converting them to OpenSec objects. Next, it must implement the policy using the southbound interface component (controller). Finally it must also check periodically that the policy is implemented appropriately. The operation of the policy manager is explained in detail in Section 3.5.

3.4.4 Processing units

OpenSec relies on external processing units (or middleboxes) to analyze traffic. The units are customized to perform the required security scan, such as a firewall, an IPS or DPI. When suspicious traffic is detected, the processing unit issues an alert to the OpenSec controller so that actions can be undertaken based on existing policies. For this to work, all units must be known to the OpenSec controller.

OpenSec implements a processing units manager that collects all the registrations and creates a list of units and the location in the network where they can be found. In our current implementation, each unit is mapped to a service id (DPI, IPS), a switchID, an input port and an output port (see 3.2). This is all the data needed by OpenSec to manipulate the flow table of the devices in order to re-route traffic to the processing units.

Note that, if a processing unit is vendor-specific and this automatic registration cannot be implemented, a network operator can easily complete the information in the controller manually. In our current implementation, both automatic and manual registrations are supported.

Table 3.2: Registered security processing units for Fig. 3.1.

Service type	Switch ID	In-interface	Out-interface
DPI	1	25	26
DDoS	2	48	49
Encrypt	3	25	26

3.4.5 Security event processor

One of the most important features of OpenSec is the automatic reaction to security alerts. Usually, a network operator will react to an alert by either ignoring it or blocking the source of the suspicious traffic. In OpenSec, the network operator can

define such a reaction in advance using three possible choices: *alert*, *quarantine* or *block*. The security event processor is responsible for collecting the notifications issued by the processing units and modifying forwarding rules according to the policies involved. This component is explained in detail in Section 3.6.

3.4.6 OpenFlow controller

OpenSec uses OpenFlow to interface with the switches. To do so, a module running in the Floodlight controller [39] implements the required interfaces to listen to network events and communicate with switches. When a request is received from the policy implementer to push a new rule, this module is responsible for sending the message to the right switches. A more detailed explanation of how an OpenSec policy is converted to a list of OpenFlow messages is provided in Section 3.5.

Although the controller of OpenSec is centralized, multiple controllers can work together to increase the availability of the control plane. In the current implementation we rely on a single software to do this. However, it has been proposed to have multiple controllers working in a synchronized way [174, 175, 176, 177]. We do not address the implementation of a distributed control plane in this dissertation.

3.4.7 Data repository

OpenSec uses a data repository to store several pieces of information. First, all implemented policies are stored to check for conflicts when new policies are received, and also to know how to react to security events raised by the processing units. Similarly, all the information needed to route traffic to the middleboxes (device id, switch id, input port and output port) is also stored. Finally, OpenSec also records when hosts are blocked from accessing the network.

In the next section we describe how these components interact with each other to implement security policies.

3.5 Operation of OpenSec: policy implementation

In this section we describe how the policy manager of OpenSec converts a new policy created through the GUI into forwarding rules in the switches. Figure 3.3 provides an overview of the entire process.

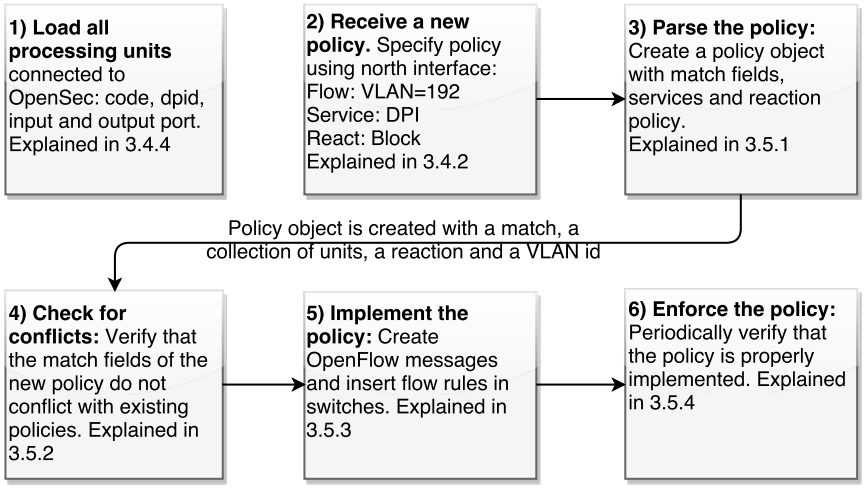

Figure 3.3: Steps needed to implement a policy.

3.5.1 Policy parsing

The policy parser receives the policy definition file from the northbound interface (GUI) component (as described in Section 3.4.2). Next, it converts it into a Policy object that can be processed by OpenSec to update flow rules in the switches. First, the parser builds a Flow object and sets the attributes based on the values given by the policies. A Flow object is a simple structure where we store the VLAN, MAC source and destination, IP source and destination and TCP port. Next, the parser

queries the processing units manager (described below) to retrieve the switch id associated to the processing units specified in the policy, as well as the port needed to send data to those units. This is added as a collection to the Policy object. Finally, a flag is used to indicate whether the reaction in case of a security event is to alert, quarantine or block. Algorithm 1 shows how policy definitions are parsed.

Note that we intentionally keep the policy parser OpenFlow-independent. The fields used to describe a flow are standard and the Policy object can be used by any other SDN protocol to update the forwarding tables of switches.

```
Data: new file path path
//Parse new policy from file Policy policy = new Policy( )
lines = readFile(path)
for each line f in lines do
    if line starts with 'Flow' then
        //Create a match based on fields match = createMatch(line);
    end
    if line starts with 'Service' then
        //save codes of units (DPI, DoS, ...) servicesCollection = getServices(line);
    end
    if line starts with 'React' then
        //Remember expected reaction
        reaction = getReaction(line);
    end
    policy.setMatch(match);
    policy.setUnits(units);
    policy.setReaction(reaction);
    policy.setVlan(getNextVLAN( ) );
end
```

Algorithm 1: Parsing a file into a policy object.

3.5.2 Policy checking

The policy checker receives a Policy object from the parser and verifies that it does not conflict with existing ones. Checking for conflicting rules is particularly challenging in deployments with multiple controllers sharing network control through a slicing technique such as FlowVisor [51]. In such scenario, the checker

should verify that controllers do not use unauthorized resources and do not override rules pushed by other controllers.

In the context of OpenSec, all rules are pushed by the controller and this simplifies the task. Our policy checker verifies that the Flow attributes of each Policy object do not conflict with each other. For example, two identical Flow objects cannot be part of different policies, since this will become ambiguous when the rules are pushed to the switches.

The policy checker becomes increasingly important as the north API of OpenSec becomes more sophisticated. We plan to allow for multiple security applications to automatically add new policies and the policy checker needs to verify that those applications do not interfere with each other. However, we do not address this issue in the current work.

3.5.3 Policy implementation

Once a policy has been parsed and checked for conflicts, the policy implementer can convert all objects into OpenFlow-compliant rules. The policy implementer is the only component in OpenSec that is currently tied to the OpenFlow protocol. It converts the Flow objects created by the policy parser into the specific instance needed by the OpenFlow controllers (described below) to push new flow rules into the switches. Note that several rules can be created simultaneously if the policy specifies multiple security units. Rules will be pushed into the switches attached to those devices.

When a new rule is pushed to a switch using OpenFlow, the message issued by the controller must include an input port, a match and a set of actions. Note that at least one rule will be inserted for each processing unit listed in the policy. Therefore, for each unit OpenSec first finds the input port where traffic is expected.

This is available in the data repository where all existing forwarding rules are inserted (see Table 3.2). We assume that a rule already existed to carry the traffic either in our out of the local network and, as a consequence, we expect that OpenSec will find a rule that matches the pattern given in the policy. Once the input port has been found, the match is created based on the policy matching information. Finally, the action is computed as follows. First, an output port must be added so that traffic reaches the middlebox. Second, a VLAN tag must be added to uniquely map this traffic to a policy. Indeed, when a processing unit informs the controller that malicious traffic has been detected, the VLAN id and the source IP address are provided in the notification. These two fields uniquely map a source to a policy, allowing OpenSec to react to traffic coming from the identified source as specified by the policy. This is explained in more detail in Section 3.6. All the steps described are shown in Algorithm 2. As a result, FlowMod OpenFlow messages are issued for each middlebox to match on the input port and the policy matching fields and to mirror traffic to the units through the port retrieved from the database.

3.5.4 Policy enforcement

One important step of policy-based management is policy enforcement. This step consists of periodically checking that policies are effectively implemented. To do so, the controller issues *packet_in* messages that match the fields of each implemented policy. Next, the controller verifies that the issued packets were routed appropriately. One possible way to check this is to craft packets that will actually trigger alerts. Another one is to have the units notifying the controller that a test message has been received. We are currently working on this component and therefore do not provide evaluation results yet.

```
Data: Policy policy
//Implement policy in network
servicesCollection = policy.getServicesCollection( );
for each service u in unitsCollection do
    //For each service code, find the unit
    Unit unit = unitManager.getUnit(service);
    //Get DPID, inPort and match to create a flow rule
    dpid = unit.getDPID( );
    inPort = unit.getInPort( );
    match = policy.getMatch( );
    //Get input port from existing rule
    inputPort = database.findInputPort(match);
    //Get next available VLAN tag
    vlanTag = database.getNextTagAvailable();
    //Update existing rule
    writeFlowMod(match on: inputPort and match, actions: add vlan tag, output to
    port inPort );
end
```

Algorithm 2: Implementing a policy.

3.5.5 Step-by-step example

To summarize this section, we provide a step-by-step example of how an operator can implement a policy using OpenSec. Suppose that we implement the following policy using the sample network in Fig. 3.1:

Flow: VLAN=192; **Service:** DPI; **React:** block.

This policy ensures that traffic tagged with VLAN 192 is mirrored to the DPI unit. Also, any source sending malicious traffic should be blocked.

To implement the policy:

1. A network admin should write the policy using the GUI shown in Fig. 3.2.

2. OpenSec parses the policy and locates the switch where the DPI unit is connected, as well as the interface (switch 1, port 25 as shown in Table 3.2).

3. OpenSec assumes that one or more rules already exist so that traffic from VLAN 192 can go through the network.

4. OpenSec finds the rule in switch 1 that matches packets tagged with VLAN 192 to get the appropriate input port.

5. OpenSec also finds the next VLAN tag available to identify traffic from this policy, assume it's VLAN 20.

6. OpenSec modifies the rule so that traffic is forwarded as specified by the original rule, but also forwarded to port 25. Additionally, the VLAN tag 20 is also added.

Once a policy has been implemented, traffic is routed to the processing units and security alerts might be issued by the middleboxes. Next we describe how OpenSec reacts to these alerts.

3.6 Operation of OpenSec: reaction to security events

One key feature of OpenSec is the ability to automatically react to security alerts without involving the network administrator. In this section we describe with more detail how this process is achieved. The overall process is shown in Fig. 3.4.

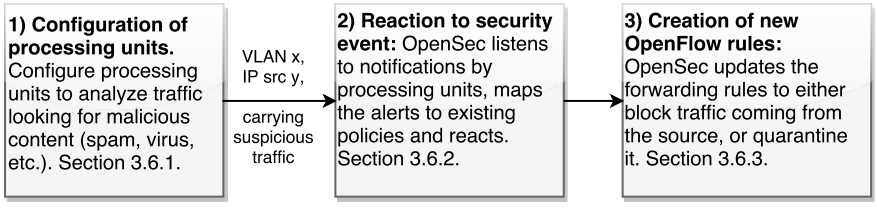

Figure 3.4: Steps needed to react to a security event.

3.6.1 Configuration of processing units

OpenSec relies on the processing units to perform security scans on incoming traffic. In the current implementation, each middlebox must be configured by the network operator to do a specific task. However, enabling automated configuration by OpenSec is a possible future work direction. To be compatible with OpenSec, a processing unit must be capable of sending a message to the controller indicating that a tuplet {VLAN, IP source} is behaving suspiciously. As we described in Section 3.5.3, OpenSec tags traffic with a VLAN to uniquely map it to a given policy. Therefore, by sending the VLAN tag and the IP source, the processing unit identifies the sender but also indicates OpenSec which policy caused this flow to be routed to the unit.

3.6.2 Reaction to security event

OpenSec listens to notifications from processing units using application layer sockets. When a new message arrives, a new threat handles it. First, the process reads the VLAN tag and the IP source of the suspicious node. Next, the process queries the data repository to get the policy mapped to the received VLAN tag. Finally, the process retrieves the type of reaction specified in the policy. As a result, OpenSec now knows if the source must be blocked or sent to quarantine. If the specified reaction is 'alert,' forwarding rules are not modified and the network administrator is notified by e-mail. Otherwise, we describe next how to modify the forwarding rules.

3.6.3 Creation of new OpenFlow rules

The creation of new rules depends on the reaction type. To quarantine traffic, a processing unit logging all traffic is attached to one of the switches on a given port. By default, a rule exists on all switches to forward to the quarantine unit all traffic tagged with a specific VLAN tag. Therefore, to react to such attack, OpenSec must simply insert a rule at the edge switch that will tag all incoming traffic from the suspicious source with the VLAN tag associated to the quarantine unit. Similarly, if the policy indicates that traffic should be dropped, then OpenSec inserts a rule at the edge switch to do so.

3.6.4 Step-by-step example

Consider once again the example started in Section 3.5.5 that forwards traffic to a DPI unit and blocks suspicious sources. Now we describe the steps followed when an alert is sent by the DPI unit to OpenSec.

1. The DPI processing unit is configured by the administrator to perform some security scan.

2. The processing unit detects malicious traffic coming from source 174.145.23.3 and sends a notification to the controller: {20, 174.145.23.3} (20 is the VLAN used in the example started in Section 3.5.5, the IP source is only an example).

3. OpenSec retrieves from the data repository the policy mapped to VLAN tag 20 and the reaction specified by the policy (block).

4. OpenSec issues a flowmod message to the edge switch asking to block all traffic coming from source 174.145.23.3.

Next we describe two use cases that demonstrate the advantages of OpenSec. First, we show how the framework can be used to enforce network access requirements in a campus network in Section 3.7. After that, we show how it can be used to deploy a scientific demilitarized zone (Science DMZ) in Section 3.8.

3.7 Use case 1: traffic analysis for campus networks

The first use case we consider is a residence hall network where both outgoing and incoming traffic must be monitored. We describe through emulation the OpenSec policies implemented and the benefits obtained. First we look at controlling outgoing traffic and later we focus on incoming messages.

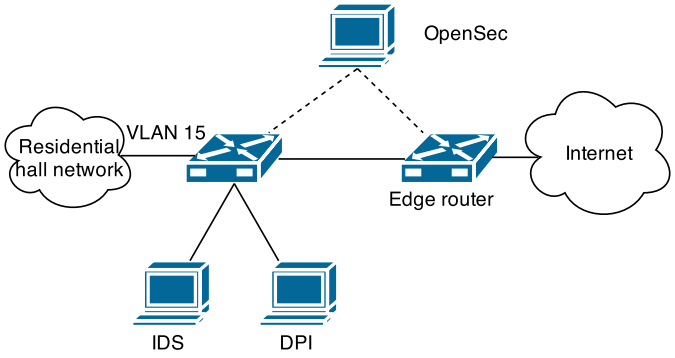

Figure 3.5: Campus topology for housing Internet traffic.

3.7.1 Controlling outgoing traffic using OpenSec

Consider the residence hall network control requirements of the Carnegie Mellon University (CMU) campus [178]. The campus policy establishes acceptable practices for residence hall and dedicated remote access network connections. One of the requirements is that mail bombing, ping flooding, smurf attacks and SYN flooding are forbidden. Mail bombing consists of sending huge volumes of email

to a specific address to overload the mailbox or the mail server. Ping flooding is a denial of service attack achieved by sending a large amount of ICMP echo requests. A smurf attack consists of contacting multiple nodes using a forged IP source address. When the nodes reply, the attacked node is overloaded with messages. We deployed in GENI [122, 179] a topology as shown in Fig. 3.5 to demonstrate how OpenSec can be used to enforce the requirements of CMU. All OpenFlow switches are implemented using virtual machines running Open vSwitch.

The experimental setup of the first scenario is based on traffic collected at the University of Twente in the Netherlands [180]. A 300 Mbit/s (a trunk of 3 x 100 Mbit/s) Ethernet link has been measured, which connects a residential network of a university to the core network of this university. On the residential network, about 2000 students are connected, each having a 100 Mbit/s Ethernet access link. The residential network itself consists of 100 and 300 Mbit/s links to the various switches, depending on the aggregation level. The measured link has an average load of about 60%. Measurements were made in July 2002. This trace ensures that OpenSec is receiving a realistic traffic load corresponding to an actual campus residential network. In total, there are 24 GB of data with 14M flows.

We also developed two security units: one for intrusion detection and one for deep packet inspection. The IDS unit runs Bro [181], an open source network analysis framework capable of intrusion detection. The DPI unit is built on top of nDPI [182], an open source DPI tool. nDPI supports all major networking protocols at any layer, such as IPv4, IPv6, UDP, TCP, HTTP, DNS, SSH, SMTP, Flash and many others. The IDS unit is configured to detect ping flooding and SYN flooding attacks by sending an alert when a given source sends more than 25 ICMP echo requests per second. Similarly, if a node sends multiple TCP handshake requests and then drops the connection, an alert is also raised. Finally, malicious traffic is

generated using nmap [183] and scapy [184].

We use OpenSec to ensure that outgoing traffic tagged with label 15 goes through the implemented security units. To do so, we deploy the following policy:

Flow: VLAN=15, **Service:** DPI, IDS, **React:** block.

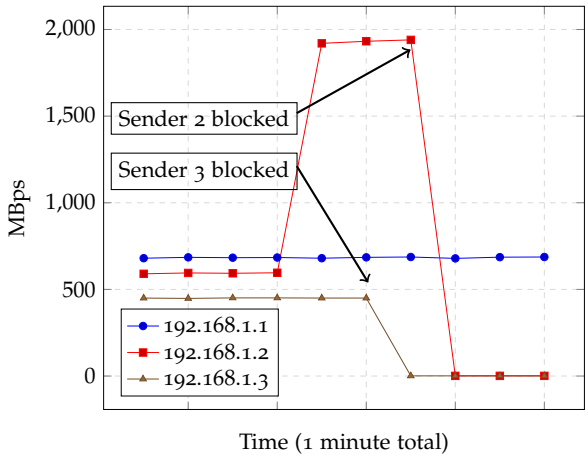

Figure 3.6: Automated blocking of sources 192.168.1.2 and 192.168.1.3 after detecting a SYN flood and a Smurf attack. Source 192.168.1.1 remains unblocked.

The focus of the experiment was to demonstrate how OpenSec can detect traffic that does not adhere to the policy and block a source. We created traffic at three different nodes inside the residential network. The first sender (IP address 192.168.1.1) generates normal traffic by copying a file from a server on the Internet side network. However, the second sender (IP address 192.168.1.2) launches a ping flood attack and a third one (IP address 192.168.1.3) sends a smurf attack by manually crafting packets using the scapy tool.

Figure 3.6 shows the traffic reaching the Internet coming from the three senders. First, The traffic from sender 192.168.1.1 remains constant and never gets blocked.

Second, the traffic from sender 192.168.1.2 starts constant but then increases significantly (this corresponds to the time when the ping flood is started). As soon as this happens, the IDS unit notifies OpenSec that the sender 192.168.1.2 is sending malicious traffic. The unit also sends the VLAN tag to uniquely identify this policy. When this happens, OpenSec retrieves the policy using the VLAN tag, finds the expected automated reaction and blocks the sender. The same happens when sender 192.168.1.3 starts sending spoofed IP source addresses, as the DPI unit informs OpenSec and the sender is automatically blocked. As a result, no more traffic is allowed from senders 2 and 3.

3.7.2 Protecting the residential network from outsider attacks

Next we look at traffic coming from the Internet into the network. To do so, we use another dataset available at the University of Twente that collected traffic directed to a honeypot to evaluate OpenSec at a higher scale [185]. The honeypot was connected to the Internet and ran network services such as SSH, FTP, HTTP and so on. By design, only suspicious traffic was forwarded to that node. The traffic trace was also labeled and organized in a database of flows, alerts and alert types. As a consequence, it is very convenient for experimentation because each flow has been labeled with one attack type, such as SSH scan, SSH connection, FTP scan, FTP connection or HTTP connection. Table 3.3 shows a detailed classification of the dataset.

Table 3.3: Type of traffic in the dataset

Traffic type	Number of IP sources	Number of flows
SSH_conn	103,104	13,939,813
FTP_conn	5	12
Entire dataset	107,988	14,170,132

The experimental setup using the GENI testbed for this scenario is as follows. We replayed the honeypot traffic dataset from a node in the Internet to replicate all the messages. To detect suspicious ssh connections, we configured Bro to alert when a single source attempts more than six connections every five seconds or less. This decision is based on the traffic received by the honeypot where a majority of ssh connections were attempted every five seconds in average. Moreover, we created a policy that mirrors to Bro all incoming traffic destined to port 22.

First we evaluate the detection accuracy of simply alerting when a source attempts connections every five seconds for at least six times. Since OpenSec will only issue an alert for one out of six connections, we expect a number close to the sixth part the total number of flows. However, a close observation of the start and end times of each connection yields that, out of 1000 connections, an average of 20 will go unnoticed by OpenSec because there is a pause of more than five seconds between two or more alerts.

Second, we modify the policy to block traffic instead of only issuing an alert. Table 3.4 shows the number of flows that reach the destination instead of been blocked, as well as the number of sources blocked. Note that 95.5% of flows are stopped before reaching the destination and 99% of attacking nodes are blocked from the network.

Table 3.4: Results of implementing the blocking policy

Total flows	Flows reaching destination	Total IP sources	Sources blocked
13,939,813	628,624	103,104	102,085

In the next Section we describe another campus network scenario and we describe how to use OpenSec for scientific networking purposes.

3.8 Use case 2: Deploying a Science DMZ

In this section, we describe how Science demilitarized zone (DMZ) [186] can be deployed in a campus network using OpenSec.

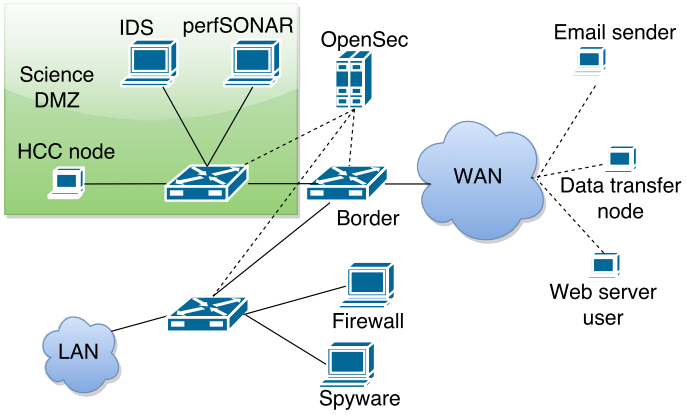

Figure 3.7: Science DMZ in a campus network.

3.8.1 Science DMZ

Science networks carry high-speed data transfer flows that need high bandwidth and are very susceptible to packet loss. Therefore, the goal of a Science DMZ is to rout traffic through a path with customized controls that ensure an acceptable level of security while guaranteeing a high-speed loss-free channel. Unlike the Science DMZ, the traditional DMZ must protect the network against multiple threats.

Figure 3.7 shows how a border router splits traffic into two paths as soon as data reach the campus network. Inside the Science DMZ, there is an end-host customized to receive high-rate data. The Science DMZ also contains a perfSONAR node [187] for performance measurement purposes. Note that packets do not traverse these monitoring devices; instead, traffic is simply mirrored so that there is no performance loss.

Table 3.5: Syntax to create policies using OpenSec and Procera.

OpenSec	Flow: EtherPort = 1 Service: perfSONAR, IDS React: alert */* Security controls for the Science DMZ */*	Flow: EtherPort = 1 Service: Firewall React: alert */* Firewall for all traffic going to the LAN */*	Flow: TCP-dp = 25 Service: Spyware React: block */* Spyware detection of incoming mail */*
Procera	**proc** *world* $\rightarrow$ **do** **returnA:** λ *req* $\rightarrow$ **if** EtherPort=1 **and** security event already exists *alert* **else** *allow* $\odot$ *redirect 10.10.1.1* $\odot$ *redirect 10.10.1.2*	**proc** *world* $\rightarrow$ **do** **returnA:** λ *req* $\rightarrow$ **if** EtherPort=1 **and** security event already exists *alert* **else** *allow* $\odot$ *redirect 10.10.1.3*	**proc** *world* $\rightarrow$ **do** **returnA:** λ *req* $\rightarrow$ **if** TCP-dp=25 **and** security event already exists *deny* **else** *allow* $\odot$ *redirect 10.10.1.4*

3.8.2 Deployment of a Science DMZ using OpenSec

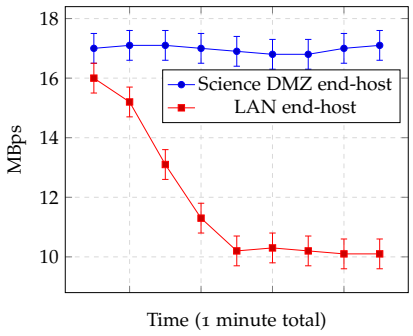

Figure 3.8: Number of bytes received by end-hosts in the Science DMZ and the LAN. The host in the science DMZ receives more traffic because the path between end-points is faster. For the host in the LAN, security devices such as the firewall decrease the performance and the traffic rate is lower.

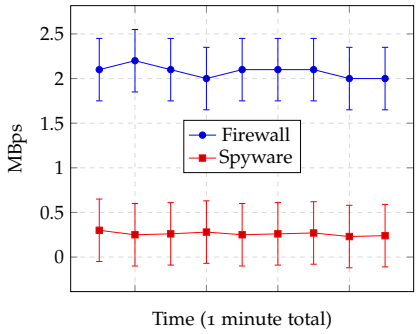

Figure 3.9: Number of bytes received by the firewall and the spyware detection units. The amount of traffic that visits the spyware detection unit is lower because only traffic with destination port TCP 25 is routed through this unit.

We created the network shown in Fig. 3.7 in GENI. The testing devices (email senders, data transfer nodes and webserver users) are deployed in the GENI aggregate located at the University of Illinois. The campus network is deployed in the University of Utah InstaGENI aggregate. In this experiment, the aggregates are "stitched" together through a layer-two tunnel. This connectivity is provided by the GENI testbed.

We also scale up the experiment using a scaling technique provided by GENI so that there are 50 nodes deployed in the Illinois aggregate. Out of 50 nodes, 10 send scientific data at a high rate, 25 send requests to the web server located in the LAN and 15 send email traffic.

The first row of Table 3.5 shows the policies used to realize the required security behavior. On the Science DMZ side, all traffic is mirrored to the IDS and the perfSONAR units. On the LAN side, all traffic is sent to the firewall and then forwarded to the LAN once it has been inspected by the firewall. Also, incoming mail is forwarded to the spyware detection unit.

Note that three simple policies have saved the user from manually adding multiple rules to individual switches. This is a clear advantage of leveraging SDN

to abstract the complexity of the network and show a simple abstraction to the end user.

In terms of network performance, we run two additional experiments. First, we send high-rate flows to a host in the Science DMZ and we do the same with a host in the LAN. Figure 3.8 shows the result of comparing these two transfers. In both cases we use the secure copy (scp) tool to transfer a large file and we measure the number of bytes per second received at each host. We observe that the end-host in the Science DMZ receives a constant number of bytes per second, whereas the rate of traffic sent to the host in the LAN decreases. The reason for this decrease is that the path through the campus network is much slower than the one traversing the Science DMZ and packet loss occurs at the firewall and the spyware detection units. Thus, the TCP implementation of the sender assumes that this is due to congestion and lowers the transmission rate. For this reason, after the initial decrease the throughput becomes stable.

Second, we measure the amount of traffic that traverses the firewall and we compare it with the amount of data routed to the spyware detection unit. Table 3.5 shows the OpenSec policies in this experiment. Note that the third policy ensures that only TCP-25 traffic goes to the spyware unit. As a result, Fig. 3.9 shows the difference between the traffic going to the security units. By filtering mail traffic, we significantly reduce the load of the spyware unit. The second row of Table 3.5 also shows how the equivalent functionality would be achieved in Procera. Notice how the syntax is closer to a programming language rather than a simple list of match fields and security services. While Procera provides other functionalities, OpenSec's syntax is sufficient to provide adequate routing of traffic to the appropriate middleboxes.

The two use cases described show how OpenSec simplifies policy implementa-

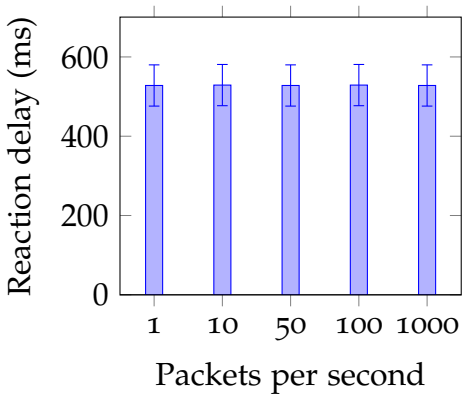

Figure 3.10: Time elapsed between the detection of malicious traffic and the blocking of the source. Independently of the traffic rate, the time needed by OpenSec to detect malicious traffic and block the sender remains constant.

tion in a campus network. Next, we focus on the performance of the framework by studying its scalability and gain in throughput.

3.9 Performance evaluation

In this section we evaluate the scalability of OpenSec and we measure the throughput gain of not having to traverse middleboxes.

3.9.1 Scalability

OpenSec scales well because only the processing units (instead of the controller) deal with the increasing amount of traffic. In the DPI example, the processing unit is capable of inspecting traffic at high bit rates, but the load at the controller remains minimum. As shown in Fig. 3.10, the delay needed to block malicious traffic remains constant independently of the number of packets per second. This delay remains constant because it does not depend on the number of attacks detected or the packet arrival rate. For every alert, the controller simply finds the matching policy and modifies the forwarding rules as required. Also, if the

capabilities of the processing units must be improved, this task is independent of OpenSec and can be performed without modifying the controller. This framework is easier to deploy in comparison to when a middlebox is located in the main data path.

3.9.2 Gain in throughput

Next we show the advantage of removing middleboxes from the main data path in terms of throughput. There are two factors that impact throughput based on our experiments: increased latency and packet loss. To show the impact of latency on the throughput, we first measured how the round-trip delay increases as the traffic traverses more units placed on the datapath (see Fig. 4.7). The impact of this increase on the throughput can be observed in Fig. 3.12. In both figures, we show how the latency and the throughput remain constant when OpenSec is used to mirror traffic to the units instead of traversing all the middleboxes for in-line processing. Although this impact is considerable, the packet loss caused by in-line processing is even more significant. To illustrate this, suppose that a supercomputing facility wants the IDS to analyze all incoming traffic. However, security middleboxes such as firewalls and IDS units are not yet capable of dealing with big data flows without dropping packets. Dart et al. [186] show the TCP throughput can be reduced by a factor of 9 if a router is losing a very small percentage of traffic. To verify this, we simulated a DPI unit that is traversed by all traffic and then sends all data back to the main data path. To evaluate the benefit of using OpenSec, we compared the throughput achieved using an in-line IDS with the throughput achieved using OpenSec. We experimented with different percentages of packet drops and Fig. 3.13 shows how this solution heavily impacts the TCP throughput between the server and the client. By duplicating the traffic,

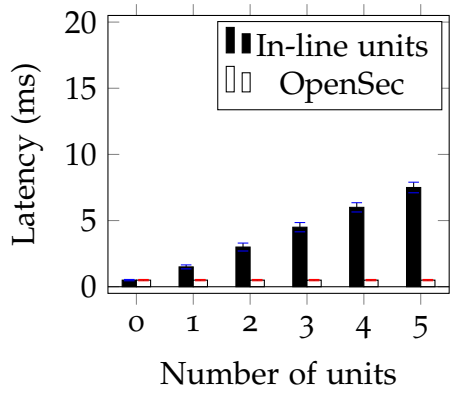

Figure 3.11: Increase in round-trip latency as more middleboxes are traversed by the traffic.

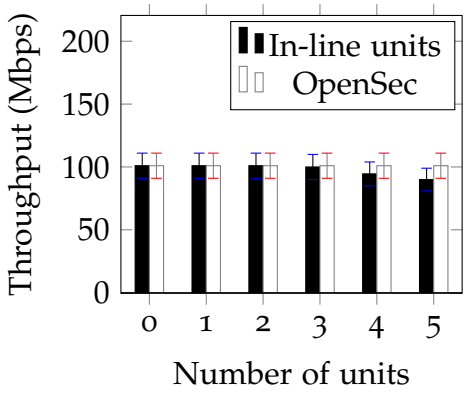

Figure 3.12: Decrease in throughput as more middleboxes are traversed by the traffic. This experiment only considers the decrease due tu an increased latency. The throughput can be reduced further based on packet loss as shown in Fig. 3.13

OpenSec increases the amount of traffic, but also allows the data to traverse the network at a faster rate.

3.10 Comparison with existing solutions

In this section we compare OpenSec against CloudWatcher [157], Fresco [158] and Procera [76], the three similar solutions described in Section 3.2.3.

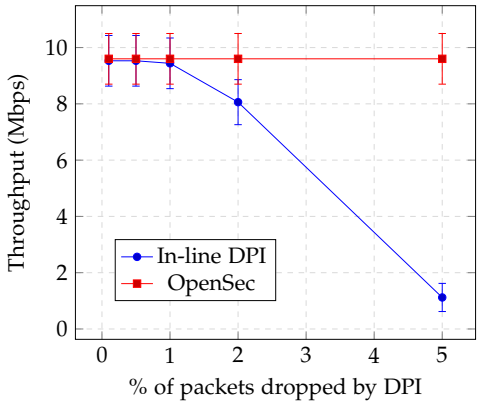

Figure 3.13: TCP throughput achieved using OpenSec and in-line DPI using a 10Mbps link. The packet loss caused by in-line DPI reduces the throughput significantly, whereas it remains constant when using OpenSec because the traffic is only mirrored to the DPI and the packet loss is smaller.

3.10.1 Procera

The main advantage of OpenSec with respect to Procera is simplicity. We showed in Table 3.5 how OpenSec's syntax is simpler than Procera's to deploy a Science DMZ. Also, the fact that OpenSec does not expose switch events to the end-user simplifies the network administration. We do not provide a quantitative comparison because no comparable numerical results are provided in Voelli et al. [76].

3.10.2 CloudWatcher

Next, we compared the time needed by OpenSec to translate policies into OpenFlow messages with the results achieved by CloudWatcher. The results, shown in Table 3.6, do not include the time needed to send the message from the controller to the switch, but only the time needed to translate a policy into a set of OpenFlow messages. Table 3.6 shows the results when there are one, two or three processing units involved in the policy. Because CloudWatcher evaluates multiple algorithms, a range of time is given. In all cases, OpenSec achieves a faster time because we

do not consider routing in our proposed solution. However, we do note that times for both solutions are similar.

Table 3.6: Time needed to create OpenFlow rules in OpenSec and CloudWatcher for a single policy

	One unit	Two units	Three units
OpenSec	0.07 ms	0.07 ms	0.07 ms
CloudWatcher	0.1-1.1 ms	0.1-1.15 ms	0.1-1.2 ms

3.10.3 Fresco

Table 3.7 compares the time needed to implement the network rules using only one controller, Fresco or OpenSec. The results show that OpenSec needs less time to parse the policy and push rules into the switches.

Table 3.7: Time needed to create and push OpenFlow rules in OpenSec and Fresco for a single policy

	NOX/Floodlight	Scan detection
Fresco	0.823 ms (using NOX)	2.461 ms
OpenSec	0.45 ms (using Floodlight)	0.46 ms

One limitation common to OpenSec and Fresco is that a certain delay exists between the detection time and the moment when traffic is actually blocked. Due to the fact that the delay remains constant, the number of packets that bypass the quarantine unit grows linearly. This behavior is shown in Fig. 3.14. The policy deployed for this prototype works well to detect attacks that are carried over multiple packets, such as a denial of service attacks. In such scenarios, reacting to the attack after a small number of packets have reached the server is acceptable because the number of packets that go through is too small to effectively launch an

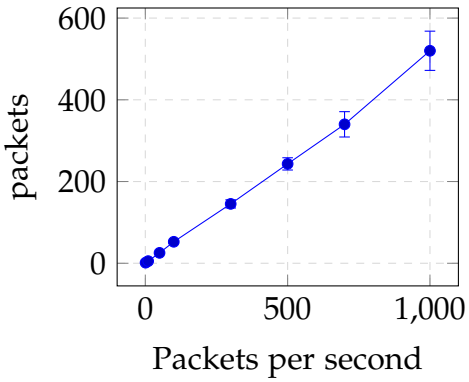

Figure 3.14: Number of packets that go through after detection of malicious traffic. As the traffic rate increases, the number of packets that go through while the blocking is being implemented grows linearly.

attack. However, if the requirement is to detect smaller attacks that are carried over a small number of packets (SQL injection, for example), then in-line processing is needed.

OpenSec can also achieve real-time blocking using a a different policy that sends all traffic to the unit and then receives all traffic back and forwards it. This allows the processing unit to drop packets. However, we showed in this chapter how having an in-line processing unit can significantly reduce the throughput of a flow. Therefore, a trade-off exists between achieving higher performance and having a faster response time to block suspicious packets.

Finally, one advantage of OpenSec in comparison to other solutions is that the policy language is simpler. Solutions such as Procera rely on existing languages that are fairly complex to read and look more like programming languages. OpenSec provides the simplest way to establish OpenFlow matching fields, a list of security units and a reaction, in comparison to other solutions.

3.11 Conclusion

In this chapter we presented OpenSec, an OpenFlow-based framework that allows network operators to describe security policies using human-readable language and to implement them across the network. OpenSec acts as a virtual layer between the user and the complexity of the OpenFlow controller and automatically converts security policies into a set of rules that are pushed into network devices. OpenSec also allows network operators to specify how to automatically react when malicious traffic is detected. OpenSec allows for automated reaction to security alerts based on pre-defined network policies. By doing so, it contributes to hiding the complexity of the network to security operators, who only need to focus on defining the policies.

Our evaluation shows several advantages of OpenSec. First, moving the analysis of traffic away from the controller and into the processing units makes our framework more scalable. Even when the load is high, the controller is not a bottleneck. Second, OpenSec is a first step towards moving the security controls away from the core of the network. This is a key requirement in a network that leverages Cloud security, for example. Instead of controlling every device, the local network just sends data to the cloud and reacts based on the alerts received by the cloud service provider. Third, OpenSec fits well in scenarios that require mirroring of traffic to monitoring devices. This is particularly true for the Science DMZ use case that we demonstrated, because of the significant throughput gain achieved. This use case also showed how multiple policies can be combined together to achieve a campus-wide deployment. Likewise, the use case of residential networks showed how a simple policy can be used to control network access from housing networks. Moreover, our scalability results show that OpenSec achieves a constant

reaction time for different traffic rates.

Future work includes securing the OpenFlow framework and enforcing policy implementation. In order to deploy OpenSec at any production level, several security measures must be taken into consideration, such as switch authentication, physically distributed control plane and also authenticating the users that are allowed to add policies to OpenSec. We are also interested in addressing automatic enforcement of the policies. Likewise, the implementation of some OpenSec's features such as the policy enforcer and a more sophisticated northbound API is left as future work. Finally, we plan to investigate how to speed up in-line processing using programmable boards such as NetFPGA cards.

The usage of SDN and OpenFlow in datacenters and LAN is widely accepted by now. However, for larger scale deployments it is still a challenge to identify the best way of using SDN to provide flexibility to network operators. In the next Chapter, we describe how SDN can also be used at the WAN scale by proposing a framework that virtualizes the network and provides bandwidth provisioning to the WAN tenants.

Chapter 4

WAN scale: Dynamic Network Provisioning for SDN Transport Networks

4.1 Introduction

Current science projects rely on large-scale collaboration and data transfers. The need for high-speed networking has enabled innovation on high-bandwidth, multi-domain and multi-layer networking. Research and education organizations such as ESnet and Internet2 are currently capable of transmitting data at 100 gigabits per second and are aiming at 400 Gbps across their wide area network (WAN) [55, 188]. However, one challenge that continues to exist is building a dynamic end-to-end circuit. By end-to-end, we refer to a circuit that goes all the way from a scientific processing unit to another, instead of a circuit that only traverses the WAN. The difficulty is to go from the WAN to the LAN dynamically and for end-sites to have control without being exposed to the WAN topological complexity introduced by slicing architectures [188].

The first contribution of this chapter is to describe the main challenges in deploying SDN to achieve efficient end-to-end circuit provisioning for science networks. Several problems have been identified in the past for any SDN deployment, such as dependency on the controller or the delay needed to insert forwarding rules in switches. In this chapter we focus on specific challenges of SDN that can significantly impact the ability to deploy SDN-based science networks. For

example, we describe the challenge in efficiently virtualizing a WAN, optimally placing controllers to reduce latency or dealing with the interoperability and security problems raised when hosting multiple tenants. We also describe the challenge in achieving multi-domain provisisioning for scientific flows.

The second contribution of this chapter is to propose the eXtensible Traffic Engineering Framework (XTEF), an SDN-based transport network model capable of WAN virtualization and on-demand network provisioning for scientific flows. The two main components of XTEF are OneSwitch [16] and DTS (dynamic tunnel setup). OneSwitch is a WAN abstraction model that exposes the entire WAN as a single switch so that external application controllers can dynamically create flows across the network. DTS is an algorithm used to provision the WAN that reactively bypasses busy IP routers using WDM tunnels. In this chapter, we provide a detailed explanation of OneSwitch and DTS.

To evaluate the framework, we first evaluate the performance of OneSwitch with multi-tenant usage on the GENI testbed. To do so, we measure the delay introduced by OneSwitch to act as a virtual layer between the physical network and the end-users OpenFlow controller. We also evaluate the scalability of OneSwitch by investigating how many tenants can use instances of OneSwitch at the same time. Next, we emulate a large network using Mininet to evaluate the scalability of the DTS algorithm. As a result, we compare the network performance with and without on-demand WDM tunnels and show how 10% additional flows when using WDM tunnels given that one wavelength per ROADM is available to be used for a tunnel.

The remainder of this chapter is organized as follows. First, we survey the related work in Section 4.2. Next, we describe the challenges in using SDN for scientific networks in Section 4.3. After that, we describe the main components

of XTEF in Section 4.4 and we explain the implementation details in Section 4.5. Next, we describe the experimental setup in Section 4.6 and we evaluate XTEF in Section 4.7. Finally, we conclude and discuss future work in Section 4.8.

4.2 Related work

This section includes related work on network virtualization using SDN, SDN at WAN scale and multi-layer bandwidth provisioning using SDN.

4.2.1 Network virtualization using OpenFlow

SDN and OpenFlow have been commonly applied to network virtualization, in particular at a data-center scale. Casado et al. [189] provide a good explanation of what does it mean to virtualize the network forwarding plane and how it can be done. They clarify the difference between providing a logical view of the network to a software running on top of the virtual layer and just slicing the forwarding plane to multiplex access to it. They describe an OpenFlow-based prototype implementation of such virtualization and they discuss possible scenarios.

Matias et al. [190] propose a network virtualization scheme to simulate a data center's network. They use virtual switches, links and nodes. A node virtualizes a VM host and in the physical network, multiple virtual hosts can reside in that node. Instead of using VLAN tagging to achieve virtualization, the authors suggest using locally managed mac addresses. They use part of the address to identify a node.

A whitepaper by NEC [191] proposes ProgrammableFlow, another end-to-end virtualization scheme that shows the end user virtualized bridges, routers and nodes. This study focuses on allowing users to create virtual networks using

scripts. Internally, ProgrammableFlow uses a topology service, a path mapper and a flow mapper to translate between the physical network and the virtual one. ProgrammableFlow supports L3 virtualization as well.

Drutskov et al. [192] propose an implementation of a Big Virtual Switch. The virtual switch uses OpenFlow to program the physical switches and is also OpenFlow compliant so that an application controller can program it. The virtual switch hides the physical topology and presents the user nodes, interfaces and links. To improve the performance of the mapping, they query a database to do this.

Finally, Skoldstrom et al. [193] discuss different schemes for a virtual switch in a WAN. They also point out important requirements that should be met by such a switch. The authors also mention that the scheme must ensure that tenants use the bandwidth of the control plane fairly. A compromised application controller could overload the network by sending a large number of configuration messages to the virtual switch. The authors mention how virtual separation can be done through slicing (FlowVisor) or through encapsulation (assign a virtual network id). They also describe how to separate the flow tables: flow-based partitioning or table partitioning.

4.2.2 SDN at a WAN scale

There are significantly less SDN-based deployments at a WAN scale. To the best of our knowledge, only Google and Microsoft have proposed using SDN to control their WANs [10, 194]. Google has deployed SDN in one of their backbone inter-datacenter network [10]. The authors describe how SDN allows them to meet requirements that are very specific to Google. Also, the fact that the number of end-points is relatively small makes it possible to use SDN. In this scenario, Google

controls the WAN as well as the applications running on the end hosts. Having an end-to-end control of the network simplifies the administration of the WAN.

Microsoft has experimented deploying SWAN (SDN-driven WAN) on a testing environment that simulates their WAN. They describe the limitations of MPLS TE to maximize bandwidth utilization and they show how using SDN can highly increase the bandwidth usage. Services communicate to the SDN controller how much data they need to send and how sensitive they are to congestion. The controller computes new paths that guarantee that no congestion occurs for those services that are sensitive to it. According to the authors, simulations show that SWAN carries 60% more traffic than the current production network.

4.2.3 Multi-layer bandwidth provisioning using SDN

Bandwidth on-demand is a well studied problem [195, 196]. Doverspike et al. [196] also proposed using SDN to enable cost-effective bandwidth-on-demand for cloud services. The novelty of our work consists of combining WDM tunneling with a WAN abstraction framework and application-driven traffic engineering to ensure bandwidth guarantees, while also supporting other requirements such as maximum latency limits.

4.3 Description of challenges

In this section we describe some of the challenges that must be addressed when designing SDN-based science networks. Although a majority of these challenges are inherent to SDN, we highlight aspects that are specific to science networks.

4.3.1 WAN virtualization

WAN abstraction is a challenge that has major relevance in the context of science networks. Indeed, the need for dynamic, on-demand circuits motivates network providers to allow for network tenants at scientific facilities to easily setup circuits across the WAN. Naturally, this goes hand-in-hand with network virtualization as the carrier needs the end-user application controller to program the network to achieve end-to-end connectivity. However, the provider must also ensure that the end-user does not have direct access to the network devices. There are different virtualization models based on the level of virtualization. For example, the entire WAN can be abstracted as a single switch and the entire path across the WAN is hidden from the end-user application controller. In contrast, a framework could also allow the tenant to control a customizable amount of network through a virtual set of routers and links.

OneSwitch is an example of WAN abstraction framework. However, we do not describe it in detail here since we do so in Section **??**. Other virtualization models have also been proposed. For instance, Drutskoy et al. [197] propose a similar virtualization technique where OpenFlow is used to control both the network devices as well as the virtual routers. This virtualization model allows for end-users to decide if they want a fully virtualized network or if a finer-grained control is required.

The choice of the appropriate WAN virtualization model is an important challenge since it affects the amount of information revealed to the tenants, as well as the level of control application controllers have on the routing across the WAN. For instance, a tenant could be interested in having two link-disjoint paths between two nodes. Such a request cannot be granted in a model such as OneSwitch. However,

Table 4.1: Existing network virtualization models

Model type	North API	South API	Abstraction level
OneSwitch [16]	OpenFlow	OpenFlow	Entire WAN as a single switch
Big Switch [197]	OpenFlow	OpenFlow	Virtual links and switches
RouteFlow [198]	BGP, OSPF and other standard protocols	OpenFlow	Virtual links and routers
Programable Flow [199]	Proprietary API	OpenFlow	Customized network with routers and bridges

supporting this feature requires revealing a significant amount of topology information. There are other challenges, such as chosing a communication protocol between the provider and the tenant, ensuring interoperability and efficiently virtualizing messages. These challenges are described in the following sections.

4.3.2 Scalability

Two requirements are critical in making SDN scientific networks scalable: efficient WAN virtualization for a large number of tenants and appropriate controller placement to reduce delays.

First we address the problem of efficient virtualization. Any WAN virtualization model needs to consider the overhead needed to map physical to virtual resources every time there is interaction with network tenants. This virtualization includes finding the right tenant based on the VLAN, switch id and ingress port for incoming messages. Similarly, messages coming from the tenants must be translated to appropriately insert forwarding rules in the physical network devices. This overhead becomes significant when the number of tenants increases. Drutskoy et al. [197] propose a container-based virtualization technique that consists

Table 4.2: Order of priority for short-term deployments

Challenge	Rationale
WAN virtualization	Virtualization is necessary to allow tenants to program the WAN. Without virtualization, manual operation is required and dynamic, on-demand circuit reservations are unlikely to be deployed successfuly.
Multi-layer provisioning	Virtualization alone does not provide end-to-end connectivity and the WAN must be provisioned efficiently. Integrated management of circuit and packet networks is necessary to achieve flexible transport networks.
Scalability	Controller placement and efficient virtualization must accompany WAN virtualization for the solution to be feasible for science networks. However, designing the WAN virtualization model and provisioning comes before ensuring scalability.
Multi-domain circuits	Inter-domain communication between SDN peers is important but current deployments can rely on systems such as OSCARS to be deployed. It is more important to successfuly deploy SDN at the edge of the network for virtualization, rather than requiring SDN to control all the path across domains.
Security	An acceptable level of trust is assumed in science networks scenarios. Although security should also be a design priority, the items above are more relevant in order to achieve successful deployments soon.
Interoperability	Interoperability will become an important issue when the number of tenants increases. However, it is hard to address the topic in the short term because there is too much variation in the selection of north and south APIs, and current deployments are likely to be closely tied with a specific technology.

of mapping between controller API calls rather than OpenFlow messages. They combine this with database-driven mapping and show how their virtualization

model can scale up to multiple tenants.

Second, we address the problem of efficient controller placement. This problem needs little motivation, as it has been adequately addressed lately [200, 201, 202, 203]. However, this problem has been less well studied in the specific context of extreme-scale science infrastructures. Given the requirements for bandwidth, latency, and rapid provisioning and re-provisioning of resources, the usual two questions remain: how many controllers are needed and where should they be in the network? Previous studies have proposed latency metrics such as average latency or worst-case latency as optimization problems to decide where to place network controllers. Alternatively, another solution is to place controllers to maximize the number of nodes within a latency bound. Another decision related to the controller placement is the number of controllers that should be deployed in the domain. Heller et al. [200] argue that for most topologies, adding controllers yields slightly less than proportional reduction; that is, k controllers reduce latency nearly to $\frac{1}{k}$ of the baseline latency with one controller.

The problem of dynamically loading controllers in the context of science networks is still open. The bursty nature of scientific traffic makes it hard to predict what is the best location for one or more controllers. Moreover, although studies have proposed deploying a distributed control plane for SDN, controllers are placed and loaded manually. A more dynamic deployment is needed to efficiently place controllers appropriately.

4.3.3 Interoperability

Dealing with multiple tenants results in interoperability challenges for SDN deployments. In particular, the decision on the appropriate northbound and southbound APIs is still open. WAN abstraction models face the challenge of choosing an

SDN protocol twice. On the one hand, the controller needs to communicate with network devices. On the other hand, network tenants must use some protocol to control the WAN.

So far, OpenFlow has been the de-facto standard to deploy SDN networks. This protocol has been supported by plenty of hardware switches and it is also available using Open vSwitch, which greatly simplifies low-cost, research deployments. However, there are reasons to believe that, while SDN is here to stay, OpenFlow is not. For instance, this protocol faces the problem of either providing a standardized common denominator only (OpenFlow 1.0), or providing a very complete set of features that make it hard for vendors to fully comply with the standard (OpenFlow 1.1 and above). The functionality provided by OpenFlow 1.0 is too basic and constantly necessitates extensions to the protocol, which yields interoperability problems. The dificulty of fully complying with OpenFlow 1.1 and above makes it hard to guarantee that an application functioning on a given hardware will also work on another vendor's hardware.

As a consequence, network vendors have looked at other options such as XMPP (Extensible Messaging and Presence Protocol), BGP-LS (Border Gateway Protocol - Link State) or PCEP (Path Computation Element Protocol). These protocols bring advantages such as providing a management plane (which is not provided by OpenFlow) and the ability to support legacy hardware during the transition to SDN. Network providers with WAN virtualization models need to decide which protocols are exposed to the tenants, or if a single protocol should be enforced.

4.3.4 Security

When securing a science network, a fine balance between security and efficiency must be considered. As opposed to a traditional enterprise or campus network, a

deployment for transfer of scientific data cannot place a battery of middleboxes performing functions such as firewall, intrusion detection and spam detection between the two end-points. Indeed, these security mechanisms create packet loss and it has been shown that TCP throughput can be reduced by a factor of 9 if a path loses a very small percentage of traffic [186, 204]. This has motivated the design of the so-called Science DMZ, a section of the deployment between the WAN and the organization's network that is optimized for high speed data transfers [204]. The design of Science DMZs has been extensively discussed by researchers at ESnet. Therefore, we focus our attention on a less studied problem: securing a multi-tenant network in the context of science networks.

The challenge when designing a science network with multi-tenancy is achieving sufficient security while maintaining a network desing that is capable of high speed data transfers. For example, it is important to decide whether to follow a model of control plane "partitioning" or a model of full virtualization. By "partitioning" we mean tools such as FlowVisor [51] that allow for tenants to program some but not all forwarding rules. In full virtualization, instead, a complete translation occurs between the messages received from the tenants and those sent to the network devices. Similarly, attention must be given to prevention and detection of malicious or compromised application controllers. Once again, the delays incurred by the security mechanisms implemented must still enable efficient data transfers for science networks.

4.3.5 Multi-domain circuits

Before addressing multi-domain circuit reservation using SDN, we point out that inter-domain operation is still an open problem for SDN in general. The limited amount of existing work has focused on how an SDN-based Autonomous

System (AS) can co-exist with IP-based ASes ([205]) and on information exchange mechanism between SDN controllers in different domains ([206]). One work goes a step further and suggests outsourcing routing decisions to an external party that will then be responsible for the routing decisions of several domains [207].

In the context of science networks, current solutions such as OSCARS rely on trust between domains and a path computation engine is allowed to directly modify the forwarding rules of a device in another domain. For inter-domain circuit reservation to scale appropriately, a significant amount of work is needed in the east-west interface between controllers.

4.3.6 Multi-layer provisioning

Finally, SDN-based networks must consider the challenge of multi-layer provisioning. Indeed, managing each layer (IP, WDM, OTN) separately greatly reduces the flexibility of the network and the ability to adjust based on traffic demands. In traditional networks, when the IP control plane requires additional bandwidth, then operators of the optical layer must configure new circuits that can later be used by the IP layer. Because this operation requires human intervention, it is not feasible to achieve on-demand network provision automatically. In contrast, an integrated control plane allows for the network controller to optimize how flows are routed across the network. If the same network controller can orchestrate how the IP layer and the optical layer interact with each other, then it becomes feasible to adjust the optical layer capacity in real time based on the needs of the IP layer.

In the next section, we describe how we propose to address the WAN virtualization and multi-layer provisioning challenges using OneSwitch and XTEF.

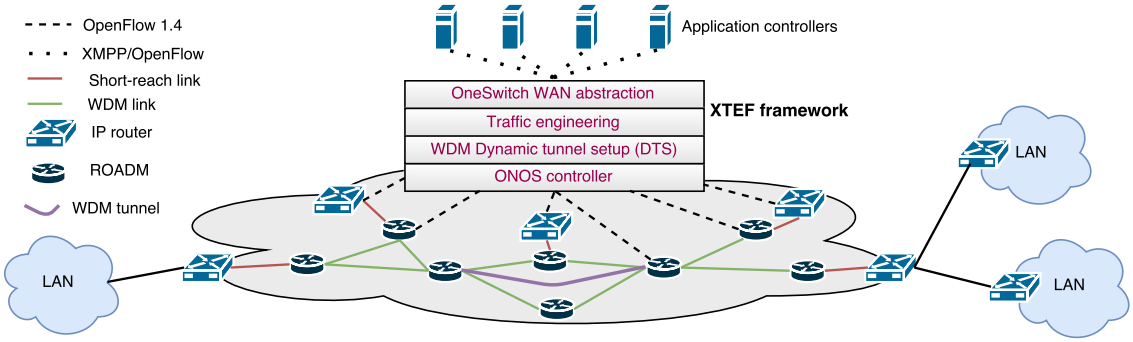

Figure 4.1: The XTEF framework uses the OneSwitch WAN abstraction model, the DTS provisioning algorithm and the ONOS controller.

4.4 Components of XTEF

The goal of XTEF is to provide application-driven network provisioning to external applications. To do so, it first relies on OneSwitch to abstract the WAN and collect traffic engineering requirements from the application controllers. Second, XTEF uses MPLS the DTS algorithm to provision the network. Third, XTEF uses wavelength-based routing to route traffic between IP routers. In this section we describe each of these components.

4.4.1 OneSwitch

OneSwitch [208] abstracts the entire topology of the WAN as a single switch, where each virtual port is mapped to a physical port of a physical switch at the edge of the network. OneSwitch provides a virtual switch to each tenant as shown in Fig. 4.1. Suppose an organization A has access to the WAN through router 1 and port 10. Also, suppose a scientist in organization A wants to retrieve data from a scientific laboratory B connected to the WAN through switch 2, port 11. OneSwitch provides the application controller with a virtual switch that has two virtual ports 1 and 2. By forwarding data from port 1 to port 2, the application controller is actually

sending data from organization A to the laboratory B. Internally, OneSwitch is responsible for translating all messages.

There are several advantages in this model. First, it is simple. An application controller can forward data across the WAN by simply forwarding packets from one virtual port to another. Second, it only involves the routers at the edge of the network and internal topology does not need to be shown to the tenants. Third, the virtualization layer adds security to the deployment, because all rules issued by application controllers must first be handled by OneSwitch.

4.4.2 Traffic engineering

OneSwitch is not responsible for provisioning the WAN. Instead, the framework assumes that there is a path for packets to traverse the network and some way to tag packets (VLAN tag, WDM wavelength, etc.). XTEF uses MPLS label forwarding to provision the WAN resources at the IP layer. For each pair of edge routers, at least two link-disjoint paths are configured. To add variety in our experiments, we configured the network so that the bandwidth available on both link-disjoint paths is the same but one path provides a smaller latency (5 ms instead of 25 ms). The establishment of the paths is not within the scope of this work, so we simply assign MPLS labels across the network to achieve the connectivity between all edge routers. At the optical layer, we use up to 80 wavelengths per link between two ROADMs. Short-range links between IP routers and ROADMs have a capacity of 100 Gbps. Both networks (IP and optical) are managed by the same controller, so we combine MPLS labels and wavelength assignment to create two end-to-end paths between each pair of edge routers. Once the network has been provisioned, the traffic engineering component informs OneSwitch of the label that must be used between all pairs of edge routers.

4.4.3 Dynamic Tunnel Setup algorithm

As the amount of traffic increases, more flows get aggregated in the network links. This can lead to congestion on some short-reach links. Next, we propose DTS, an algorithm that exploits unused wavelengths to temporarily increase network capacity to deal with traffic increase. DTS creates a bypass at the WDM layer between the two closest ROADMs before and after busy short-reach links.

DTS works as follows. First, it monitors the load on all short-reach links. Second, when the load on one of the links is above a threshold (70% in our implementation described below), the algorithm finds which flow is contributing the most to the increased load. Third, given the chosen flow and its path, the two closest ROADM neighbors r_{n-1} and r_{n+1} are selected and a WDM two-hop tunnel is created between them. As a consequence, the flow will not be dropped at ROADM r_n and the load at the busy short-reach link is reduced. Fourth, the matching rules at the entry point of the WDM tunnel are modified so that, instead of matching on the MPLS label only, the IP router also matches on the IP source address of the flow and adds a different lambda for the flows that will be routed through the bypass. Finally, when the load on the bypass is below a threshold (10% in our experiments), the tunnel is removed.

4.4.4 ONOS controller

XTEF uses the ONOS controller [209] to control the routers using OpenFlow. ONOS is an open source SDN controller designed for service providers created in conjunction by ON.Lab and the Linux Foundation. ONOS provides sample code to create applications that provision point-to-point circuits at the WDM layer. This is the main reason why we chose it instead Open Daylight or other similar

controllers.

In the following section we describe in detail the implementation of OneSwitch, DTS and the ONOS application.

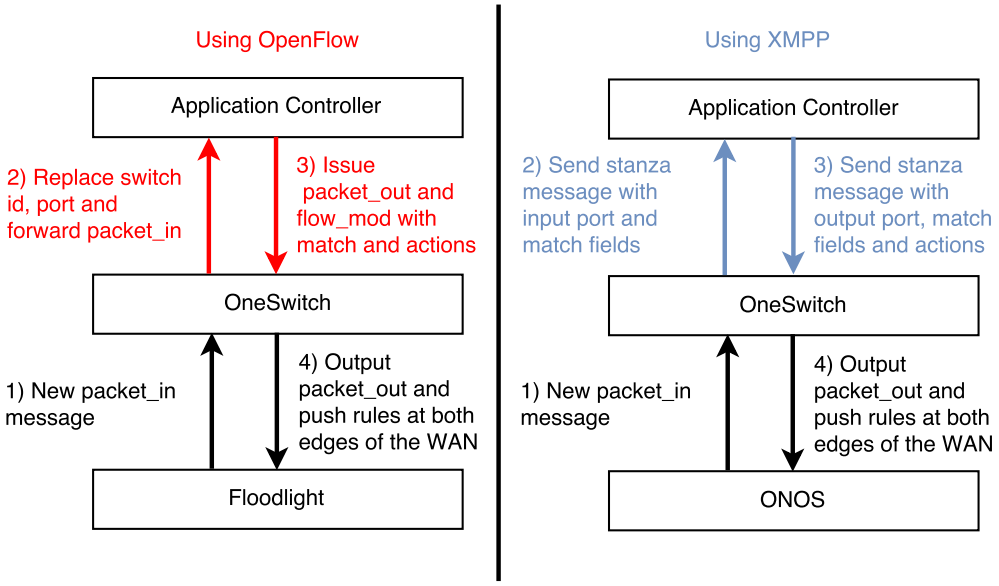

Figure 4.2: Message exchanges using OpenFlow or XMPP as northbound API

4.5 Implementation of XTEF

In this section we describe the implementation of OneSwitch, which varies depending on the chosen northbound API. We also explain how the DTS algorithm monitors link load and how it uses the ONOS app to request for WDM tunnels. Lastly, we describe how such app is implemented in ONOS.

4.5.1 OneSwitch implementation

OneSwitch is the component responsible for virtualizing the WAN. Therefore, its main task is to collect network events from the underlying OpenFlow controller, translate them and forward them to the application controllers. Likewise, when an

application controller sends a message to OneSwitch, this component must convert the message and issue it back to the OpenFlow controller.

As we described earlier, we experimented using two different northbound API's: OpenFlow and XMPP. The implementation using OpenFlow was done on top of the Floodlight controller. In Floodlight (and in other OpenFlow controllers in general), new applications can be added and registered with the controller. Next, when the controller loads, the application is loaded automatically. Therefore, we created a new application that listens to incoming packets (*packet_in* events). When a new packet is received, OneSwitch inspects the input switch id, port and VLAN tag. By doing so, the packet can be uniquely mapped to an application controller. To communicate with application controllers, OneSwitch must act as an OpenFlow-compliant switch. To do so, OneSwitch uses the Netty java library to establish an asynchronous channel with every application controller and it also implements all the OpenFlow messages needed to establish a handshake: Hello, EchoRequest, EchoReply, FeaturesRequest and FeaturesReply. While this increases the complexity of the implementation, it does simplify the operation of application controllers, as they simply need to run an OpenFlow controller to talk to OneSwitch.

The implementation using XMPP was done on top of the ONOS controller to leverage existing sample code to create optical circuits between two IP routers. XMPP was first proposed as a messaging protocol between a provider and a subscriber. To implement a chat, for example, two or more client applications would subscribe to the XMPP server to exchange messages. In our case, we use XMPP for two reasons. First, it provides a standardized way to communicate between two parties. Second, Juniper is already using XMPP as a northbound API, which indicates that the protocol is worth being considered. In our implementation,

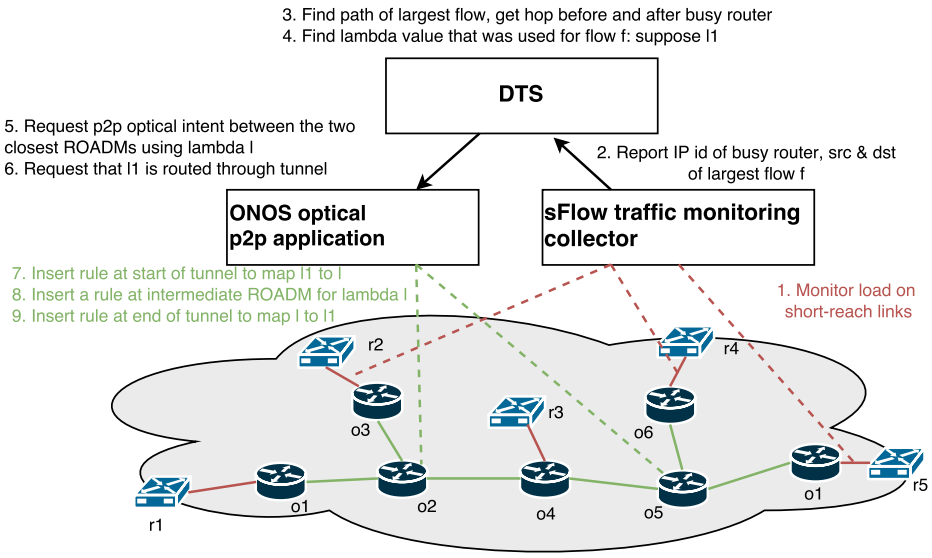

Figure 4.3: Operation of DTS using sFlow and a point-to-point optical intent application on ONOS

we developed the XMPP server on OneSwitch using Openfire [210], an open source Java library for XMPP. In this implementation, OneSwitch is also an application of the ONOS controller that listens to *packet_in* messages and finds the application controller based on the switch ID, VLAN and input port. However, instead of forwarding OpenFlow messages, we created a simple syntax to exchange information between OneSwitch and the application controller. On the one hand, OneSwitch concatenates the switch ID, the input port and the IP source and destination addresses. On the other hand, an application controller concatenates the match fields and the output port.

Figure 4.2 compares how OneSwitch communicates with application controllers using OpenFlow or XMPP. When using OpenFlow, all messages are standard to this protocol. Therefore, it is slightly more efficient because OneSwitch only needs to modify properties of each packet, such as replacing the input port or the switch id. It is also easier for application controllers to simply use an OpenFlow

controller. In contrast, when XMPP is used, all the communication between OneSwitch and the application controllers is based on XMPP *stanzas*, which is the XMPP terminology for messages and this must also be implemented by the application controllers. However, the XMPP implementation is easier to deploy because an open source XMPP server is used and there is no need to implement the communication details, as opposed to when using OpenFlow. Furthermore, bandwidth and latency requests are only possible using XMPP. To do this with OpenFlow, protocol extensions are needed.

Using any of the two implementations, OneSwitch is responsible for tagging incoming packets with the appropriate label to forward flows across the WAN. To do so, it queries the traffic engineering component for an available route given the bandwidth and latency requirements. If the request is not feasible, OneSwitch rejects the flow. Otherwise, it tags packets with an MPLS label and an output port based on the information provided by the traffic engineering component.

4.5.2 DTS implementation

Figure 4.3 shows the implementation steps of DTS. In **step 1**, DTS uses sFlow [211] to monitor the load on each of the short-reach links between IP routers and ROADMs. The DTS algorithm is triggered when the capacity of one of these links exceeds 75%. sFlow is installed on all IP routers following these instructions [212]. A central collector is also installed in the same node as the OpenFlow controller. When a load is higher than 70%, the sFlow component notifies DTS and informs of the router ID, as well as the source and destination of the flow (**step 2** in Fig. 4.3). DTS is network-aware and knows the path of the identified flow, as well as the wavelength that is being used for that flow. Therefore, it locates the two closest ROADMs before and after the busy short-reach link (**steps 3 and 4**). For example,

if r3 is the IP router to be bypassed, then the tunnel is created between o2 and o5 using wavelength l. If the identified flow was using wavelength $l1$, then o2 must replace the labels to send the flow through the tunnel and reduce the load at the short-reach link. In **steps 5 and 6**, DTS requests the ONOS application to create such tunnel using wavelength l.

Next we describe how to expose an application through the ONOS web interface to satisfy tunnel requests issued by DTS.

4.5.3 ONOS application implementation

The ONOS application is responsible for inserting flow rules in the switches to create the tunnel requested by DTS. For a tunnel between ROADMs r_1 and r_3 to bypass r_2 using wavelength l, three rules are required (**steps 7, 8 and 9**). At r_1, the application needs to change the output wavelength. At r_2, a new rule must be issued so that flows with wavelength l are forwarded to the following ROADM to bypass the IP router connected to r_2. To achieve this, the ONOS application is aware of the topology and ports needed to connect r_1 with r_3. Finally, at r_3, wavelength l must be replaced with the original wavelength assigned to this flow.

To make this application accessible to external users, we create a user interface overlay as shown in this tutorial [213]. By doing this, the application can be used from the ONOS web interface. Therefore, we use a *curl* command from the DTS script to request tunnels.

In the next section, we describe the experimental setup used to evaluate the proposed framework.

4.6 Experimental setup

4.6.1 GENI testbed topology

We use two topologies for our experiments. Topology 1 (see Fig. 4.4) is a physically distributed network on the GENI testbed to experiment with large scale and realistic delays. This topology is based on Internet2's network and consists of nine IP routers distributed across the country. We consider Los Angeles, Seattle, Washington D. C. and New York as the edge routers that serve as entry points to LANs. In total, we simulate 50 scientific facilities by deploying 50 nodes connected to the WAN through one of those edge routers. Furthermore, we deploy 50 application controllers, one for each of those 50 facilities. The application controllers are deployed on different aggregates on the GENI slice, but they all have four virtual ports, one for each edge router. Finally, the controller that runs OneSwitch is hosted on the Houston node, which is the most centralized.

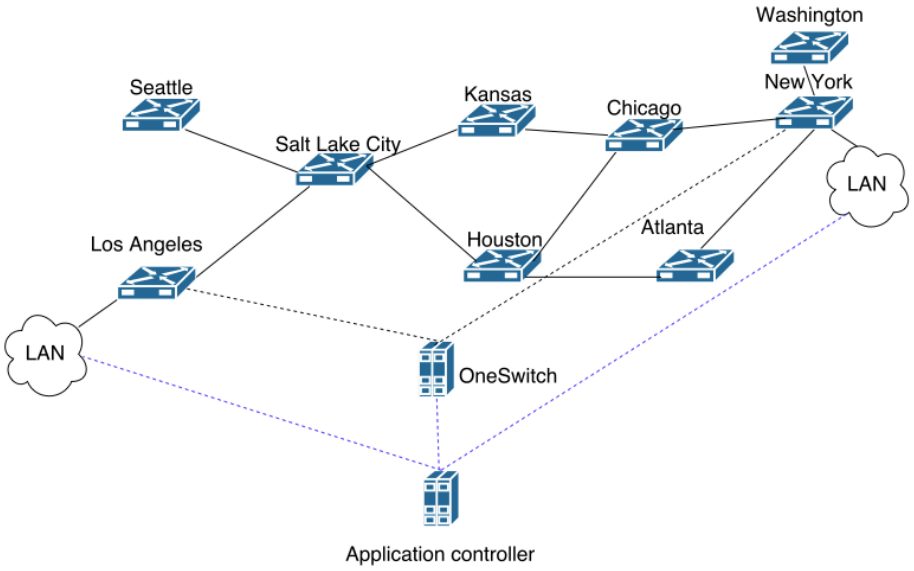

Figure 4.4: Experimental topology emulated in GENI

For topology 1, we are only interested in evaluating the performance of OneSwitch and we do not consider network provisioning. Therefore, we provision ahead of time end-to-end Ethernet-layer circuits between all pairs of edge routers. For example, all traffic between Los Angeles and New York is forwarded through Salt Lake City, Houston and Atlanta. Moreover, each of those circuits is identified with a VLAN tag. Therefore, when an application controller requests traffic between Los Angeles and New York, OneSwitch simply adds the appropriate VLAN tag and the data is delivered to the exit edge router.

4.6.1.1 Scenario 1

Using this topology, we create the first experimental scenario (referred in the evaluation as Scenario 1). The goal is to gradually increase the number of flows from one to 50 and measure the time delay added by OneSwitch during the virtualization. To do so, we measure the average delay needed by OneSwitch to setup a new flow rule at the edge of the network when receiving 1 Gbps flows from each sender node. To have a point of comparison, we use a default SDN learning switch that reacts to incoming *packet_in* messages by pushing new rules to forward traffic. Therefore, we expect to see a longer delay when OneSwitch is used, because more processing is done at the controller.

4.6.2 Mininet emulation topology

The motivation to use Mininet is that large-scale, multi-layer networks can easily be emulated. Indeed, Mininet uses LINC-Switch to emulate reconfigurable optical add-drop multiplexers (ROADMs). These devices are OpenFlow-compliant and support three actions which can be added to forwarding rules: add, forward and drop. The *add* action adds an output port and a *lambda* to specify which wavelength

must be used. The *forward* action matches on an input port and a *lambda* value and outputs traffic to another optical port. Finally, the *drop* action forwards the data through the short-reach link to the IP router. Therefore, ROADMs and IP routers can both be managed by an OpenFlow controller.

The experimental topology is too large to be drawn, but it consists of 15 IP routers, 75 ROADMs, 30 hosts, 98 WDM links with 80 wavelengths each, 54 short-reach links and 100 flow requests. Each wavelength provides a bandwidth of 50 Mbps We also provisioned the network resources statically using MPLS labels to ensure that two link-disjoint equal cost paths exist between each pair of edge routers. We consider 5 IP routers as edge nodes. Each IP router is connected to a ROADM device, but not all ROADMs are connected to an IP router. Finally, each end-host is assigned a wavelength per end-to-end path. To illustrate this, suppose that two senders are connected to router 1 and a total of ten paths are available to another edge router, then router 1 handles twenty different wavelengths. As a side node, the LINC-Switch emulator does not restrict the maximum amount of wavelengths.

4.6.2.1 Scenarios 2 and 3

Using topology 2, we create two scenarios. First, we create 10 traffic flows with latency requests between 5 and 25 milliseconds between the same pair of source and destination nodes. The network is provisioned to provide two paths: one with 5ms latency and another one with 25ms latency. We compare using Equal Cost Multiple Path (ECMP) versus using the proposed XTEF. The scale of this experiment (scenario 2) is small, but our goal is simply to evaluate if XTEF is capable of satisfying the latency requirements issued by the application controllers of all the flows.

Second, we created 60 flow requests using iperf to send data at 5 Gbps on average. The goal of this experiment is to evaluate how much can the DTS algorithm increase the capacity of the network using WDM tunnels. In this experiments, flows are added sequentially to the network load.

4.7 Evaluation

In this section, we first evaluate the virtualization delay added by OneSwitch when new flows enter the WAN. Second, we demonstrate the benefits of having application-driven traffic engineering as opposed of sending all flows through equal cost paths. Third, we evaluate how DTS can increase the network capacity using WDM tunnels. Finally, we evaluate the scalability of the framework.

4.7.1 Virtualization delay and scalability of OneSwitch

Each instance of OneSwitch acts as a virtual layer between the physical devices of the network provider and the virtual Ethernet switch programmed by the application controller. Naturally, this virtual layer adds a processing delay when translating OpenFlow messages coming from the physical switches to similar messages directed to the application controller. Next, we quantify these delays to determine how scalable OneSwitch is and what is the impact on the transmission times of having an intermediate virtual layer. To do so, we use the GENI topology shown in Fig. 4.4 and experimental scenario 1.

Figure 4.5 shows the round-trip time needed to send packets with and without using OneSwitch. When OneSwitch is not used, we hardcode the flow rule at the edge of the WAN so that traffic tagged with a given VLAN is forwarded to the destination. This is the traditional approach in which a network operator

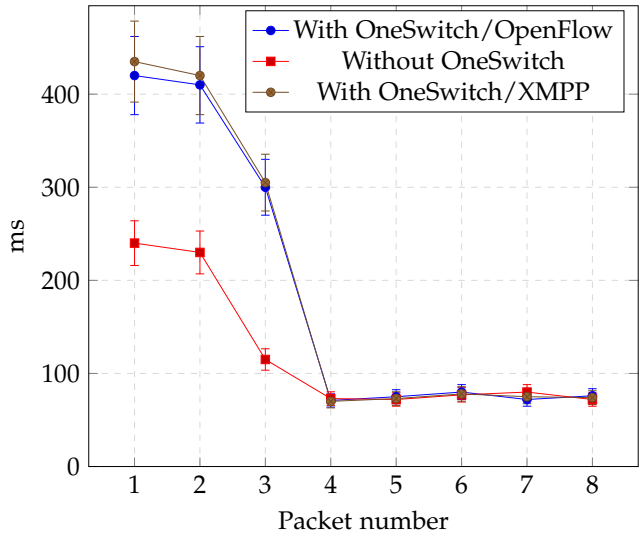

Figure 4.5: Round-trip ping time between Los Angeles and New York for a single flow.

manually configures the edge switch of the LAN to forward packets to the WAN with a given VLAN. By using OneSwitch, the application controller can now automatically control both the LAN edge switch and the WAN edge switch. This adds a delay which is needed to push the flow rules during the first packets. However, notice that once the flows have been pushed, the performance is identical in both cases. Therefore, OneSwitch only adds a delay when the controller needs to handle packets often. In practice, few packets end up going to the controller since rules are pushed proactively. We argue that this delay is acceptable given the great benefit in terms of flexibility offered to the end-user.

Figure 4.5 also shows that the difference in the delay between using OpenFlow or XMPP as the northbound API is very small. We expected the OpenFlow delay to be slightly lower because the processing tasks at the controller are slightly faster. Indeed, when using OpenFlow, OneSwitch simply modifies fields of the already existing *packet_in* and *flow_mod* objects. In contrast, when XMPP is used, new

message objects must be created from scratch. However, this result shows that both protocols can be used without one of them being significantly better than the other in terms of processing delay.

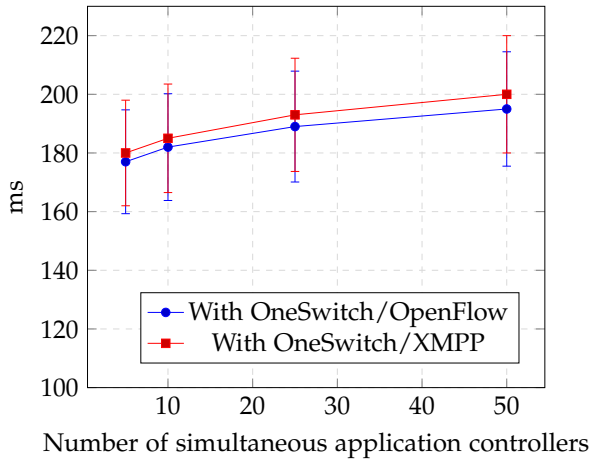

Figure 4.6: Delay introduced by OneSwitch for varying number of tenants.

The results in Fig. 4.5 only consider one single packet arriving at the edge of the WAN. Next we focus on evaluating how OneSwitch handles multiple simultaneous flows. To this end, Fig. 4.6 shows that the delay added by OneSwitch is almost constant when the number of application controllers using instances of OneSwitch increases. OneSwitch handles each end-user application controller using different threads and is thus capable of handling multiple tenants at the same time. In this experiment, we increase the number of end-user applications sending data from 5 to 50. The results show that both implementations of OneSwitch handle the load with only a small increase in the total delay.

4.7.2 Application-driven TE

To demonstrate how XTEF is capable of application-driven TE, we used scenario 2. The network was provisioned to provide two paths: one with 5ms latency

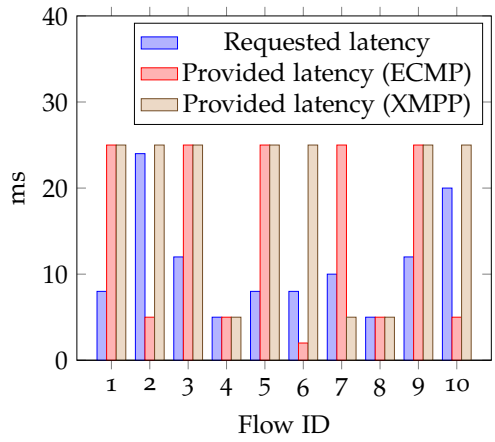

Figure 4.7: Latency limits guaranteed for each flow with and without traffic engineering.

and another one with 25ms latency. We compared using ECMP versus using the proposed XTEF. Figure 4.7 shows how, using ECMP, flows 1, 3, 7 and 9 do not receive the required latency using ECMP. In contrast, XTEF is able to satisfy the requested latency for all flows. This simple proof-of-concept experiment shows how XTEF can respond to the application controller's request and forward flows through the appropriate path.

4.7.3 Network provisioning

Figure 4.8 shows the percentage of flows that received the requested bandwidth with and without WDM tunnels. Using ECMP, the framework was capable of satisfying the demands up to 37 flows but started rejecting new flows afterwards. To compare, we first create a network with sufficient free wavelengths at each ROADM and show how the demand could be completely satisfied. However, this is not realistic and we also experimented on how to use a single extra wavelength per ROADM (i.e. only one tunnel can start, traverse or end at each ROADM). The results show that the demand can be increased by more than 10% using a single

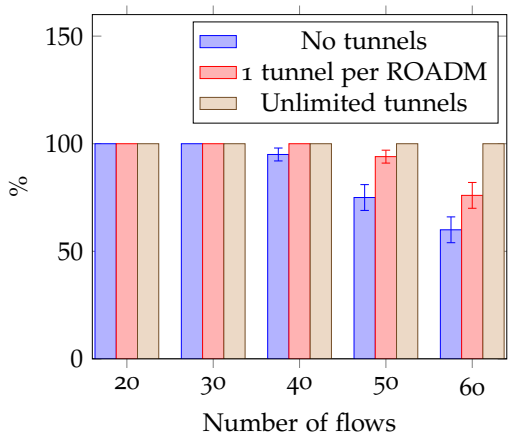

Figure 4.8: Percentage of flows that received the requested bandwidth guarantee.

additional wavelength.

4.8 Conclusion

The XTEF framework is capable of application-driven traffic engineering in a WAN through the following contributions. First, OneSwitch achieves topology virtualization and allows end-users to program the network without having access to the physical devices. By controlling the entire WAN as if it were a single Ethernet switch, an end-user application controller can create multiple paths between different locations with simple OpenFlow or XMPP statements.

Second, by replacing OpenFlow with XMPP as the northbound interface between the WAN controller and the application controllers, we allow for the latter to specify bandwidth and latency requirements for each flow. Below OneSwitch, the proposed traffic provisioning algorithm uses MPLS forwarding and ECMP to satisfy the demands of the application controllers.

Third, the DTS algorithm allows for additional network provisioning when short-reach links reach a load of 70% of their capacity. This algorithm requests an ONOS application to create a WDM tunnel between the two closest ROADMs to

bypass the busy IP router.

Our evaluation of OneSwitch shows that the delay introduced by the virtual layer is small. It also shows that OneSwitch can host up to 50 application controllers simultaneously. Our results also show that XMPP guarantees latency requirements for all received flows. In contrast, when the application controller does not specify latency limits and routing is done using ECMP, only 50% of flows receive satisfactory latency. Finally, our results also show that DTS can increase the network capacity by more than 10% using a single additional wavelength at each ROADM.

As future work, we will investigate the following two problems. First, there is a trade-off in cost between having a wavelength available instead of using it permanently. We argue that the advantage of using short-term WDM tunnels allows for the network to adapt to traffic conditions and use the tunnel in a different way depending on the network load. The second research direction involves the number of ROADMs to include in the tunnel. In our current implementation, only two-hops tunnels are created because it allows to aggregate more flows in the tunnel. Indeed, if the tunnels were longer, it would be less likely to identify more flows sharing the same path. However, we will experiment with using end-to-end tunnels as well to compare against two-hop tunnels if a better performance can be achieved.

Chapter 5

Internet scale: Intra-domain cut-through switching in MobilityFirst

5.1 Introduction

Mobile devices are becoming dominant in current networks and significant core architecture changes have been proposed to support them. Current protocols such as TCP/IP were not designed with mobility as a key design requirement. The inferior performance of these protocols in highly mobile networks and the increasing number of mobile devices has motivated the research community to design Future Internet architectures (FIA) that consider mobility as a key design requirement ([3, 214, 215, 216, 217, 218, 219, 220]).

MobilityFirst [3, 13] is a project funded by the NSF FIA and FIA-NP programs that proposes a mobility-centric architecture for the future internet. MobilityFirst supports secure identifiers that inherently support mobility and trustworthiness. These mechanisms greatly enhance the support of mobile devices in the network. In the MobilityFirst architecture, data is transmitted between adjacent routers in a hop-by-hop manner. Entire chunks of data are received at the next hop before being forwarded again. Also, routing decisions are performed at each hop to ensure proper delivery if a node has disconnected and connected to another point of the network. However, this process also increases the delay needed to send data in a hop-by-hop manner [3]. Furthermore, certain segments of the network are

stable and allow exceptions to the storage and routing delays. If we know that a node will remain connected to the same access point for a period of time, we do not need to make routing decisions at every hop between the source and the destination. Also, segments within the core of the network are exempt of mobility requirements. In scenarios like this, it is possible to bypass the routing layer of MobilityFirst.

In this chapter, we propose an SDN-based control plane to cut-through MobilityFirst routers across a single domain. Such technique can improve the performance of the network, because the delay of forwarding data at a lower layer is smaller. Another advantage is that it enables flow aggregation. Multiple data transmissions can be encapsulated in the same flow. To illustrate the advantage of flow aggregation, imagine a football stadium with 80,000 users accessing resources on the Internet. Without any bypass, routing decisions will be made at each hop of the way between the source and the destination for each of the 80,000 users. If we assume that the users will remain in the stadium for a period of time, we can bypass the routing layer. It is very likely that the routes between the sources on the Internet and the destinations in the stadium share more than one MobilityFirst router. For those sections of the network that are shared, we can forward all the data using the same rule (informally, we can think of it as "Tag all traffic going to the stadium with VLAN 1"). Once the data reaches the last hop of the bypass, each packet is routed to its specific destination accordingly. Therefore, using a small number of rules, we can forward the traffic intended for all the users in the stadium.

To realize the cut-through switching capability, we first describe an OpenFlow-based implementation of intra-domain cut-through switching for MobilityFirst. OpenFlow is the most commonly deployed Software Defined Networking tech-

nology today. SDN decouples the data plane from the control plane in a network switch, by migrating the latter to a software based component. In an OpenFlow-based network, the controller can dynamically update the forwarding rules of a network device. The controller also has a centralized view of the network. Because of these capabilities, an OpenFlow-based network can be used to create, modify and delete layer 2 circuits to bypass the routing layer.

To evaluate the performance and scalability of the cut-through mechanism, we first run experiments on the ORBIT [4] testbed to evaluate the performance of a file transfer using real hardware. Next, we deploy a MobilityFirst network on the GENI testbed and we show how a faster transfer time is achieved and how the number of packets processed by the controller is significantly reduced. Moreover, we demonstrate the scalability of the solution by exploiting the flow aggregation capability of the network.

The contributions of this chapter are:

- We describe how the MobilityFirst architecture can be adapted to be deployed in an SDN/OpenFlow-based network
- We propose a new cut-through technique that, unlike the vast majority of related work, does not bypass an IP layer but a GUID-based routing mechanism.
- We address the challenge of a bypassing technique in an inherently mobile network architecture.
- We show how OpenFlow can be used to aggregate multiple data transfers using the same flow table rule to achieve higher scalability.

The remainder of this chapter is organized as follows. We begin by providing background information on MobilityFirst and an SDN-based implementation of MobilityFirst in section 2. In section 3 we survey the related work and in section 4, we describe how to bypass the L3 routing in MobilityFirst using L2 VLAN switching. In section 5 we explain how OpenFlow can be used to achieve the bypass using VLAN switching. In section 6, we evaluate the proposed solution and we conclude in section 7.

5.2 Background and related work

Next we provide an overview of the MobilityFirst architecture. We also describe the ORBIT testbed that was used to run the experiments.

5.2.1 Overview of MobilityFirst

The MobilityFirst project proposes a clean slate redesign of the Internet architecture [3]. This design aims at supporting mobile devices and applications as the main elements of the network. Cisco has predicted that by 2014, wireless devices will account for more than 60% of IP traffic [221]. The current IP protocol was not designed for mobile applications and the emergence of such traffic offers an opportunity to evaluate what should be the purpose of functionality of the network [3].

Figure 5.1 shows the main building blocks of the MobilityFirst architecture. MobilityFirst provides three meta-level services: the global name resolution service (GNRS), the name-based services and the optional compute layer plug-ins. MobilityFirst also provides three core transport services: the hybrid GUID/NA global routing service, the storage aware routing (GSTAR) and the hop-by-hop

transport [3]. In this background section we focus on explaining how routing is performed in MobilityFirst.

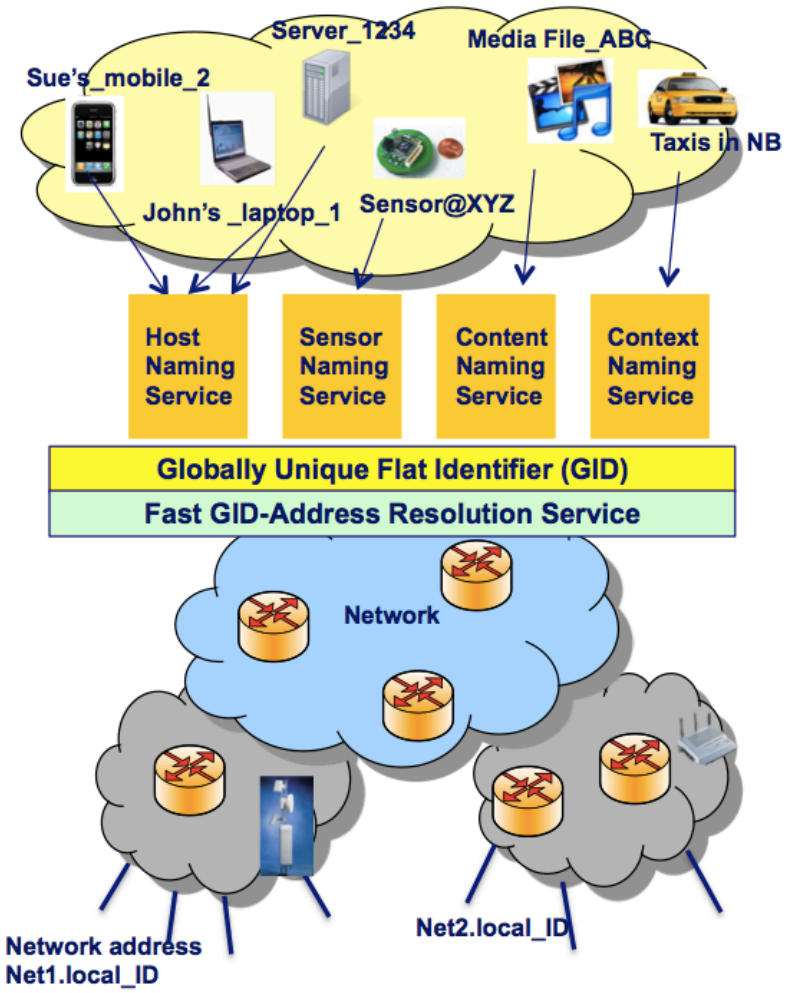

Figure 5.1: Basic Protocol Building Blocks in MobilityFirst. Figure was redrawn by co-authors (Source: Raychaudhuri et al. [3]).

Storage-assisted segmented data transport: MobilityFirst uses generalized storage-aware routing

(GSTAR) [222]. In Fig. 5.1, suppose that a host wishes to send data to John's laptop. First the host should acquire Johns GUID. Then a packet is sent with the GUID as the destination. A MobilityFirst node resolves the GUID using the GNRS and

obtains a list of NAs where the destination is connected to the network. The router sends a packet containing a destination GUID, a service identifier and a list of NAs. At each hop, a router will decide if the NA is within its reach or if the data must be forwarded to another router.

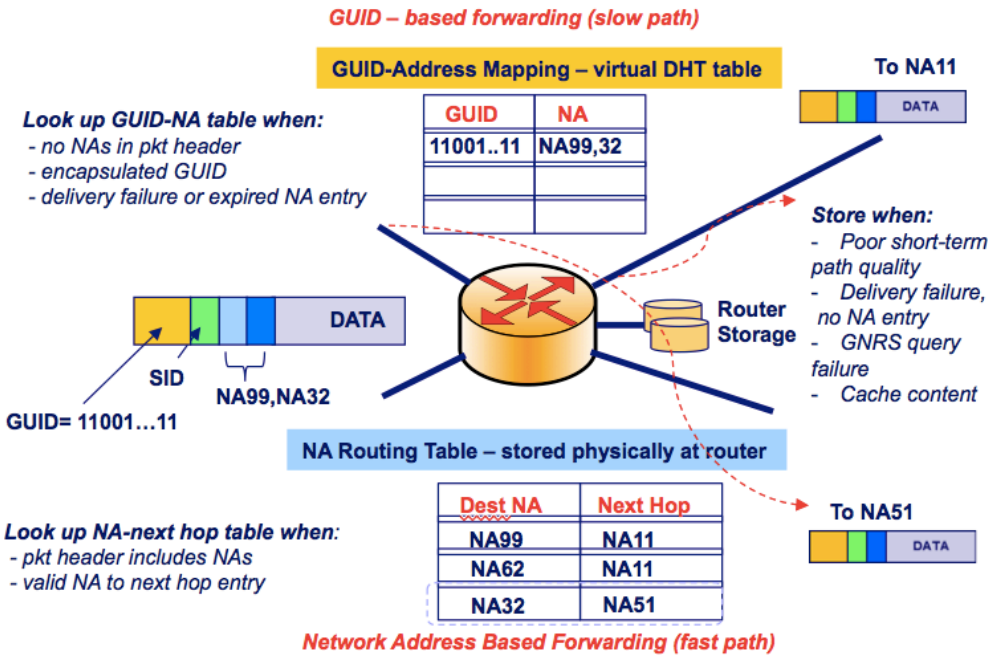

Figure 5.2: Hybrid GUID/NA packet headers in MobilityFirst. Figure was redrawn by co-authors (Source: Raychaudhuri et al. [3]).

MobilityFirst uses a hybrid name/address based routing to achieve scalability. The number of GUID objects is expected to be in the order of billions, but network addresses are expected in the order of millions. By mapping GUIDs to NAs, routing is greatly simplified [3]. Figure 5.2 shows how GUIDs and NAs are used during the routing process.

Another important feature of MobilityFirst is that the transmission of data in a hop-by-hop manner to support mobility. In this architecture, the entire file is received at each hop before transmitting it to the next one. Using this approach, it

is possible to do storage-aware routing and late binding [3].

5.2.2 Software defined networking implementation of MobilityFirst

Next we describe an SDN implementation of MobilityFirst provided by Krishnamoorthy [223]. Our implementation of the cut-through switching capabilities are built on top of the prototype described below.

In a MobilityFirst network, data is split into entities called "chunks" before being transmitted. The size of a chunk can be anything ranging from MTU size of the link to large values like 64 MB or 128 MB. Each chunk is then made up of several packets (each packet being of the MTU size, 1500 bytes in case of Ethernet link). Suppose host1 wants to send a 5 MB file to host2. First, it splits the file into chunks (let's assume each chunk is 1 MB). So host1 now has 5 chunks, and each of those chunks has approximately 700 packets (of 1500 bytes each). When host1 transmits each chunk to MFRouter1, only the first packet of each chunk has the routing header (as in the destination GUID, service ID, etc.).

In our SDN implementation of MobilityFirst, the network controller is responsible for finding a path to transmit all the chunks from the source to the destination. When the first packet of each hop arrives at a switch, there is no forwarding rule for it. Therefore, the controller must perform several tasks. First, it must use the destination GUID of the packet to find the destination in the network. Second, it must compute which switch is the next hop of the path. Third, it must push a rule into the switch so that all the data of that chunk is forwarded to the next hop. This process is repeated for each chunk of data.

Finally, we describe briefly the two testing environments: the ORBIT testbed and the GENI testbed.

5.2.3 ORBIT testbed

The Open-Access Research Testbed for Next-Generation Wireless Networks (ORBIT) [4] is an emulator/field track network testbed [4]. ORBIT emulates wireless networks with customizable topologies and channel modes. ORBIT is funded by the NSF and operated by WINLAB, Rutgers University.

Using ORBIT, different users can share the same testbed. Using an online scheduler, a user reserves a testbed for a period of time. The user can then create or load an image into the nodes, execute experiments, save the images and then release the testbed for another user. The measurements of the experiments are stored in a database that can be queried after running the experiments.

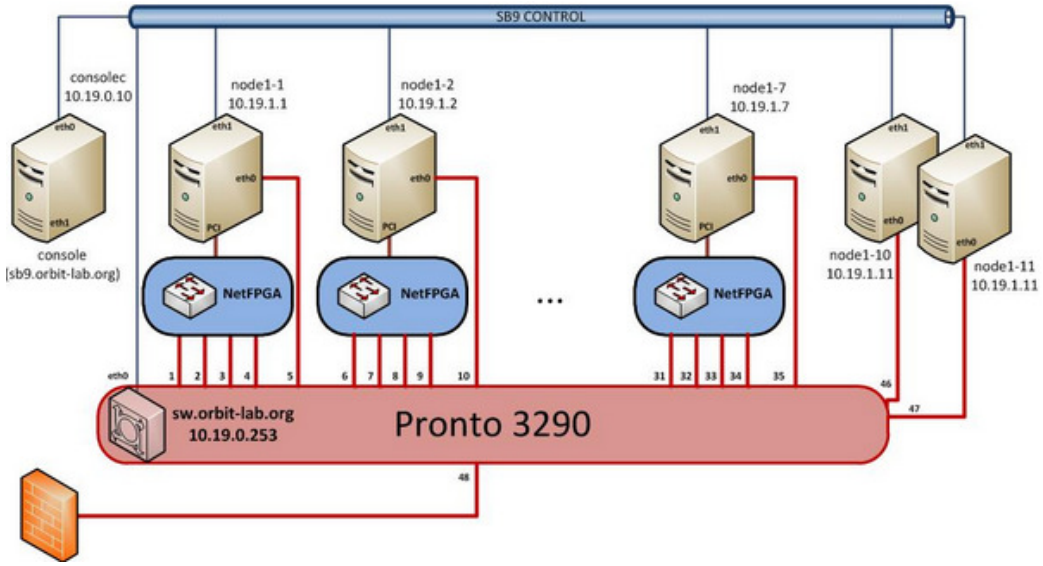

Figure 5.3: Diagram of the SB9 testbed. Figure was redrawn by co-authors (Source: ORBIT [4]).

One of the testbeds of ORBIT, the SandBox9 (SB9) is dedicated to experiment with OpenFlow. Figure 5.3 shows the design of the SB9 testbed. All nodes are connected to a Pronto 3290 switch and one of the nodes can be used as the

OpenFlow controller.

5.2.4 GENI testbed

The Global Environment for Network Innovations (GENI) is an NSF-funded initiative to enable large-scale network experiments. GENI provides a test bed with nodes across more than 50 aggregates [179]. Hosts deployed in the GENI testbed can implement any Layer 3 protocol to communicate, as Layer 2 connectivity is provided between sites. This greatly simplifies the large-scale deployment of MobilityFirst using GUID-based routing instead of IP.

Before describing our proposed solution, we first survey related work to motivate the need for a routing layer bypass.

5.2.5 Related work

Previous studies have proposed bypassing the IP routing layer to achieve benefits such as energy savings, bandwidth protection, cost reduction and better network performance.

Melle et al. [224] describe an optimization of an IP over WDM network architecture based on bypassing the IP routing layer to save deployment costs. The authors show how the cost per interface is cheaper at the WDM layer. They argue that the number of IP-layer interfaces needed can be reduced by bypassing the IP routing layer when the capacity of intermediate hops is sufficient to make the bypass cost-effective. Melle et al. show that a bypassing technique does not make a significant difference when the traffic volume is low (0.15-0.30Tbps) but can save up to 30% of costs when the traffic volume increases.

Lui et al. [225] address energy saving in IP over WDM networks through IP routing bypassing. The authors design an energy-minimized IP over WDM

network by combining lightpath bypass and router-card sleeping techniques. They argue that turning off ports on a port-by-port basis is more complex than turning off an entire router card (which makes all ports in that card sleep). By creating bypasses at the WDM layer, it is possible to turn of a router card and the authors show how the energy consumption can be reduced by 40%.

Karol [226] proposes a distributed algorithm to set up maximal-length circuits to bypass IP routers. These circuits can be created dynamically or in advance. The algorithm minimizes the number of IP routers needed, reduces the network energy requirements and reduces the end-to-end packet latency.

Although MobilityFirst does not use IP as the routing protocol, these studies show some of the advantages of bypassing the routing layer in terms of cost and energy saving, as well as performance. Although the related work suggests implementing the bypass at the optical layer, for convenience we use an electronic switching-layer approach using OpenFlow. A detailed explanation of the solution is provided in the next section.

5.3 Bypassing the routing layer

In this section we discuss how to bypass layer 3 routing in MobilityFirst. First we describe the challenges of a bypassing technique. Next, we explain how to bypass the routing layer using layer 2 VLAN switching.

5.3.1 Challenges and design goals of a bypassing technique

Several challenges must be considered to bypass the routing layer in MobilityFirst: when to setup circuits and for how long; how many circuits are needed and their granularities and how to implement automated circuit creation in the MobilityFirst

context.

- Mobility: It is important to keep in mind that nodes are assumed to be mobile. A circuit reservation solution cannot assume that a node will remain at the same location.
- Efficiency: The overhead of setting up circuits should be low and the circuits should significantly improve the performance of end-to-end deliveries.
- Scalability: The MobilityFirst architecture should be able to support a large number of users. The delay of setting up circuits must remain low for a large number of users and the number of circuits reserved should be able to scale as well.
- Reliability: A successful delivery must be ensured, even if a circuit exists and the node location changes or the link fails.

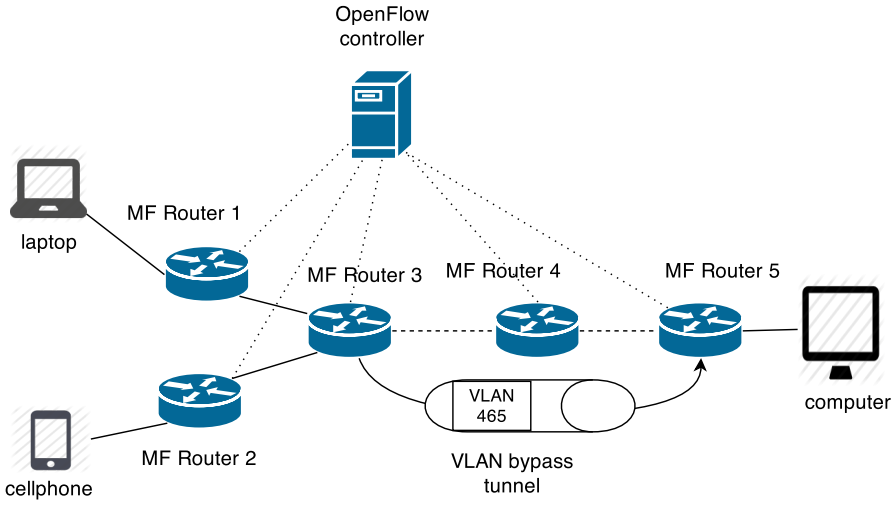

Figure 5.4: Example of a bypass in MobilityFirst.

5.3.2 Bypassing L3 using Layer 2 VLAN switching

One way to bypass Layer 3 routing is to create Layer 2 circuits using VLAN tags. Recall that MobilityFirst works on a hop-by-hop basis. A MobilityFirst router sends the data to the next router and this is repeated until the destination is reached. Using this bypassing technique, a circuit can be created at L2 between the host and the destination. In order to do this, a path must first be found at the first hop to the destination. Next, a forwarding rule must be added in all forwarding elements so that the traffic is automatically forwarded to the next hop.

To identify each flow, a VLAN tag can be used. Since VLAN tagging faces well known scalability issues, a more scalable solution can be implemented using Virtual eXtensible Local Area NetworkS (VXLAN) [227]. VXLAN was proposed to meet an important requirement of virtualized data centers: to have a layer 2 infrastructure that can scale at the datacenter scale [227]. In our implementation we use VLAN tagging because this is supported by OpenFlow, but extending the protocol to support VXLAN is an alternative to increase the scalability of the bypassing functionality.

Figure 5.4 shows an example of a bypass. One source is attached to the MobilityFirst router 1 and another one is attached to the router 2. Since all destinations are attached to router 5, then a bypass between routers 3 and 5 can be created. Once the bypass is pushed, no routing operation is performed at router 4. The way to create this bypass is to add a forwarding rule to router 4 that forwards all traffic with a given VLAN from router 3 to router 5. In router 3, when we forward packets belonging to the bypassed flow (source is S1 or S2 and destination is D1 or D2), we tag them with the same VLAN number. When the data reaches router 5, routing decisions are taken based on the destination GUID of each packet.

This ensures that different routes are chosen for destinations D1 and D2.

This design enables flow aggregation. In Fig. 5.4, a single rule in router 4 can be used to send data to multiple destinations. In any scenario where many destination nodes are connected to the same router, this feature is key to ensure the scalability of the system. In a more realistic topology, it is likely that end users are connected to edge routers and these devices are interconnected through other devices across the network. Flow aggregation enables connecting multiple users connected to the same edge routers using a small number of rules. By reducing the number of rules needed at each hop, we significantly increase the scalability of the network.

As discussed earlier, this solution should also take mobility into consideration. If a circuit exists and a node changes the location, the delivery must still be guaranteed. If a bypass is in place and a node disconnects from the network, we must ensure that the current chunk of data is delivered to a MobilityFirst router that will find a new route. Also, subsequent chunks of data should not be sent through the bypass. In the example shown in Fig. 5.4, suppose the destination node D2 disconnects from router 5 and reconnects to router 4. When the data reaches router 5, it is still possible to locate node D2. By querying the GNRS about the location of the GUID of D2, we can learn that the location of the node has changed. Next, we can forward the packets to the next hop and we can also remove node D2 from the bypass.

This solution should be efficient as well. There is a trade-off between the time and resources that it takes to create a circuit to bypass L3 and the delay required at L3 routing. If a circuit is to be created, the time it takes to set it up should be significantly shorter than the time saved by bypassing L3. Also, the controller should require an acceptable amount of resources to detect when and how to

create circuits. If the controller's performance is significantly decreased because of this, then the solution is not acceptable.

Finally, reliability must be taken into consideration. As we mentioned above, the delivery of the message must be guaranteed. If one of the links that are part of the bypass path fails, the data must be forwarded to a MobilityFirst router and the bypass must be deleted.

Another way to implement this traffic engineering technique would be to use multi-protocol label switching (MPLS) [228]. Using MPLS, the ingress edge router computes the route from source to destination, communicates this route to all the routers involved and inserts a label into each packet. Successive hops can then forward packets based on the label. Note that this technique does not completely bypass the routing layer, as packets must still be processed by routers. In our approach, there is no need for the packets to be processed at the routing layer and all packets can be forwarded by simple L2 switches.

5.3.3 Deciding when to create a bypass

One of the major challenges of this implementation is deciding when to create a bypass. We envision two alternatives: proactive and reactive bypass creation. A proactive implementation is easier, but it requires that the nodes provide prior notice. A MobilityFirst node could notify that a given number of bytes will be transferred to a destination. If this information is known, the controller can create a Layer 2 circuit between the sender and the receiver to ensure a faster communication. The advantage of a proactive approach is that the rules can be pushed in advance and the network controller does not need to make dynamic changes once the data starts flowing. However, a proactive solution only works when the information of the data transfer is known in advance, which is not always

the case.

When no previous information is available, the bypass must be created in a reactive manner. In this case, the controller must dynamically identify for which flows to create a bypass. One possible approach is for the controller to store information about the location of devices. If multiple flows for a single destination are repeatedly forwarded to the same hop, the controller can assume that the node will not change the location for a period of time. Then, a bypass can be created for data sent to that device. The advantage of this approach is that it is completely dynamic and no previous information is required about the characteristics of the communication. On the other hand, the controller has to do more processing and this increases the delay. Also, the controller must store additional information and this can compromise the scalability of the solution.

5.3.4 Deciding when to remove a bypass

We also address how to remove a bypass. Once again, this can be done proactively or reactively. If a bypass was proactively created and we have information regarding when the data transfer will end, then the controller can automatically remove the bypass at a given time. However, a reactive solution must exist at any time, in case a disconnection happens. The controller can monitor which nodes get disconnected from the network. For each disconnected device, a clean way to remove the bypass is to maintain the flow rules for the current chunk, so that all the data of that chunk reaches the destination network device. However, for the next chunk, the standard data processing is applied and a hop-by-hop route is used.

In the next section we describe the implementation details of the proposed solution.

5.4 Implementation using OpenFlow

In this section we show how OpenFlow can be used to bypass L3 routing using L2 VLAN switching. We discuss how to push a circuit using OpenFlow and we discuss how we address the challenges mentioned in the previous section. This work is built on top of an existing SDN-based implementation of MobilityFirst [223]. Our contribution consists of adding the bypass functionality to this code. We begin by explaining how the existing prototype works and next we explain how we implemented the bypass technique.

5.4.1 Mapping chunks to VLANs

We first describe some technical details of our OpenFlow-based implementation of MobilityFirst. In MobilityFirst, data is split in chunks and packets include information to know which chunk they belong to. For each chunk, the first packet is forwarded to the controller and a flow is pushed into the switch so that all the remaining packets of that chunk are forwarded to the next hop. To make this compatible with OpenFlow, the routing header is introduced in the L3 Source IP Address field. The controller can then parse the data of the first packet and use the routing information to compute where to forward all the packets of this chunk. When the next destination has been decided, a new flow rule is pushed to forward all the packets in this chunk to the next hop. To match all packets to the inserted rule, the hop ID is used as a VLAN tag. This hop ID identifies all packets belonging to one chunk across the link. Coming back to the example, for each of the five chunks, all the 700 packets will have the same hop ID and this hop ID is also inserted as a VLAN tag in all the packets. If we use incremental hop IDs, then in the above scenario, all packets in chunk 1 will have hop ID 1, those in chunk 2

will have hop ID 2 and so on. This helps us identify which chunk a specific packet belongs to (since the packets themselves do not have any such information, except for the first packet of the chunk).

The key to achieve the bypass is to push a flow rule into all the switches between the source and the destination instead of only for one hop. In an OpenFlow-based network, the controller is aware of the topology. Thus, an end-to-end path can be found and all forwarding devices can be reached from the controller to push a new flow rule. To find a path between the host and the destination, we need to know the Layer 2 MAC address and the input and output ports at each hop. Next, specific flow entry rules can be pushed at each switch. The VLAN tag is the same for all the switches, but the source and destination MAC addresses and ports are different.

5.4.2 Bypassing functionality

In the current OpenFlow-based deployment of MobilityFirst, the entire route between the host and the destination is computed using the service provided by the Floodlight controller. Krishnamoorthy also implemented a mapper between GUID numbers and MAC addresses. Given a GUID, the controller can find the MAC address associated to that node. Therefore, the information on the entire path is available. To achieve a bypass, we collect the following information for each hop between source and destination: VLAN ID, destination GUID, in-port and out-port. The existing prototype pushes a flow rule to the first switch of the path only. To implement a bypass, the controller pushes a flow rule into each switch using the proper port values and keeping the same VLAN id and destination GUID. As a result, all the packets of the current chunk are forwarded at layer 2 until they reach the final hop. By creating a bypass, we ensure that the intermediate switches know

how to handle each packet and do not need to forward data to the controller for each new chunk.

5.4.3 Discussion: Challenges addressed

We mentioned four key challenges for the bypassing technique: mobility, efficiency, scalability, reliability.
Next we discuss how our solution addresses those points and what are the challenges that must still be overcome.

Our solution addresses mobility by routing packets at the end of the bypass. If a bypass goes from router 3 to router 5, then the data will be received at router 5 and a route will be computed for the GUID or NA. If a node has connected to a different location of the network, the controller can query the GNRS for the new NA and find a new route. We expect this to be relatively infrequent since a bypass should be pushed only when a node is not expected to move. However, if the device does move, a new route can always be found. One challenge that remains is to actually be able to push bypasses only when the nodes will remain in the same location. Otherwise, the delay introduced can become significant.

In terms of efficiency, OpenFlow is a convenient approach to dynamically manipulate forwarding rules. The application running on the controller can proactively or reactively modify the flow table of one or more switches. Therefore, creating or deleting a bypass can be done efficiently. If a bypass is created proactively, the controller only needs to act at a specific time. If the bypass is to be created reactively, the controller must incur a delay to process the first packet of each chunk to decide if a bypass is needed or not. We expect this delay to be acceptable, as only the first packet of a chunk must be processed. However, an interesting scenario occurs when there is a failure during the transmission and

the distance between the start and end of the bypass is far. In this case, the time to send the contents again can introduce an important delay. This raises the question of whether to bypass a large number of hops or if it is more convenient to keep the number of hops small.

Regarding scalability, we discussed earlier how flow aggregation can help the network scale. Using OpenFlow, we can easily update any flow entry of a device. If a bypass already exists, the controller can easily modify the rule so that the bypass includes a new source or a new destination. If a bypass must be created, it can be done efficiently too. Also, the fact that the controller has a centralized view of the network allows the application to be aware of changes in the topology fairly quickly. This simplifies updating a bypass when necessary. On the other hand, the limited number of VLANs and the size of the flow table are known limitations in an OpenFlow-based network. It is important to evaluate if these limitations significantly impact the scalability of this deployment.

Finally, our solution also addresses reliability because the architecture is still storage-aware. In MobilityFirst, the data is stored at each hop before being transmitted. If the controller detects that the data is not properly delivered to the destination router at the end of the bypass, it can use a hop-by-hop delivery. It is important to evaluate how often does this occur in real-life scenarios, in order to measure the impact on the performance of the network.

5.4.4 Discussion: Centralized control plane

One key feature of OpenFlow-based networks is that the control plane is centralized. The advantage of a centralized control plane is that the controller has a network-wide knowledge of the network. This simplifies reacting to failures and creating new paths when necessary. The OpenFlow protocol includes features that allow

a controller to listen to switch events and thus learning about broken links and connected devices. The main drawback of a centralized control plane is the scalability challenge, as well as becoming a single point of failure. To overcome this, distributed control plane architectures such as HyperFlow [229] and ONOS (Open Network Operating System) [209] have been proposed.

Also, although OpenFlow does not provide support for interdomain circuit creation, recent proposals such as software defined Internet exchange (SDX) [230] could enable multi-domain bypass functionality. In SDX, *exchange points* are devices in the topology where a network owner could grant access to the forwarding table to an authorized neighbour operator. In this scenario, multiple controllers operated by different parties can coordinate to enable a wide-scale SDN-based network.

5.4.5 Expected improvements using OpenFlow 1.4

The implementation described in this section could be improved using the recently released OpenFlow 1.4 specification. First, the scalability issues of using VLAN tags can be solved using the IPv6 fields to perform GUID-based addressing and adding the chunk id. The IPv6 source and destination fields allow for a much larger number than the VLAN tag. They can be used to store a chunk id, a GUID or a network address. A distributed control plane is also easier to deploy using OpenFlow 1.4, since a switch can be connected to multiple controllers at the same time. One of the main challenges of OpenFlow-based deployments is to deal with a single point of failure (the controller) and having a distributed control plane mitigates this problem. Although these improvements were already possible using OpenFlow 1.3, one addition of OpenFlow 1.4 is the ability to control optical ports. Two new OpenFlow messages have been added to configure and monitor either Ethernet optical port or optical ports on circuit switches [231]. This allows for a

much smarter bypassing technique, capable of deciding whether a bypass is more efficient at the optical layer or at the Ethernet layer. We expect to explore this in future work.

In the next section we evaluate the proposed solution in terms of performance and scalability.

5.5 Results and analysis

In this section we describe two experiments to evaluate the performance of the bypassing technique. First, we use a simple topology in ORBIT [4] with two nodes and one single switch and we measure the delays incurred by the controller and the time it takes to transfer a file en a single-switch topology. Next, we use a virtual topology in GENI [179] to compare the performance of the network with and without a bypass.

5.5.1 Single-switch network

We deploy in ORBIT [4] the topology shown in Fig. 5.5. One node acts as the source and one as the sink. The source node runs a script that sends files of different sizes using a chunk length of 4MB. Each chunk consists of 4096 packets and each packet carries a payload of 1024 bytes. A third node acts as the OpenFlow controller.

Figure 5.6 shows the total transfer time for 20, 100, 400 and 1000 chunks of data. The transfer time grows linearly at an approximate ratio of 100 chunks per second. In order to motivate the need for a cut-through technique, we also measure the total needed by the controller to process the first packet of each chunk. Recall that the first packet of each chunk goes to the controller and a new flow rule is pushed

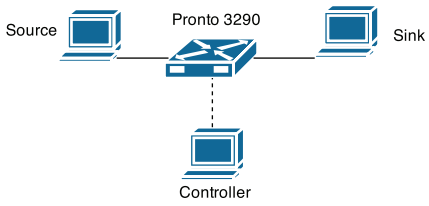

Figure 5.5: Single-switch topology deployed in ORBIT

into the switch. Therefore, for 20 chunks of data, 20 packets go to the controller and so on. Also, Figure 5.7 shows the sum of all the time that the controller needs to process first packets. The results grow linearly for an average delay of 688ms for every 1000 chunks of data sent.

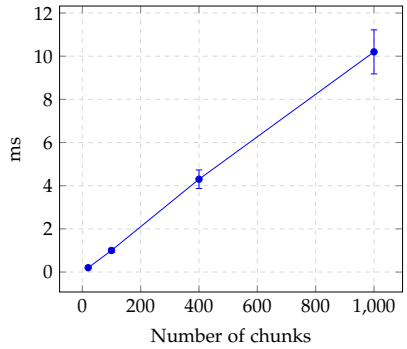

Figure 5.6: Total transfer time for a varying number of chunks

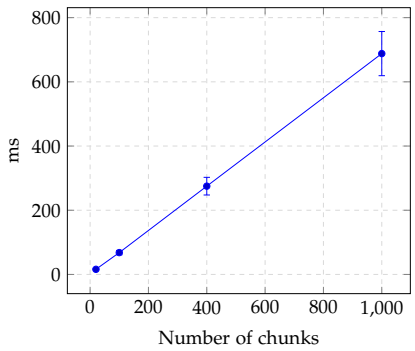

Figure 5.7: Total delay at the controller for a varying number of chunks

Now that we measured the delay incurred by the controller, next we move on to a multi-switch network where a bypass can be implemented.

5.5.2 Multi-switch network

We deploy in GENI [179] the topology shown in Fig. 5.8. In this topology we use Open vSwitch instances instead of hardware switches. All resources are deployed in the Kentucky ProtoGENI aggregate and we use emulab-xen images for all nodes (all nodes use 64-bit CPUs and run Ubuntu version 12.04). The controller and the Open vSwitch instances communicate using the OpenFlow 1.0 protocol.

5.5.2.1 Reducing the transfer time

In this experiment, we compare the time needed to transfer a file with and without a bypass. We implement a proactive bypass that includes all switches between the source and the sink. Figure 5.9 shows the transfer times with and without bypass between multiple sources and the sink. The transfer time is faster when a bypass is created between the source and the destination (average improvement of 10% when sending 1000 chunks of data).

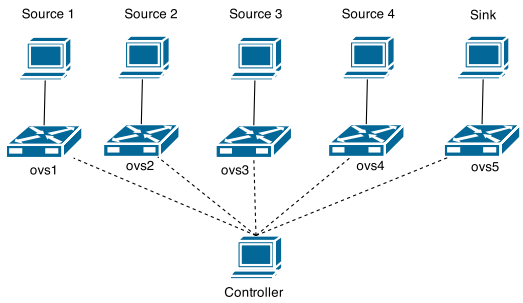

Figure 5.8: Experimental topology deployed in GENI

Another crucial advantage of bypassing the GUID-based routing layer is that the number of packets received by the controller is significantly reduced. In our

experiment, we create a bypass between the first and the fifth switch. By doing this, we save one packet for each chunk that uses the bypass. Figure 5.10 shows the reduction in the number of packets received by the controller. The number of packets decreases because once a bypass is implemented, intermediary switches do not need to forward the first packet of each chunk to the controller, since the forwarding rule has already been pushed. To better illustrate this, one can think that when a bypass is pushed between two switches, then these two devices the source and the destination of the hop-by-hop transfer and the data travels from one switch to another without further participation from the controller.

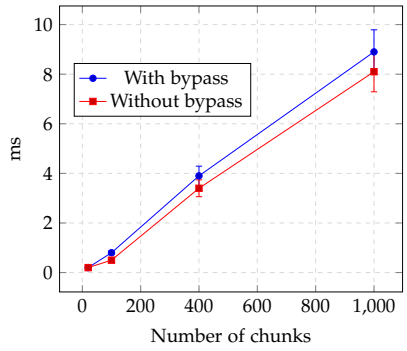

Figure 5.9: Total transfer time with and without bypass. 95% confidence intervals are shown.

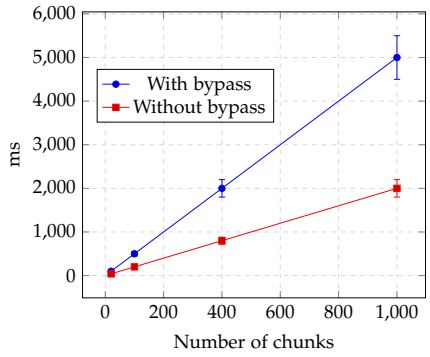

Figure 5.10: Number of *packet_in* messages received by the controller with and without bypass.

5.5.2.2 Scaling through flow aggregation

Flow aggregation is key to the scalability of our framework. We explained in Section 3.2 how a single rule can be used to carry multiple flows. In Figure 5.11, hosts 1, 2, 3 and 4 send data to the sink. Without flow aggregation, the controller must push a flow rule for each chunk in each transfer. For example, in Fig. 5.8, when all hosts send 20 chunks to the sink, then the *ovs*1 switch needs 20 rules for the traffic coming from host1, but the *ovs*2 switch needs 40, the *ovs*3 switch needs 60 and so on, for a total of 280 flows. In this experiment, however, the controller knows that all traffic is heading to the sink. Therefore, only 20 rules per switch are sufficient to carry all traffic.

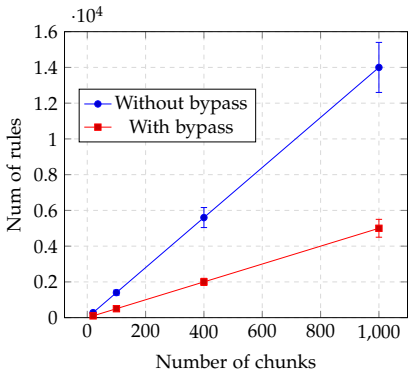

Figure 5.11: Number of flow rules pushed with and without aggregating flows.

5.5.3 Routing and bypassing in a mesh topology

Lastly, we experiment using a mesh topology to show how the implementation exploits the services provided by the Floodlight controller to find a route between the source and the destination. Also, we show how traffic coming from more than one switch can also be aggregated. The topology is shown in Fig. 5.12.

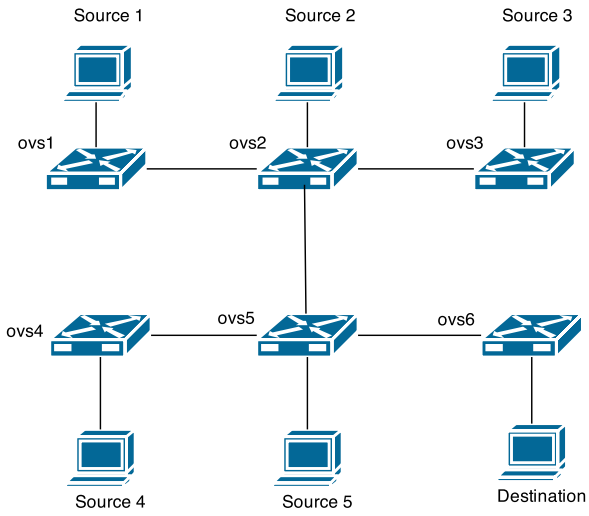

Figure 5.12: Experimental mesh topology.

The experiment consists of transferring a file from sources 1 and 3 to the destination. Our implementation uses the services provided by the Floodlight controller to find a path between the sender and the receiver. Floodlight finds that the appropriate routes are: ovs1, ovs2, ovs5, ovs6 to transfer between source 1 and destination, and ovs3, ovs2, ovs5, ovs6 to transfer between source 3 and destination.

Given that both paths share nodes 2, 5 and 6, next we experiment bypassing node 5. First, individual bypasses are setup, one for all traffic coming from source 1 and another for all traffic coming from source 3. In this scenario, the gain in transfer time when the bypass is pushed is similar to the results shown in Fig. 5.9. Here we focus on analyzing if the number of flows needed can be reduced. In ovs1, ovs3 and ovs6, the number of flows needed is the same as shown in Fig. 5.11. However, in ovs2 and ovs5, we achieve a reduction of 50% since the same flows are used to carry all traffic. Although this topology is small, it clearly shows how the number of flows can easily be reduced when two paths have more than two

hops in common.

5.6 Conclusion

In this chapter we proposed an approach to bypass the routing layer in MobilityFirst using cut-through switching and OpenFlow. The advantage of such a bypass is to eliminate the delay introduced at that layer. We discussed how to use OpenFlow to bypass layer 3 routing in MobilityFirst using layer 2 VLAN tagging. Instead of pushing a flow rule to one switch only (as would be done to ensure a hop-by-hop communication), we push rules into all the switches of the path between the source and the destination. By doing this, we ensure that all data is forwarded at layer 2. We also discussed how this technique enables flow aggregation. By managing several data transfers using a small number of forwarding rules, we increase the scalability of the network.

The evaluation using the ORBIT testbed showed that the controller introduces a delay of 688 milliseconds for every 1000 chunks of data sent, in average. This delay is due to the first packet of each chunk having to be processed at the controller. Our experiments using the GENI testbed show how the number of packets processed by the controller can be significantly reduced by the cut-through technique proposed in this chapter. Finally, we show how the flow aggregation capability of SDN-based networks can be used to make this solution scalable. By aggregating flows from multiple sources to a single destination using the same flow rules, the number of rules that must be pushed into the switches is also reduced.

Given the performance gains achieved at intra-domain scale, in the next Chapter we investigate how to implement inter-domain cut-through switching in MobilityFirst.

Chapter 6

Internet scale: Inter-Domain Routing with Cut-Through Switching in MobilityFirst

6.1 Introduction

In this chapter we investigate how MobilityFirst can benefit from cut-through switching tunnels across multiple domains, given the benefits of intra-domain tunnels demonstrated in Chapter 5.

To motivate the need for inter-domain cut-through switching in SDN, we first model the dynamic creation of inter-domain tunnels as a linear optimization problem. Particularly, we describe this problem in the context of inter-domain SDN. The problem minimizes the total transfer time while considering the costs of creating inter-domain tunnels. Using this problem formulation, we demonstrate how inter-domain controller latency plays a key role on how tunnels are created. Indeed, we show how inter-domain tunnels are better when the latency is small, but intra-domain tunnels are better otherwise. Next, we propose a greedy heuristic that considers the inter-controller latency to decide how to create tunnels that scales much better than the optimization problem formulation. To the best of our knowledge, our work is the first one to formulate and solve the creation of inter-domain tunnels in SDN.

After that, we propose a routing framework for MobilityFirst that enables

dynamic inter-domain cut-through switching. The framework is based on the following design requirements: inter-domain topology visibility, naming the tunnels as network objects and per-flow traffic engineering. We first describe how Mobility-First uses aggregated nodes and virtual links to share topology information across domains using network state packets. Next, we propose a novel technique that names inter-domain tunnels as network objects to simplify how domain controllers create and maintain tunnel information. Finally, we incorporate the heuristic into this routing framework to decide when is it beneficial to create a new inter-domain cut-through tunnels and which flows should be forwarded through the tunnels.

To evaluate the proposed framework, we developed a prototype for Mobility-First using the GENI testbed based on the implementation described in Chapter 5. The results show that in-transit *packet_in* messages can be reduced by 75% using inter-domain tunnels. Furthermore, naming tunnels as network objects scales better than current protocols such as label distribution protocol (LDP) to setup tunnels. Finally, we also demonstrate the scalability of the solution by showing that incoming packets face a small delay even when the traffic load increases to up to 1000 flow requests.

The remaining of this chapter is organized as follows. We first survey the related work in Section 6.2. Next we describe the optimization problem and heuristic in Section 6.3. Next, we describe the routing framework in Section 6.4. After that, we describe thes traffic engineering techniques using by the framework in Section 6.5. Finally, we evaluate our work in Section 6.6 and conclude in Section 6.7.

6.2 Background and related work

In this section we include related work on inter-domain routing using SDN, inter-domain cut-through switching and inter-domain optimization studies. For an overview of MobilityFirst, please refer to Chapter 5.

6.2.1 Inter-domain routing using SDN

We first describe previous work on inter-domain routing using SDN. Existing studies include seamless integration of an SDN-based AS [205], outsourcing of routing [207] and a framework to evaluate the effect of centralization on the BGP convergence time [232].

Lin et al. [205] propose how an SDN-based AS can co-exist with IP-based ASes. The authors proposed a fully SDN-based where the controller communicates with other domains using BGP. Intra-domain is performed based on MAC addresses and edge switches match incoming packets based on IP prefixes. There are some differences between the architecture proposed by Lin et al. and our work. First, their framework exposes an entire AS as a single router that acts as a single hop in BGP. We aim for sharing more information about each AS (using aNodes and vLinks). Second, their AS is fully SDN-based, which means that the controller can program all network devices in the AS. Their results show that it is feasible to seamlessly integrate an SDN-based AS to the network.

Another paper by Lin et al. [206] proposes an information exchange mechanism between SDN controllers in different domains. The authors implement an Inter-domain Path Computation (IDPC) application and a Source-address based Multi-path Routing (SMR) application. Their work relies on message exchanging between controllers, whereas our approach attempts to use the GNRS as much as possible

to keep the exchange of messages to a minimum.

Konotris et al. [207] go one step further towards centralization. Instead of just centralizing routing information for a single domain, they argue that the next step is to outsource routing decisions to an external party. If multiple ASes do this, then the external party ends up being responsible for the routing mechanisms of multiple ASes. The authors point out significant advantages of having centralized control of routing within one or more AS, such as simplified policy enforcement and security. Although we do not aim at routing outsourcing, collaboration between ASes is certainly simplified when central entities can exchange information, as opposed to having multiple routers having to coordinate.

Finally, Gämperli et al. [232] proposed an emulation framework to experiment with multiple SDN-based ASes. The framework builds on top of Mininet and incorporates Quagga, a BGP software. As a use case, the authors evaluate the convergence time of BGP when a centralized control plane is used.

6.2.2 Inter-domain cut-through switching

Next we describe previous work on multi-domain scenarios where a cut-through switching circuit is created.

Yang et al. [233] present an inter-domain provisioning solution tested over the Dynamic Resource Allocation via GMPLS Optical Networks (DRAGON) and the Internet2 Hybrid Optical and Packet Infrastructure (HOPI). This application provides end users with Ethernet circuits over WDM paths. In the proposed solution, the negotiation between domains is achieved through network-aware resource brokers that are also responsible for the path computation. The main difference with our work is that DRAGON computes paths in a distributed manner, whereas we have a centralized control plane.

Guok et al. [234] propose OSCARS (On-Demand Secure Circuits and Advance Reservation System) [235], a software developed by the Energy Sciences Network (ESnet) [129] to provide end-to-end circuits for high performance networks. OSCARS relies on a centralized path computation engine to stitch a bandwidth-guaranteed path across domains. The amount of information shared by each domain with the path computation engine depends on each network administrator. This centralized system offers significant advantages, but at an Internet scale, it is unlikely to have multiple domains being programmed by a central entity. Hence, our approach is only centralized at an intra-domain scale, while inter-domain routing is still performed in a distributed manner across controllers.

6.2.3 Inter-domain optimization

Finally, existing studies have considered inter-domain routing as an optimization problem [236, 237, 238]. Tomaszewski et al. [236] consider the problem of bandwidth reservation on inter-domain links for different traffic classes. Roughan et al. [237] tackle the problem of traffic engineering with limited information shared across domain. Finally, Chamania et al. [238] explore how to achieve IP routing stability through dynamic creation of tunnels at the WDM layer. Our model of tunnels, described in the next section, is based on the formulation proposed by this work.

6.3 Dynamic creation of inter-domain tunnels

In this section, we model the dynamic creation of inter-domain tunnels as an optimization problem. The objective function minimizes the total transfer time, including control plane delays, while considering the different costs of creating

and maintaining inter-domain tunnels.

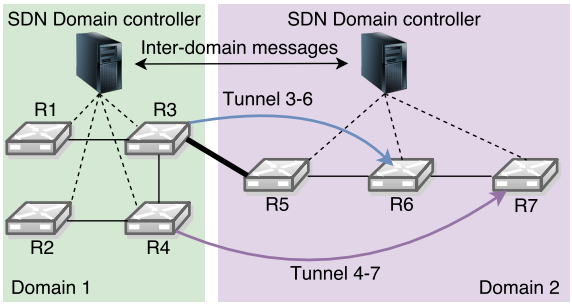

Figure 6.1: Sample network with two domains and two cut-through tunnels (3-6 and 4-7).

6.3.1 Assumptions

We make the following assumptions in the formulation of this problem. First, we assume that the computation of the tunnels is being performed by the controller of one of the domains. Moreover, we assume that such controller has visibility into the bandwidth of all links, including those belonging to other domains. In the framework described in Section 6.4 we propose a way to achieve this. Second, we assume that, when the optimization begins, no tunnels exist and there is no traffic flowing. We leave studying the steady-state scenario for future work. Third, we assume that the cost of inter-domain tunnels is computed based on the individual cost of each link traversed by the tunnel. Fourth, we assume that the flow rate and duration between each source and destination node are known. Finally, we assume that only one tunnel can exist between two routers, but the tunnel can carry more than one flow between different sources and destinations.

6.3.2 Settings

Consider a network of V nodes (OpenFlow-compliant routers) belonging to at least two different domains. The path between all pairs of nodes is known and the

optimization problem explores all combinations of using links individually or in tunnels.

We consider the following parameters:

- $\lambda_{s,d}$: Bit rate in Mbps between source s and destination $v \in V$;
- $\delta_{s,d}$: Duration in ms of flow between source s and destination $v \in V$;
- $c_{i,j}$: Capacity in Mbps of link i, j;
- $\psi_{x,y}^{i,j}$: Link i, $j \in L$ is used when a tunnel between x, $y \in L$ is setup (boolean);
- $\phi_{s,d}^{i,j}$: Link i, $j \in L$ belongs to the path between s and $d \in V$ (boolean);
- $b_{i,j}$: Time in ms needed to setup inter-domain tunnel when link i, j is used;
- $m_{i,j}$: Maintenance cost due to inter-domain messages to maintain a tunnel when link i, j is part of it;
- l_x: Latency in ms between switch $x \in V$ and the controller it is connected to;
- s_x: Time in ms needed by the controller to handle a packet_in message sent by switch $x \in V$;
- u_x: Time in ms needed by the initiating domain controller to compute a tunnel path;
- $w_{x,y}$: Inter-domain latency in ms between domain controllers to setup a tunnel between switches x and $y \in V$;
- T: Maximum number of tunnels allowed;
- M: Maximum maintenance cost allowed per domain;

- D: A flow duration must be at least D times longer than the time needed to create a tunnel in order to be routed through that tunnel. This parameter is used in constraint 6.6 and is explained below in more detail.

Likewise, we consider the following decision variables:

- $t_{x,y}$: A tunnel between nodes $x, y \in V$ is created.
- $f_{x,y}^{s,d}$: Flow between $s, d \in V$ is routed through a tunnel between nodes $x, y \in V$.
- $r_{s,d}^{i,j}$: Flow between $s, d \in V$ is routed through direct link i, j.

We also use two additional parameters to simplify the objective function:

- $\tau_{x,y}$: Time needed to setup a tunnel between switches x and $y \in V$. This parameter is computed as $\tau_{x,y} = \psi_{x,y}^{i,j} \times l_x + u_x + w_{x,y}$ (i.e. the sum of the switch to link latencies, the inter-domain controller latency and the time needed by the initiating controller to calculate the path).
- $\iota_{i,j}$: Time needed by the domain controller to handle a packet_in message when a flow is routed through the direct link i, $j \in V$. This parameter is computed as $\iota_{i,j} = 2 \times l_j + s_j$ (i.e. the round-trip latency between the switch and the controller added to the time needed by the controller to handle the packet_in message).

6.3.3 Problem formulation

For each flow, there is a known transfer time $\delta_{s,d}$. However, additional delays happen every time a *packet_in* message is received a rule is inserted in the flow table of the switch. Similarly, the flow can be delayed if a cut-through tunnel is

being created. The goal of this problem is to minimize the total transfer time for all flows combined with the delays caused by the control plane.

Objective function: minimize

$$\sum_{s,d \in V} \delta_{s,d} + \sum_{x,y,i,j \in V} r_{s,d}^{i,j} \times \iota_{i,j} + f_{x,y}^{s,d} \times \tau_{x,y} \tag{6.1}$$

Subject to:

$$\forall i,j \in V, \sum_{x,y,s,d \in V} \lambda \times (\phi_{s,d}^{i,j} \times r_{s,d}^{i,j} + \psi_{x,y}^{i,j} \times f_{x,y}^{s,d}) \leq c_{i,j} \tag{6.2}$$

$$\forall i,j,s,d \in V, \phi_{s,d}^{i,j} \times r_{s,d}^{i,j} + \sum_{x,y \in V} \psi_{x,y}^{i,j} \times f_{x,y}^{s,d} = 1 \tag{6.3}$$

$$\forall x,y,s,d \in V, t_{x,y} \geq f_{x,y}^{s,d} \tag{6.4}$$

$$\forall x,y \in V, \sum t_{x,y} \leq T \tag{6.5}$$

$$\forall s,d \in V, D \times \sum_{x,y \in V} f_{x,y}^{s,d} \times \tau_{x,y} \leq \delta_{s,d} \tag{6.6}$$

$$\forall x,y \in V, \sum t_{x,y} (\sum_{i,j \in V} m_{i,j} \times \psi_{x,y}^{i,j}) \leq M \tag{6.7}$$

The minimization objective function of this problem (Equation 6.1) uses the τ and *th* parameters to take into account all the delays. For each direct link used, the *th* delay is considered. Similarly, for each tunnel, the τ delay is counted. By adding these delays to the duration of each flow $\delta_{s,d}$, the problem minimizes the total transfer time.

Constraint 6.2 ensures that the link capacity limit is enforced for all links in

the network. Constraint 6.3 ensures that all the links $\{i, j\}$ between a source s and a destination d are used, either as a single link ($\phi_{s,d}^{i,j} \times r_{s,d}^{i,j}$), or as part of a tunnel that uses that link ($\psi_{x,y}^{i,j} \times f_{x,y}^{s,d}$). By making the equation equal to one, we also guarantee that only zero or one bypass using the link can co-exist, thus eliminating incompatible tunnels.

Constraint 6.4 is used to ensure that $t_{x,y}$ is true if, for any pair of source s and destination d, the tunnel $f_{x,y}^{s,d}$ is used at least once. $t_{x,y}$ is then used to keep track of the maintenance cost of the system, instead of using f values that are specific to each flow and could thus be duplicated for the same pair of nodes.

Constraint 6.5 guarantees that the number of tunnels created stays below T, the maximum number of tunnels allowed.

Finally, constraint 6.6 ensures that a flow between source s and destination $d \in V$ only goes through tunnels if the time needed to setup the tunnels is D times smaller. The goal of this constraint is to avoid tunneling a flow that lasts 5 seconds if creating a tunnel will take 3 seconds. Instead, a longer flow will benefit more from the tunnel. Similarly, constraint 6.7 ensures that the maintenance cost of all tunnels is below the maximum threshold M.

6.3.4 Study of sample topologies

To validate the problem formulation, we ran the optimization for two different topologies created randomly. Topology 1 is a small topology with 11 nodes, two domains and 6 flow requests. Topology 2 has 35 nodes, two domains and 15 flow requests. In our experiments, all flow requests are inter-domain.

First, we note that the inter-domain latency plays a key role in how tunnels are created. We experimented using Topology 1 for different values of inter-domain controller latency across domains (w parameter in the optimization problem). The

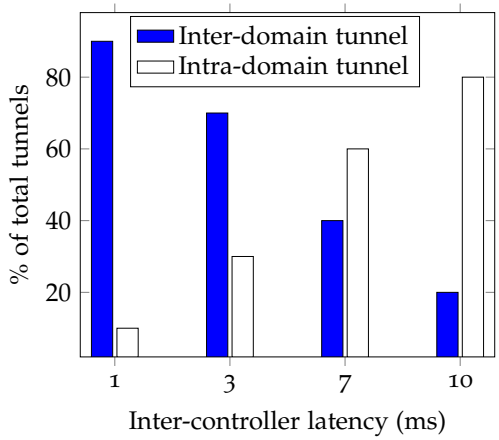

Figure 6.2: Distribution of intra-domain and inter-domain tunnels for varying inter-domain controller latency.

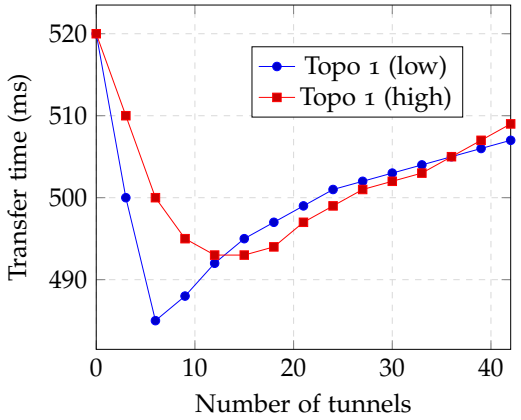

Figure 6.3: Total delay caused by controller processing with low and high inter-controller latency for Topology 1.

results in Fig. 6.2 show how the number of inter-domain tunnels created decreases as the latency increases. Indeed, the benefit of bypassing multiple hops (i.e. reducing the impact of switch to controller latency at each hop) is only beneficial as long as the time needed to create a tunnel is reasonable. As a result, for a small inter-domain latency, the optimization problem tends to create a single tunnel for each flow. However, when the value increases, the solver tends to create more edge-to-edge intra-domain tunnels for each domain.

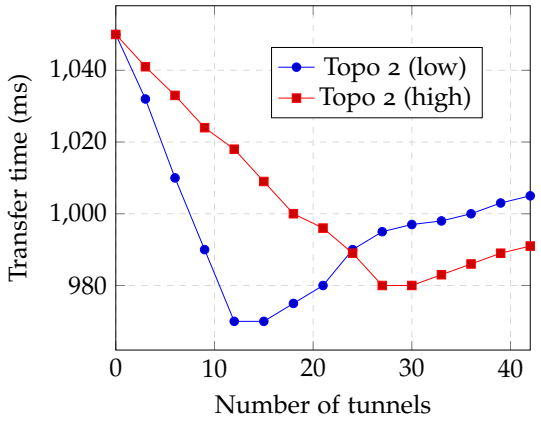

Figure 6.4: Total delay caused by controller processing with low and high inter-controller latency for Topology 2.

Such a behavior is confirmed in Fig. 6.3 and Fig. 6.4. We made a modification to Constraint 6.5 to enforce an exact number of tunnels instead of just setting an upper bound T. As a result, we observe the minimum delays obtained for topologies 1 and 2, when the inter-domain controller latency is 3 or 10 (low and high in Fig. 6.2). Notice how, for a low latency, the solver creates approximately one tunnel per flow. In contrast, for a higher latency, on average two tunnels per flow are created.

Finally, these experiments also show that the maximum delay is obtained when there are no tunnels. Likewise, once the minimum has been obtained, the increase in the delay is slow in comparison to how it decreases before reaching the optimal value. Therefore, we can conclude that the best way to set up the tunnels depends on the inter-domain controller latency and also having too many tunnels is better than not having enough tunnels.

6.3.5 Heuristic

The proposed optimization problem has scalability problems when there are more than one thousand 15-hop flows. Hence, it is not feasible to simply run the optimization problem at the domain controller. Therefore, we propose a heuristic that finds near-optimal solutions to the same problem but scales better.

The key parameter of this heuristic is the inter-domain latency. Indeed, we showed in Figs. 6.2, 6.3 and 6.4 that inter-domain, end-to-end tunnels are preferred when the latency is small and multiple intra-domain tunnels are preferred otherwise. For this reason, Alg. 3 compares the inter-domain latency with the controller-switch latency and decides to create an inter-domain tunnel only when this ratio is below 30%. Notice also that we consider multiple domain controllers so that each controller handles the flows that start within its domain.

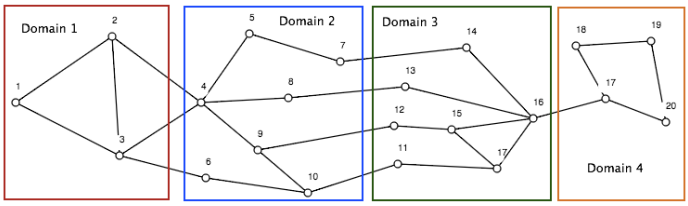

Figure 6.5: Sample topology used to evaluate the heuristic.

Table 3 compares the performance of the proposed heuristic with the optimal values found by the ILP solver. For this experiment, we created a larger topology shown in Fig. 6.5. The results show how the heuristic is capable of finding good solutions even beyond 800 flows, when the ILP solver starts having scalability problems.

In a real network, several implementation challenges must be addressed to solve the problem of inter-domain cut-through switching. For example, we assumed a known data rate and flow duration, as well as inter-domain link visibility.

```
Data: flows: Collection of flows
Data: controllers: All domain controllers
Result: Output how to route each flow
for each controller c ∈ controllers do
    boolean latencyOk = true;
    for each flow f ∈ flows do
        if flow.source ∈ domain c then
            for each controller ic ∈ flow.intermediateControllers do
                if latencyBetween(c, ic) > THRESHOLD then
                    latencyOk = false;
                end
            end
            if ¬ flow.tunnel then
                if latencyOK then
                    Create end-to-end inter-domain tunnel;
                    flow.tunnel = true; //Mark flow as tunneled so that other
                    controllers know
                end
                else
                    Create intra-domain tunnel;
                end
            end
        end
    end
end
```

Algorithm 3: Greedy heuristic to assign segments.

Table 6.1: Comparison of heuristic against ILP solver

Number of flows	ILP - transfer time (s)	Heuristic - transfer time (s)	ILP - running time (s)	Heuristic - running time (s)
100	6.9	7.1	0.1	0.05
300	22.3	25.1	0.5	0.07
500	34.1	39.2	4.1	0.1
800	54.6	61.3	21.3	0.11
1000	*	81.7	Out-of-memory error	0.12
1200	*	9963	Out-of-memory error	0.13

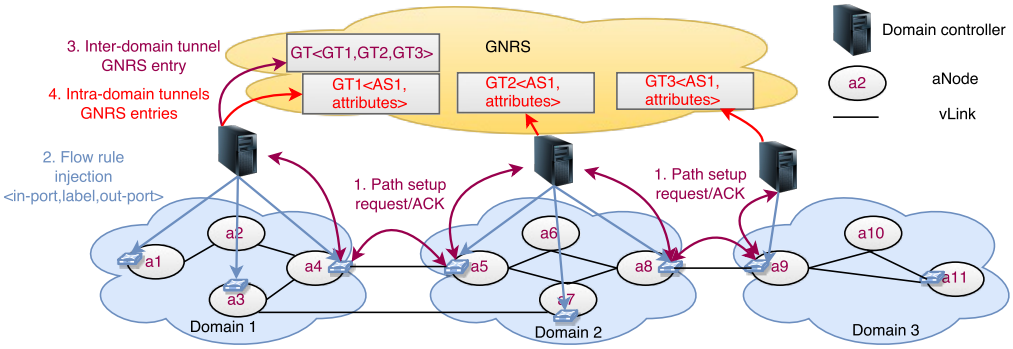

Figure 6.6: Steps needed to setup an inter-domain tunnel across multiple ASes

After demonstrating the importance of inter-domain controller latency in this section, next we propose a routing framework capable of inter-domain cut-through switching that simplifies how the tunnels are created and maintained. After that, in Section 6.5 we will explain how to incorporate the proposed heuristic into the routing framework.

6.4 Routing framework description

In this section, we first motivate using SDN for the proposed routing framework. Next, we explain how the framework meets three requirements to implement dynamic creation of inter-domain tunnels. First, domain controllers need to be aware of at least part of the topology from other domains (we assumed in the formulation problem that each domain controller was aware of the bandwidth of all links in the network). Second, the domain controllers must be capable of exchanging messages with other domains to setup cut-through tunnels across domains. The problem of deciding when to create a tunnel and which maps to forward through each tunnel is addressed in Section 6.5.

6.4.1 The need for SDN

An SDN-enabled routing framework provides several benefits to the proposed inter-domain protocol. First, SDN simplifies how to process in-transit traffic within a single domain. Indeed, managing intra-domain paths from a centralized controller is simpler than doing it in a distributed manner. For example, finding shortest paths in a fully known graph is simpler than exchanging topology messages between routers.

Second, managing the label-based intra-domain table is equally simplified. In fact, having distributed routers agreeing on multiple labels is possible, but it is more complex and requires an additional label distribution protocol. Also, the information at each router is not necessarily complete or up-to-date. Instead, assigning labeled-paths in a fully known graph becomes a much simpler problem. Moreover, since the controller is aware of all labels being used across the domain, it can verify that label-based paths do not collide.

Third, implementing routing policies within a domain is easily implemented when the controller has full control of routing decisions. Instead of propagating information across domains, the policies are given to the OpenFlow controller and they can easily be implemented since the controller is responsible for all routing decisions. Besides this, the controller can also provide the domain operator with a human-friendly API where different types of policies can be defined.

Fourth, SDN provides a unique opportunity to perform traffic analysis at the edge of the network. When the border switch must send data to another domain, an application running on the controller can use traffic statistics to detect elephant flows, mice flows or traffic with specific requests by the sender. As a consequence, SDN allows for fine-grained, per flow analysis that allows the controller to make

routing decisions for each flow.

Fifth, SDN also allows the integration of the routing mechanisms with lower layers such as Optical Transport Networks (OTN) or Wavelength Division Multiplexing (WDM). An SDN-based, integrated control plane can do multi-layer provisioning based on the behavior of a flow. In particular, flows with large bandwidth demand can benefit from being sent across multiple domains through an optical layer tunnel.

Finally, mobility awareness becomes crucial in a routing framework for MobilityFirst. SDN provides a way to create routing paths that react to mobility. Using an OpenFlow controller, we can query the GNRS, realize that the destination has moved to a different network address and re-define how the flow should be routed. The ability to create mobility-aware, multi-domain tunnels is significantly enhanced by using an SDN-based routing framework.

6.4.2 Increased visibility between domains

In MobilityFirst, autonomous systems (ASes) have the flexibility to expose their internal network characteristics in terms of aggregated nodes (aNodes) and virtual links (vLinks) (see Fig. 6.7). Each AS has the flexibility to decide on the aggregation granularity and hence the amount of state it wants to advertise. State is announced and exchanged in the form of a network state packet (nSP) similar to link state routing (see Fig. 6.8). Each domain controller is responsible of creating the virtual topology of aNodes and vLinks and it is also responsible of propagating link information such as bandwidth, availability, variability and latency. The advantage of sharing information of aNodes and vLinks through nSPs is two fold. On the one hand, it allows each domain to customize the topology information to be shared with other domains. On the other hand, it provides useful information to

other domains that can now decide how to route packets to get a given bandwidth, availability, variability and latency.

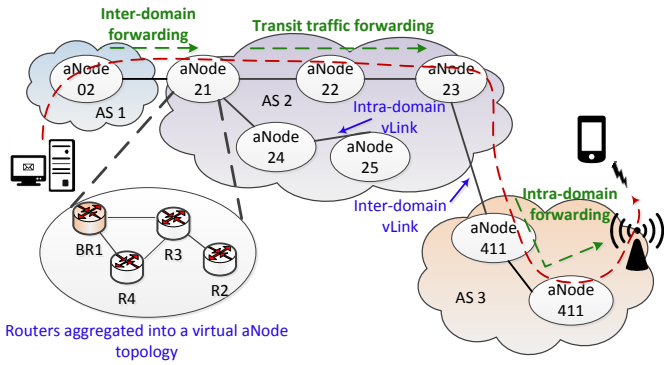

Figure 6.7: aNode-vLink topology abstraction for an AS.

Msg_Type	AS_Num:Source_aNode	Hop_to_Src
Internal Topologies:		
aNode#1-vLink<B,V,A,L>-aNode#2 aNode#2-vLink<B,V,A,L>-aNode#3 ... aNode#x-vLink<B,V,A,L>-aNode#y		
Neighbor Info:		
Neighbor_aNode#1-vLink<B,V,A,L> Neighbor_aNode#2-vLink<B,V,A,L> ... Neighbor_aNode#z-vLink<B,V,A,L>		

Figure 6.8: Structure of network state packets propagated across domains.

6.4.3 Dynamic creation of inter-domain tunnels using the GNRS

Next we describe a novel technique to setup cut-through switching tunnels across multiple domains that leverages the globally available name resolution service (GNRS). The main advantage of having a globally available entity is that the number of messages needed to be exchanged between domain controllers is significantly reduced.

To take full advantage of the GNRS, the routing framework names the tunnels. In other words, every tunnel created in a MobilityFirst network is an object that can be identified with a GUID (see Fig. 6.6). When a domain controller initiates a request for a inter-domain tunnel, it contacts other domain controllers with a setup request. The request includes a label to identify traffic, as well as the GNRS entry containing information of the tunnel. When a neighbor domain controller accepts the request to create a tunnel, it creates a GNRS entry that contains information about the tunnel to be shared with other domain controllers. As a result, once the tunnel has been created, the domain controller that initiated the request knows the GUID of all entries needed to collect information about the tunnel.

Although some initial messages are needed to create the tunnel, one advantage of this technique is that tunnel maintenance and tear-down do not need further messaging. First, a domain controller can use the GNRS entry to share tunnel attributes with other domains, such as available bandwidth or expected time before having to terminate the tunnel. Second, terminating the tunnel is as simple as deleting the GNRS entry. Fig. 6.6, suppose AS3 deletes the GNRS tunnel (GC) entry GT3. The initiating domain controller (AS1) notices that the entry has been deleted and concludes that AS3 is no longer part of the tunnel. Next, it deletes the GT entry known to all domains members of the tunnel. Finally, AS2 notices that the GT entry has been deleted. We evaluate the reduction of inter-domain messages in section 6.6.4.

In this section we described how the framework implements inter-domain cut-through tunnels (the *how*). In the following one, we explain how to decide when to create a tunnel and which flows to forward through each tunnel (the *when)*.

6.5 Traffic engineering techniques used by the framework

In this section we first describe how a domain controller can identify flows that could benefit from a cut-through tunnel. Second, we describe what is the most appropriate way to setup a tunnel once a flow has been identified as candidate.

6.5.1 Deciding which flows to forward through a tunnel

The first step consists of analyzing traffic to identify flows that could benefit from a cut-through tunnel. The first technique detects elephant flows at the edge of the network. The second takes advantage of the GNRS capabilities and deduces the mobility of the destination device. The third uses the service type field in a MobilityFirst packet to allow the sender, or a previous controller, to indicate that a tunnel should be used.

6.5.1.1 Controller-initiated cut-through based on flow rate and duration

The first technique that we implement and evaluate is a reactive approach. The decision of whether to send a flow through a tunnel is taken by the domain controller based on the observed traffic. The goal is to detect large, long duration flows (also known as elephant flows) that can benefit from cutting through the MobilityFirst routing mechanisms. The motivation for this technique is that elephant flows are long enough to benefit from a cut-through tunnel even when a small overhead is incurred to create the cut-through. Also, elephant flows usually have a high data rate and reducing the overhead needed to forward data increases the throughput significantly. Mouse flows (short flows), on the other hand, do not necessarily benefit from this because the transmission of data could be finished by the time the bypass is created, or not long enough to actually benefit.

Previous studies have addressed elephant flow detection using packet sampling techniques [239, 240, 241]. Psounis et al. [241] propose SIFT, a detection algorithm based on biased sampling. The intuition behind the algorithm is to sample with a low probability. Using this method, when a probability of $p = 0.01$ is used, all flows with more than 100 packets are detected and the probability of sampling a mouse flow is low. We implement the SIFT algorithm to detect elephant flows.

To detect large flows, traffic statistics must be observed. Although the OpenFlow protocol provides ways to query each switch for traffic statistics, this solution is too slow (data can only be obtained every five seconds). We use sFlow [211] to counter this problem and sample traffic every second. sFlow is a packet sampling tool used to monitor network usage. sFlow agents are deployed in network devices such as switches or routers and the information gathered is sent to a centralized collector. sFlow-RT is a monitoring framework that incorporates the sFlow analytic engine to provide real-time visibility in Software Defined Networking (SDN) [242]. A script running on the domain controller server obtains the metrics collected by sFlow-RT using the REST API. Using this data, the biased sampling technique proposed by the SIFT algorithm is used to detect large flows. When a flow has been labeled as elephant, the script uses the REST API of the Floodlight controller to find a route between the source and the destination and create a layer-2 circuit.

6.5.1.2 Controller-initiated cut-through based on mobility

The second technique is also reactive and leverages a specific feature of MobilityFirst: the GNRS server. In MobilityFirst, every device connected to the network should register to the GNRS to notify the current attachment point. This notification is achieved through an update message that contains the GUID and the new set of NAs to which the device is attached. Therefore, the GNRS offers a unique

opportunity to track the mobility of all devices attached to a set of NAs.

To implement the controller-initiated cut-through based on mobility, we first expand the GNRS server so that it also keeps track of the number of updates issued by each device. With our implementation, when the GNRS receives a query for a given GUID, the response contains a timestamp of all update messages received by the GNRS for the given GUID.

Next, we leverage the SDN controller to dynamically process the timestamps and determine the degree of mobility of the destination device. To measure the degree of mobility, the controller takes into consideration the number of updates sent by the device in the last hour. If the device has remained attached to the same access point for more than an hour, it is considered static and the controller can initiate a cut-through tunnel for this flow. If the number of updates is larger than zero, then the node is considered mobile and a tunnel is not created for the flow.

6.5.1.3 Sender-initiated cut-through

The third technique is a proactive approach that can be used by the sender of data to ask for a cut-through tunnel to be used. Unlike the previous two, this approach is proactive because the sender device requests the network to use a cut-through tunnel. Thus, the controller does not react to traffic observations or GNRS responses but to a request introduced by the sender in the network packets.

This method makes use of the service type field of MobilityFirst packets. This field encodes the requested processing or delivery service(s). In MobilityFirst, it was designed to allow the user to request different delivery methods, such as unicast or multicast. By setting the service type to a specific value, a sender asks the network to forward all packets in that flow through a tunnel.

Since this decision is based on a notification from the sender, the controller

does not actually decide if a bypass should be used or not, it only creates it assuming that the sender knows why a bypass can be beneficial. Note that this is an advantage but also a risk. Certainly, this technique is the easiest to achieve by the controller (no traffic observation or timestamps parsing needed). On the other hand, the sender assumes all the responsibility of sending data through a bypass and this could be detrimental if the destination is highly mobile.

6.5.2 Deciding how to setup a tunnel

Once the controller has identified a flow that could benefit from a tunnel using the described techniques, the next step is to decide what is the best way to setup such tunnel. To do this, we use the insights learned in Section 6.3 that showed how the inter-domain latency plays a key role on the optimal creation of tunnels. Therefore, we incorporate into the framework the ability to create inter-domain tunnels only if the inter-controller latency is low, and to use intra-domain tunnels otherwise.

6.5.2.1 Combined technique

To consolidate the three techniques and the heuristic, we propose a method that combines them all. The main goal of this combined technique is to admit a flow into a tunnel when at least two of three 'conditions' exist: long flows, low mobility or user-requested tunnel. Each flow has candidate points to be accepted in a bypass tunnel. An elephant flow earns three points and a tunnel request from the user also earns three points. Finally, a static destination adds four candidate points to a flow. To ensure low mobility and another condition, this technique works by creating tunnels only to flows that have at least seven candidate points.

Another feature of the combined technique is that it allows to upgrade or downgrade flows as candidates for a tunnel. Suppose a flow has been sending

high traffic rates for a long time and the destination was static. Therefore, this flow had seven points and was part of a bypass. Later, if the destination stops being static or if the data rate is reduced, this flow can be downgraded and removed from the bypass. Similarly, it can be added again if the conditions change again.

Finally, the combined technique also considers the inter-controller latency to decide if a single end-to-end tunnel should be created, or if an intra-domain tunnel should be used instead.

Algorithms 4, 5 and 6 show how the combined traffic engineering technique operates. First, incoming flows are analyzed to identify the mobility degree and service type (Algorithm 4). Next, traffic is periodically sampled to identify elephant flows and update mobility degrees (Algorithm 5). Finally, Algorithm 6 is responsible for creating the tunnels or mapping flows to existing tunnels.

```
Data: new flow f
//New flow, check mobility and service type
serviceType = getServiceType();
mobilityDegree = getMobilityDegree();
if serviceType = bypass then
    f.candidate += 3;
end
if mobility = low then
    f.candidate += 4;
end
```

Algorithm 4: Combined technique applied to incoming flows.

6.6 Experimental evaluation

We first demonstrate that the framework is capable of detecting elephant flows and mobile destinations using the combined technique shown in Section 6.5. After that, we focus on demonstrating how inter-domain tunnels reduce the number of packets that must be handled by domain controllers. We also compare the number

```
Data: flows: collection of all existing flows
//Monitor rate and mobility of existing flows
for each flow f in flows do
    //Update elephant flow status
    elephant = isElephantFlow(f);
    if f.elephant = false and elephant = true) then
        f.elephant = true; //flow becomes elephant
        f.candidate += 3;
    end
    if f.elephant = true and elephant = false) then
        f.elephant = false; //flow stops being elephant
        f.candidate -= 3;
    end
    //Update mobility status
    mobility = getDestinationMobility(f);
    if f.mobility = high and mobility = low) then
        f.mobility = low; //destination stopped moving
        f.candidate += 3;
    end
    if f.mobility = low and mobility = high) then
        f.mobility = high; //destination started moving
        f.candidate -= 3;
    end
    if f.inBypass = true and f.candidate ¡ 7) then
        removeFromBypass(f); f.inBypass = true;
    end
    if f.inBypass = false and f.candidate ¿ 7) then
        addToBypass(f); f.inBypass = true;
    end
end
```

Algorithm 5: Combined technique applied to existing flows.

```
addToBypass (Flow f)
src = f.source;
dst = f.dst;
if tunnel exists between src and dst) then
    Add tunnel VLAN tag to f;
end
else
    for each controller ic ∈ flow.intermediateControllers do
        if latencyBetween(c, ic) > THRESHOLD then
            latencyOk = false;
        end
    end
    if latencyOK then
        Create end-to-end inter-domain tunnel;
    end
    else
        Create intra-domain tunnel;
    end
    Add tunnel VLAN tag to f;
end
serviceType = getServiceType();
mobilityDegree = getMobilityDegree();
if serviceType = bypass then
    f.candidate += 3;
end
if mobility = low then
    f.candidate += 4;
end
```

Algorithm 6: Combined technique applied to incoming flows.

of messages needed by the framework to create tunnels against a known protocol such as label distribution protocol (LDP) [243]. To do so, we provide results of the implementation on the GENI testbed [122] using the parameters shown in Table 6.2 and the topology shown in Fig. 6.12. Finally, we evaluate the scalability of the framework by emulating a large topology in Mininet.

To generate MobilityFirst traffic, we used the same host stack implementation as in Chapter 5 to use the ping and file transferring tools. For the Floodlight

controller, we worked on top of the implementation described in Chapter 5 and we added two components. The first one is responsible for creating aNodes and vLinks for each domain and exchange nSPs between controllers. The second one is responsible for implementing the traffic engineering techniques described in Section 6.5.

6.6.1 Elephant flow detection

First, we show how the techniques described can be used to create tunnels within a domain, taking traffic behavior and mobility into consideration. The experimental topology (fig. 6.9) was deployed in the GENI testbed. Virtual machines are used for the GNRS, controller, senders and receiver. OpenFlow switches are implemented using Open vSwitch instances. In this topology, nodes A, B and C are connected to the network through intra-domain switch *intra 1*. To transmit inter-domain packets, switch *intra 1* forwards packets to its default border router, *border 1*. In this example, *border 1* routes packets to *border 2*.

In these experiments, we test how the controller chooses between path 1 (going through intra-domain switch *intra 2* or through *intra 3*. The decision of which path to use is based on the following policies. First, tag flows as elephant if they maintain a traffic rate of 100,000bps during at least 10 seconds. Second, use path 2 (through *intra 3*) for large flows. Third, do not use path 2 if the destination is highly mobile, regardless of the traffic rate of the flow.

In the first experiment (fig. 6.10), senders A, B and C do the following steps. At time 5, sender A starts a flow at 608bps. At time 11, sender B starts a flow at 1344bps. At time 18, sender C starts a flow at approx 700,000bps. At time 28, the controller tags flow C as elephant and sends it through path B. Sender C sends large amounts of data during 10 secs. At time 37, sender C reduces the rate to

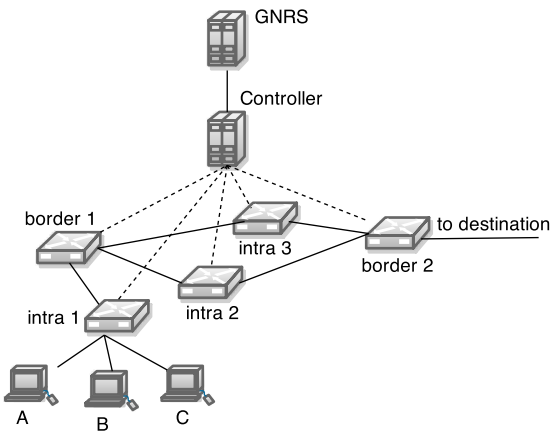

Figure 6.9: Experimental topology. Sender nodes are connected to switch intra 1 and the destination device is in another domain.

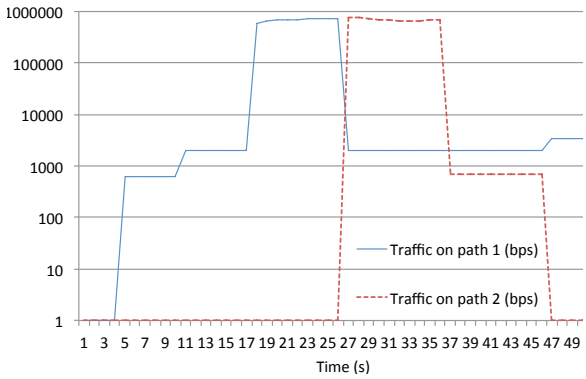

Figure 6.10: Detection of a large flow. When sender C starts a large flow, the controller tags it at elephant. Next it removes the tag when the load is reduced.

688bps.

The results show how flows from senders A and B stay on path 1 as expected. When the large flow by sender C starts, it stays on path 1 for 10 seconds, as defined by the policy. After that, the controller tags it as elephant flow and moves it to path 2. The red, dashed line shows how traffic from sender C is going through path 2 but the first two flows stay on path 1. When sender C lowers the rate at time 37, the controller waits for 10 seconds and then removes the elephant flow tag and traffic is moved to path 1.

6.6.2 Mobility-aware routing

In the previous experiment, we did not take the mobility of the destination into consideration. In the following one, we show how our framework is capable of mobility-aware tunneling.

The experimental setup is as follows. We start with an elephant flow that has already been detected and has been routed through path 2, a fast path for large flows. Meanwhile, another mouse flow is forwarded through path 1. The first 50 seconds of Fig. 6.11 show the two flows on paths 1 and 2. After 45 seconds, we simulate the mobility of the destination device by sending a message to the GNRS indicating that the GUID is now attached to a different access router. As a consequence, the controller detects that the destination is mobile and downgrades the flow using the combined technique (see Alg. 5). Figure 6.11 shows how traffic on path 2 stops and the large flow is now routed through path 1.

There are several parameters in this technique that can be customized by the domain operator. First, the controller needs to query the GNRS to know the mobility of the destination devices of existing flows. The frequency of such GNRS lookups can be modified. Second, the threshold to decide that a destination shows enough mobility to be removed from a path can also be changed. In this experiment, we removed the flow from a path with a single GNRS update detected. However, for some scenarios this value could be larger.

Table 6.2: Summary of components and key parameters used in the experiments

Type of switch	Open vSwitch version 1.9.3
Controller version	Floodlight 1.0
Controller host	Ubuntu 12.04 LTS
Controller host processor	Intel(R) Xeon(R), 2.67GHz
End-user OS	Ubuntu 12.04 LTS
Link bandwidth	100 Mbps

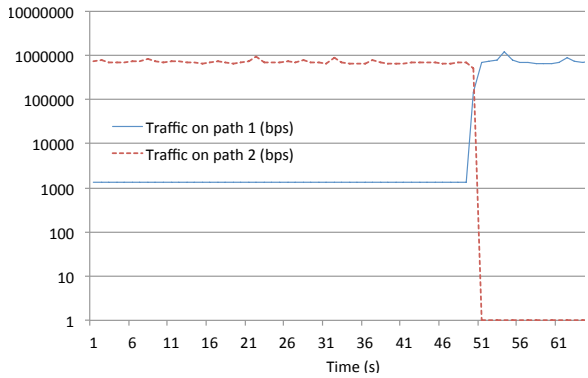

Figure 6.11: Downgrading a large flow due to destination mobility. When the destination becomes mobile, the flow is switched to a different path.

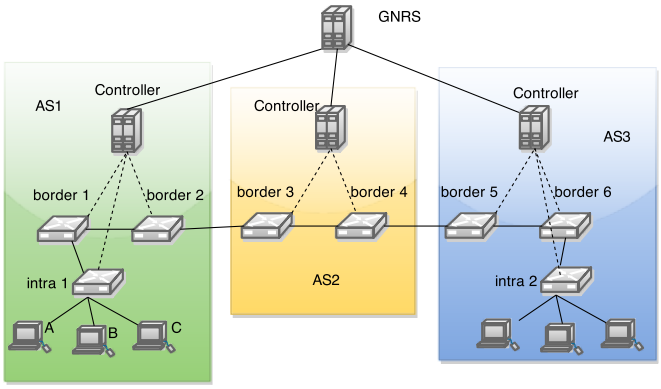

Figure 6.12: Experimental topology. Three SDN-based domains are deployed with end-nodes on ASes 1 and 3 and traffic going through an in-transit domain (AS2).

6.6.3 Inter-domain tunneling and flow aggregation

Next we show how multiple domains agreeing on an inter-domain label-based tunnel reduces the number of packet_in messages received by transit controllers. Table 6.3 shows in average how many packets must be forwarded to the controller by the switch when such packet does not match any rule in the flow tables.

The experimental setup is as follows. We create 25 flows originating in AS1 with different destinations in AS3. Since the destination is different for each flow, then the controller must process the first packets of all flows. Out of these 25 flows, six send data at rates above 1000000 bps, six at rates above 100000 bps, seven at

Table 6.3: Number of packet_in messages received by the controller based on the traffic rate.

Traffic rate	Packet_in messages per flow
1000000	10
100000	7
10000	3
1000	2

rates above 10000 bps and six transmit data at lower rates. Out of these 25 flows, 18 are sent from AS1 to AS3 through a previously setup tunnel. The remaining seven flows are forwarded un-labeled between each domain.

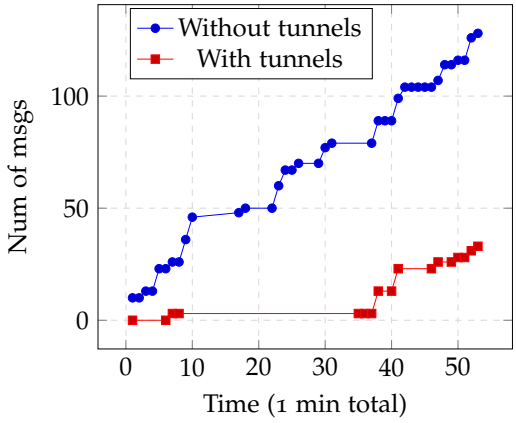

Figure 6.13: Accumulated number of *packet_in* messages received by the AS2 domain controlle with and without inter-domain tunnels.

Figure 6.13 shows the number of *packet_in* messages received per second by the controller. When all traffic is forwarded without using an inter-domain cut-through tunnel, the controller receives a total of 128 messages (top curve). However, when inter-domain tunnels are created for some of the flows, the total number of messages is reduced to 33 (bottom curve), for a 75% reduction. These results are specific to this topology and flow demands, but our goal is just to demonstrate how the creation of inter-domain tunnels can reduce the control plane delay. In

total, 17 flows are being aggregated.

6.6.4 Reduction of label distribution messages

Next we demonstrate how combining SDN with tunnel naming reduces the number of messages needed to create and maintain inter-domain tunnels. To do this, we briefly describe how all the functionality of LDP used in MPLS is implemented by our framework and we compare the number of messages needed to setup inter-domain tunnels.

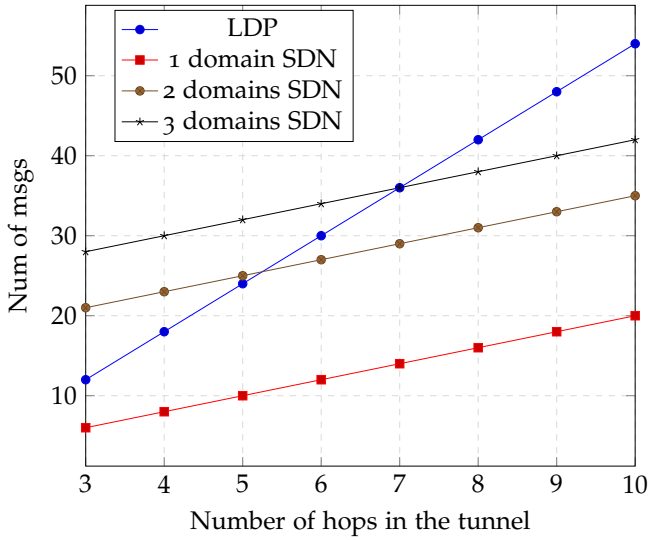

Figure 6.14: Number of messages needed to setup inter-domain tunnels using LDP or our framework.

Some benefits are due to using SDN. First, note that by using SDN, the number of intra-domain messages between peers is unnecessary. Instead, the SDN controller is responsible for pushing forwarding rules to the switches. Therefore, there is no need for intra-domain discovery messages. Second, session messages exist between domain controllers as opposed to peering routers. This reduces the number of messages needed because the only links carrying these messages are

those between edge routers of neighbor domains. Third, advertisement messages are reduced for two reasons. On the one hand, intra-domain advertisement is not necessary because the domain controller is network aware. On the other hand, inter-domain advertisement is already achieved using the network state packets described in Section 6.4.2. For this reason, advertising messages are only needed to request a new tunnel creation, as described in Section 6.4.3. Table 6.4 summarizes the key differences between messaging in LDP and the proposed framework.

Table 6.4: Message equivalency between LDP and SDN-GNRS

Message type	LDP	SDN-GNRS
Discovery	Peer-to-peer between routers	No additional messages required, since this is achieved using network state packets
Session	Peer-to-peer between routers	A session between domain controllers is required regardless of tunnels. No additional messages required.
Advertisement	Peer-to-peer between routers	Controller-to-controller and controller-to-GNRS messages are required
Notification	Peer-to-peer between routers	Controller-to-controller messages are required
Flow rule injection	Not required	Controller-to-switch messages are needed to push forwarding rules to the switches. Figure 6.14 does consider these messages in the comparison.

Other benefits are due to naming the tunnels as network objects. First, the GNRS provides a common platform to exchange information between domain controllers. In MobilityFirst, the GNRS plays a key role in how packets are routed,

so we can assume that it is a highly available entity and a session between each controller and the GNRS will exist. This reduces the complexity of establishing sessions between multiple domains. Second, relying on the GNRS reduces the number of inter-domain messages needed when more than two domains are involved. Suppose in Fig. 6.12 that AS3 needs to communicate with AS1. There is no need for AS2 to be involved in the communication and the GNRS provides a direct way for AS1 and AS3 to exchange information.

These benefits can be appreciated in Fig. 6.14. First, notice how in LDP the number of messages grows independent of the number of domains traversed by the tunnel, as it requires three pairs of messages between peering routers in all cases. In contrast, when using SDN and the GNRS, the major factor for increase in the number of messages is the number of domains traversed. For a single domain, there is no need for the GNRS and the plot only includes messages needed to insert forwarding rules in the forwarding tables of the switches. Next, as the number of domains increases, we need more controller-to-controller messages as well as controller-to-GNRS messages. However, notice how the total number of messages stays below that of LDP for tunnels with four or more hops.

6.6.5 Scalability

Finally, we focus on evaluating the scalability of the proposed framework. To do so, we emulate in Mininet the same topology (Fig. 6.5) used to evaluate the heuristic in Section 6.3. To run MobilityFirst traffic on Mininet, we collect traffic traces from our experiments on the GENI testbed and replay them from the Mininet hosts. Using the same input as in Section 6.3, we measure the average delay faced by the first packet of a flow given different traffic demands. In Fig. 6.5, we attach five hosts to every router in Domain 1 and we randomly select a source attached to

router 20 in Domain 4. As a consequence, in this experiment there are 15 hosts capable of sending packets at different rates and all flows require an inter-domain data transfer between domains 1 and 4.

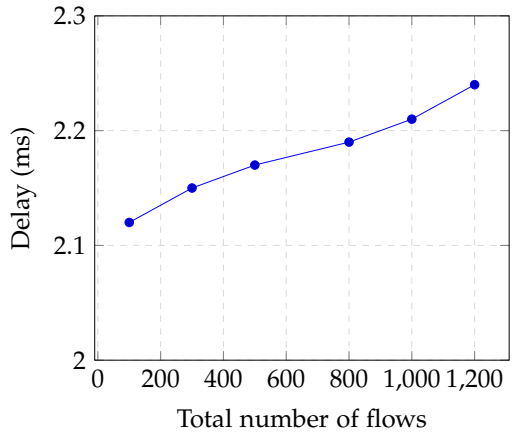

Figure 6.15: Total delay encountered by the first packet of each new flow.

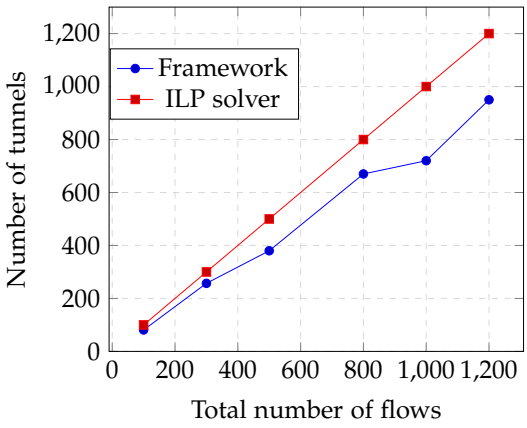

Figure 6.16: Total number of tunnels created by the framework and the ILP solver.

Fig. 6.15 shows the average delay faced by each packet given different traffic rates. This delay includes the switch-to-controller round-trip latency as well as the processing time at the controller. Given that the latency remains fairly constant, the low increase is due to the controller having to process more *packet_in* messages

at the same time. However, notice that this delays remain very small, since the running time of the traffic engineering algorithm is short.

Another interesting finding is that the optimization problem creates more tunnels than our routing framework, as shown in Fig. 6.16. The reason for this is because our traffic engineering techniques only pick some flows as candidates for tunnels and leave others out due to mobility or low bit rate. Although this increases the total delay in the network, some flows do benefit from hop-by-hop transmissions to improve content delivery.

6.7 Conclusion

In this chapter we proposed an SDN-based routing framework that enables inter-domain and intra-domain cut-through switching in the MobilityFirst architecture. Our framework tackles specific challenges of this FIA network, such as inherent mobility support and per-flow decision making based on flow behavior. Particularly, three aspects are novel in this framework. First, domain controllers advertise multiple paths to the same destination, with link information such as bandwidth, variability, availability and latency for each hop to ensure edge-awareness and efficient content delivery. Second, our framework provides mobility-aware, cut-through mechanisms using label-based forwarding at both intra-domain and inter-domain scale to ensure efficiency in the core network. Third, our framework implements granular, per-flow decision making to ensure appropriate routing based on flow behavior and mobility.

Our evaluation results show that the domain controller is capable of routing elephant flows through a faster path while remaining mobility-aware (flows with a highly mobile destination are routed through a different path to ensure efficient

delivery). We also show how a proactively created cut-through tunnel across three domains reduces the number of packets that need to be processed by the transit controller by 75%. Furthermore, we show how the proposed framework scales better than LDP when creating these tunnels. Finally, we demonstrate that the framework is scalable and capable of handling up to 1000 flows without adding any significant delay to incoming packets.

Our work also contributes to the deployment of SDN at Internet-scale by proposing a novel mechanism to exchange routing information across domains. Our work differs from other studies that have addressed inter-domain routing SDN in two aspects: first, our network is designed to inherently support mobility, a key requirement for the future Internet. Second, our architecture leverages a centralized name resolution system (GNRS in the specific case of MobilityFirst) that significantly reduces the complexity of the messages that need to be exchanged between network controllers to agree on inter-domain cut-through switching.

As future work, we will complete the implementation of the prototype and integrate it into the MobilityFirst deployment. We also plan to evaluate the proposed mechanisms on a larger scale using experimental scenarios which incorporate high-speed optical switching components.

Chapter 7

Conclusions and future directions

The frameworks proposed in this dissertation, as well as a plethora of related work, show how SDN can be used to simplify network management at all scales. By decoupling the control plane from the data plane, SDN-based networks provide a clear separation between network applications and the underlying topology and this benefit has been instrumental in making networks simpler and more manageable. SDN has renovated research in topics such as multi-layer integration, network abstraction and network function virtualization. SDN has also been widely adopted by both vendors and providers. The implementation of SDN is not standardized yet. However, the adoption of a paradigm where an operation system stands between the devices and the applications has been accepted as a new way of deploying both local and wide area networks.

When OpenFlow was proposed in 2008, it triggered a large amount of research and publications in all areas of networking *using SDN*. However, OpenFlow was considered as a *hype* by some and the challenge was to convince readers of the advantages of decoupling the control plane from the data plane. Clearly, networks can be operated without SDN. Therefore, the initial motivation of our work was to demonstrate how SDN can simplify network management and what are the advantages of using an SDN centralized controller instead of distributed data

plane protocols.

Given that OpenFlow was been proposed for LANs and data centers, in Chapter 3 we proposed OpenSec, an OpenFlow-based security framework that allows a network security operator to create and implement security policies written in human-readable language. Using OpenSec, the user can describe a flow in terms of OpenFlow matching fields, define which security services must be applied to that flow (deep packet inspection, intrusion detection, spam detection, etc.) and specify security levels that define how OpenSec reacts if malicious traffic is detected. We implemented OpenSec on the GENI testbed and demonstrated its functionality using two use cases applied to campus networks: housing network control and science demilitarized zone deployment. Our results demonstrate that up to 95% of attacks in an existing data set can be detected and 99% of malicious source nodes can be blocked. Likewise, we show that our policy specification language is simpler while offering fast translation times compared to existing solutions.

Next, we focused on demonstrating the advantages of using SDN in transport networks. In these networks, two challenges exist that are not as important in LANs: WAN virtualization and multi-layer provisioning. Network virtualization has become one of the driving applications of SDN. Providing a virtual software layer between devices and external applications is hard to achieve using distributed protocols only. Multi-layer provisioning using SDN, however, is a more challenging task. Indeed, OpenFlow was designed for standard Ethernet devices and flow-level management but optical devices are analog and proprietary.

To investigate if SDN could be used to make transport networks more flexible, in Chapter 4, we proposed XTEF to enable application-driven traffic engineering and provision transport network resources using on-demand WDM tunnels. XTEF proposes a solution to the two challenges mentioned above: network virtualization

and multi-layer provisioning. First, we used OneSwitch to abstract the entire WAN as a single virtual switch. By doing this, we proposed a solution to the problem of allowing external tenants to program paths across the WAN without revealing the network topology. Second, we proposed the DTS algorithm as a novel technique to increase network capacity using dynamic, short-term WDM tunnels through available wavelengths. As a result, we demonstrated how SDN can be used by network providers to manage a multi-tenant WAN. Our results show that 10% additional flows can be granted the requested bandwidth using the tunnels and this is possible without requiring the intervention of a network operator.

To further demonstrate how SDN enables flexible large scale networks, we investigated how MobilityFirst can benefit from it. This FIA proposes several techniques to ensure edge-awareness and efficient delivery to mobile devices. Our contribution to the project consisted on exploring how SDN can be used to efficiently forward flows that do not require mobility support using cut-through switching. To this end, we first proposed in Chapter 5 an SDN-based control plane for MobilityFirst. In this work, we introduced a general bypass capability within the MobilityFirst architecture that provides better performance and enables both individual and aggregate flow-level traffic control. Furthermore, we presented an OpenFlow-based proof-of-concept implementation of the bypass function using layer 2 VLAN tagging. We also ran experiments on the ORBIT and GENI testbeds to evaluate the performance and scalability of the solution. By implementing the bypass functionality, we were able to significantly reduce the number of messages processed by the controller as well as the number of flow rules that need to be pushed into the switches.

Given the benefits achieved at intra-domain scale using cut-through switching in MobilityFirst, in Chapter 6 we investigated how to implement this functionality

across multiple domains. First, we proposed and solved an optimization problem that minimizes the total transfer time using inter-domain tunnels. Second, we proposed an SDN-based routing framework for the MobilityFirst architecture capable of dynamically creating such tunnels. The main novelty of this framework is to name tunnels as network objects to simplify how tunnels are created and maintained. To validate our framework, we implemented on the GENI testbed a prototype for the MobilityFirst architecture. Our experiments with the optimization problem showed that the inter-domain latency between controllers plays a key role on how tunnels are setup. Furthermore, our implementation experiments showed that the control plane delay can be reduced by 75% when using inter-domain tunnels. Finally, we showed how our framework needs fewer messages than current protocols such as label distribution protocol (LDP) to setup intra-domain and inter-domain tunnels.

This dissertation addressed and proposed solutions to the following problems using SDN: policy-based management, network virtualization, multi-layer integration, WDM tunneling, intra-domain cut-through switching and inter-domain routing with cut-through switching. All these problems can be solved without SDN, but we have demonstrated how SDN can simplify the solutions.

While our work proposes solutions for different types of networks, it is reasonable to think of integrating our contributions as part of the same architecture. For example, OpenSec can be modified to enable policy-based security across domains when managing inter-domain cut-through switching. In this scenario, a domain administrator can implement policies that enable different levels of trust across domains. Using OpenSec, the operator can specify which domains are allowed to initiate cut-through switching requests that should be accepted by the controller. Likewise, a policy could be used to limit the resources available for inter-domain

tunnels, such as the number of tunnels or which routes can be used by these tunnels. Another example would be to integrate XTEF with cut-through switching. Indeed, we did not address the problem of inter-domain bandwidth provisioning and the usage of inter-domain cut-through switching is a natural next step in that direction.

7.1 Future directions

Each chapter of this dissertation provides opportunities for future work. At campus scale, we would like to pursue two directions. First, OpenSec can be extended to provide a framework for Cloud-based security, where flow processing is done in the Cloud instead of locally. Indeed, we foresee security becoming a routing problem where flows must be sent to the processing units, as proposed in OpenSec. The advantage of decoupling the control plane and the routing from the security mechanisms is that exporting the security devices to remote locations is simpler to deploy. As a consequence of this, we also plan to secure OpenSec from external attacks to prepare the framework for such a Cloud-based scenario. In a multi-tenant deployment with devices deployed in the Cloud, a model threat would include a compromised network application, a data plane attack and other scenarios that should be considered. Therefore, securing OpenSec from these attacks is a necessary step before deploying the framework.

In chapter III we described several challenges for scientific networks, such as WAN virtualization, scalability, multi-domain circuits, security and interoperability and multi-layer provisioning. In this dissertation we addressed WAN virtualization and multi-layer control. As future work, we first plan to investigate what is the most efficient way to achieve inter-domain circuits using SDN domain controllers.

While current solutions such as OSCARS already provide multi-domain circuit reservation, novel techniques must be designed so that circuits can be created without requiring as much trust among domains as OSCARS does. Secondly, we plan to investigate the security implications of having a multi-tenant WAN with automated application controllers programming the network. We are interested in designing mechanisms to identify compromised application controllers and ensure separation between tenants.

In chapters IV and V we described a routing framework with cut-through switching for MobilityFirst. While this framework provides efficient data transfers across domains, we are interested in better supporting mobility. In our current mathematical formulation and implementation, the cut-through tunnels created have static end-points. We are mainly interested in creating *mobile tunnels* that adjust the end-points to "follow" the mobile destination nodes. Furthermore, the problem formulation can be extended into many variations. In particular, we plan to investigate the steady-state scenario where traffic and tunnels exist when the optimization problem starts. This adds the additional challenge of having to tear down tunnels efficiently and measuring its effect on the network performance.

While SDN has been a widely accepted technology, the discussion on the appropriate southbound API is still open. OpenFlow 1.0 proved to be an excellent experimentation tool for researchers and for small scale deployments in campus networks and datacenters. However, larger scale deployments across WANs or domains have shown how OpenFlow 1.0 lacks of some necessary features such as path protection, recovery, management plane protocol or support for layers below L2. Newer versions of the OpenFlow protocol, such as OpenFlow 1.3 attempted to solve this by adding several features to make the protocol usable in production networks. The improvements included IPv6 support and Quality of Service (QoS),

for example. Also, the usage of multiple flow tables and buckets greatly increases what switches can do without contacting the controller. The downside of this innovation is that OpenFlow 1.3 is much harder to comply to in comparison to OpenFlow 1.0.

As a consequence, interoperability problems arise because deployments working on a specific hardware usually do not function well on different devices. As the innovation and implementation of the OpenFlow protocol continues, network vendors have looked at other options such as XMPP (Extensible Messaging and Presence Protocol), BGP-LS (Border Gateway Protocol - Link State) or PCEP (Path Computation Element Protocol). These protocols bring advantages such as providing a management plane (which is not provided by OpenFlow) and the ability to support legacy hardware during the transition to SDN. Similarly, different vendors provide different solutions for the control plane. Although OpenFlow was first proposed to standardize SDN, in reality vendors are still trying to push their own hardware combined with their own software.

Finally, the east-west interface between controllers is also at an emergent state only. We described in the related work of Chapter 6 how very little work has been done on inter-domain SDN. Although we proposed a framework for inter-domain routing, this problem is not completely solved yet and is likely to receive major attention.

Bibliography

[1] OpenFlow Switch Specification, Version 1.1.0 Implemented (Wire Protocol 0x02). [Online]. Available: http://www.openflow.org/documents/openflow-spec-v1.1.0.pdf

[2] United States Unified Community Anchor Network. United States Unified Community Anchor Network. [Online]. Available: http://www.usucan.org/about

[3] D. Raychaudhuri, K. Nagaraja, and A. Venkataramani, "MobilityFirst: A Robust and Trustworthy Mobility-centric Architecture for the Future Internet," *SIGMOBILE Mob. Comput. Commun. Rev.*, vol. 16, no. 3, pp. 2–13, December 2012.

[4] OpenFlow Experimentation in ORBIT. [Online]. Available: http://www.orbit-lab.org/wiki/Documentation/OpenFlow

[5] Cisco. Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 20142019. [Online]. Available: http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white_paper_c11-520862.html

[6] "Large Hadron Collider." [Online]. Available: http://home.web.cern.ch/topics/large-hadron-collider

[7] A. Lara, A. Kolasani, and B. Ramamurthy, "Network Innovation using OpenFlow: A Survey," *IEEE Communications Surveys and Tutorials*, vol. 16, no. 1, pp. 493–512, 2013.

[8] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, March 2008.

[9] A. Lara and B. Ramamurthy, "OpenSec: A Framework for Implementing Security Policies using OpenFlow," in *Proceedings of the IEEE Globecom Conference*, December 2014.

[10] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hlzle, S. Stuart, and A. Vahdat, "B4: Experience with a Globally-Deployed Software Defined WAN," in *Proceedings of the ACM SIGCOMM Conference*, 2013.

[11] A. Lara and B. Ramamurthy, "Dynamic network provisioning for SDN transport networks," in *Conference submission*, 2016.

[12] L. Zhang, A. Afanasyev, J. Burke, V. Jacobson, K. C. Claffy, P. Crowley, C. Papadopoulos, L. Wang, and B. Zhang, "Named data networking," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 66–73, July 2014.

[13] "MobilityFirst," http://mobilityfirst.winlab.rutgers.edu/.

[14] T. Anderson, K. Birman, R. Broberg, M. Caesar, D. Comer, C. Cotton, M. Freedman, A. Haeberlen, Z. Ives, A. Krishnamurthy, W. Lehr, B. Loo, D. Mazires, A. Nicolosi, J. Smith, I. Stoica, R. van Renesse, M. Walfish,

H. Weatherspoon, and C. Yoo, "The NEBULA Future Internet Architecture," in *The Future Internet*, ser. Lecture Notes in Computer Science, A. Galis and A. Gavras, Eds. Springer Berlin Heidelberg, 2013, vol. 7858, pp. 16–26.

[15] ChoiceNet: An Economy Plane for the Internet. [Online]. Available: http://www.ecs.umass.edu/ece/wolf/ChoiceNet/

[16] A. Lara, B. Ramamurthy, E. Pouyoul, and I. Monga, "WAN Virtualization and Dynamic End-to-End Bandwidth Provisioning Using SDN," in *Optical Fiber Conference (OFC)*, 2015.

[17] A. Lara and B. Ramamurthy, "Design Challenges in Using Software-Defined Networking for Science Networks," in *SDN for Scientific Networking Workshop at Super Computing*, November 2015.

[18] OpenFlow Current Deployments. [Online]. Available: http://www.openflow.org/wp/current-deployments/

[19] Open Networking Foundation. [Online]. Available: https://www.opennetworking.org/

[20] A. Doria, J. H. Salim, R. Haas, H. Khosravi, W. Wang, L. Dong, R. Gopal, and J. Halpern. Forwarding and Control Element Separation (ForCES) Protocol Specification. [Online]. Available: http://tools.ietf.org/html/rfc5810

[21] J. Zander and R. Forchheimer, "The SOFTNET project: a retrospect," in *8th European Conference on Electrotechnics*, June 1988, pp. 343 –345.

[22] D. L. Tennenhouse and D. Wetherall, "Towards an active network architecture," *SIGCOMM Comput. Commun. Rev.*, vol. 26, no. 2, pp. 5–17, April 1996.

[23] J. M. Smith and S. M. Nettles, "Active networking: one view of the past, present, and future," *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, vol. 34, no. 1, pp. 4 –18, February 2004.

[24] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden, "A survey of active network research," *IEEE Communications Magazine*, vol. 35, no. 1, pp. 80 –86, January 1997.

[25] J. W. Lee, R. Francescangeli, J. Janak, S. Srinivasan, S. A. Baset, H. Schulzrinne, Z. Despotovic, and W. Kellerer, "NetServ: Active Networking 2.0," in *2011 IEEE International Conference on Communications Workshops (ICC)*, June 2011, pp. 1 –6.

[26] T. V. Lakshman, T. Nandagopal, R. Ramjee, K. Sabnani, and T. Woo, "The SoftRouter Architecture," in *Proceedings of the ACM SIGCOMM Workshop on Hot Topics in Networking*, 2004.

[27] E. Keller and J. Rexford, "The "Platform as a service" model for networking," in *Proceedings of the 2010 Internet Network Management Conference on Research on Enterprise Networking*, 2010.

[28] A. A. Lazar. Aurel A. Lazar home page. [Online]. Available: http://www.ee.columbia.edu/~aurel/networking.html

[29] J. Biswas, A. A. Lazar, J. F. Huard, K. S. Lim, S. Mahjoub, L. F. Pau, M. Suzuki, S. Torstensson, W. Wang, and S. Weinstein, "The IEEE P1520 Standards Initiative for Programmable Network Interfaces," in *IEEE Communications Magazine*, vol. 36, no. 10, 1998, pp. 64–70.

[30] Internet Engineering Task Force (IETF). Proposal: Software Defined Networking Research Group (SDNRG). [Online]. Available: http://trac.tools.ietf.org/group/irtf/trac/wiki/sdnrg

[31] Internet Research Task Force (IRTF). Software Defined Networking Research Group (SDNRG) - Charter. [Online]. Available: http://www.1-4-5.net/~dmm/sdnrg/sdnrg.html

[32] Internet Engineering Task Force (IETF). Analysis of Comparisons between OpenFlow and ForCES. [Online]. Available: http://tools.ietf.org/html/draft-wang-forces-compare-openflow-forces-01

[33] OpenFlow Switch Specification, Version 1.0.0 (Wire Protocol 0x01). [Online]. Available: http://www.openflow.org/documents/openflow-spec-v1.0.0.pdf

[34] OpenFlow Switch Specification, Version 1.2 (Wire Protocol 0x03). [Online]. Available: https://www.opennetworking.org/images/stories/downloads/openflow/openflow-spec-v1.2.pdf

[35] OpenFlow Switch Specification, Version 1.3.0 (Wire Protocol 0x04). [Online]. Available: https://www.opennetworking.org/images/stories/downloads/specification/openflow-spec-v1.3.0.pdf

[36] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "NOX: towards an operating system for networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 3, pp. 105–110, July 2008.

[37] David Erickson. Beacon Home. [Online]. Available: https://openflow.stanford.edu/display/Beacon/Home/

[38] Z. Cai, A. L. Cox, and T. S. Eugene. Maestro: A System for Scalable OpenFlow Control. [Online]. Available: http://www.cs.rice.edu/~eugeneng/papers/TR10-11.pdf

[39] Floodlight. [Online]. Available: http://floodlight.openflowhub.org/

[40] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: a network programming language," in *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, 2011.

[41] Trema, Full-Stack OpenFlow Framework in Ruby and C. [Online]. Available: http://trema.github.com/trema/

[42] Node.flow. [Online]. Available: https://github.com/dreamerslab/node.flow

[43] Node.js. [Online]. Available: http://nodejs.org/

[44] B. Pfaff, J. Pettit, K. A. T. Koponen, M. Casado, and S. Shenker, "Extending networking into the virtualization layer," in *Proceedings of the ACM SIGCOMM HotNets*, 2009.

[45] Open vSwitch. [Online]. Available: http://openvswitch.org/

[46] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," in *Proceedings of the Ninth ACM SIGCOMM Workshop on Hot Topics in Networks*, 2010.

[47] NetFPGA. NetFPGA. [Online]. Available: http://netfpga.org/foswiki/bin/view/NetFPGA/OneGig/LearnMore

[48] N. Handigol, B. Heller, V. Jeyakumar, D. Maziéres, and N. McKeown, "Where is the debugger for my software-defined network?" in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks (HotSDN)*, 2012.

[49] NOX. [Online]. Available: http://www.noxrepo.org

[50] Maestro Platform. [Online]. Available: http://code.google.com/p/maestro-platform/

[51] R. Sherwood, M. Chan, A. Covington, G. Gibb, M. Flajslik, N. Handigol, T.-Y. Huang, P. Kazemian, M. Kobayashi, J. Naous, S. Seetharaman, D. Underhill, T. Yabe, K.-K. Yap, Y. Yiakoumis, H. Zeng, G. Appenzeller, R. Johari, N. McKeown, and G. Parulkar, "Carving research slices out of your production networks with OpenFlow," *SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 1, pp. 129–130, January 2010.

[52] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker, "Ethane: taking control of the enterprise," *SIGCOMM Comput. Commun. Rev.*, vol. 37, no. 4, pp. 1–12, October 2007.

[53] IETF. A Path Computation Element (PCE)-Based Architecture. [Online]. Available: http://tools.ietf.org/html/rfc4655

[54] A. Giorgetti, F. Cugini, F. Paolucci, and P. Castoldi, "OpenFlow and PCE architectures in Wavelength Switched Optical Networks," in *16th International Conference on Optical Network Design and Modeling (ONDM)*, April 2012.

[55] Energy Sciences Network (ESNet). On-Demand Secure Circuits and Advance Reservation System. [Online]. Available: http://tools.ietf.org/html/rfc4655

[56] V. Vokkarane. Progress report. [Online]. Available: http://www.cis.umassd.edu/~vvokkarane/common/reports/Y2Q1report.pdf

[57] K. Nichols, V. Jacobson, and L. Zhan. A Two-bit Differentiated Services Architecture for the Internet. [Online]. Available: http://tools.ietf.org/html/rfc2638

[58] R. Braga, E. Mota, and A. Passito, "Lightweight DDoS flooding attack detection using NOX/OpenFlow," in *IEEE 35th Conference on Local Computer Networks (LCN)*, October 2010.

[59] G. Yao, J. Bi, and P. Xiao, "Source address validation solution with OpenFlow/NOX architecture," in *19th IEEE International Conference on Network Protocols (ICNP)*, 2011.

[60] D. Staessens, S. Sharma, D. Colle, M. Pickavet, and P. Demeester, "Software defined networking: Meeting carrier grade requirements," in *18th IEEE Workshop on Local Metropolitan Area Networks (LANMAN)*, 2011.

[61] N. Handigol, S. Seetharaman, M. Flajslik, N. McKeown, and R. Johari, "Plug-n-Serve: Load-balancing web traffic using Open-Flow," in *ACM SIGCOMM Demo*, 2009.

[62] J. Suh, H. Choi, W. Yoon, T. You, T. Kwon, and Y. Choi, "Implementation of a Content-oriented Networking Architecture (CONA): A Focus on DDoS Countermeasure," in *European NetFPGA Developers Workshop*, September 2010.

[63] Y. Chu, M. Tseng, Y. Chen, Y. Chou, and Y. Chen, "A novel design for future on-demand service and security," in *12th IEEE International Conference on Communication Technology (ICCT)*, 2010.

[64] X. Liu, H. Xue, X. Feng, and Y. Dai, "Design of the multi-level security network switch system which restricts covert channel," in *IEEE 3rd International Conference on Communication Software and Networks (ICCSN)*, 2011.

[65] J. H. Jafarian, E. Al-Shaer, and Q. Duan, "OpenFlow random host mutation: transparent moving target defense using software defined networking," in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks (HotSDN)*, 2012.

[66] S. Gutz, A. Story, C. Schlesinger, and N. Foster, "Splendid isolation: a slice abstraction for software-defined networks," in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks (HotSDN)*, 2012.

[67] Y. Yamasaki, Y. Miyamoto, J. Yamato, H. Goto, and H. Sone, "Flexible Access Management System for Campus VLAN Based on OpenFlow," in *IEEE/IPSJ 11th International Symposium on Applications and the Internet (SAINT)*, 2011.

[68] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *Proceedings of ACM SIGCOMM*, 2012.

[69] M. Reitblatt, N. Foster, J. Rexford, and D. Walker, "Consistent updates for software defined networks: change you can believe in!" in *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, 2011.

[70] N. Katta, J. Rexford, and D. Walker, "Incremental consistent updates," in *ACM SIGCOMM HotSDN Workshop*, 2013.

[71] R. McGeer, "A safe, efficient update protocol for OpenFlow networks," in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks (HotSDN)*, 2012.

[72] S. Ghorbani and M. Caesar, "Walk the line: consistent network updates with bandwidth guarantees," in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks (HotSDN)*, 2012.

[73] J. Luo, J. Pettit, M. Casado, J. Lockwood, and N. McKeown, "Prototyping Fast, Simple, Secure Switches for Ethane," in *15th Annual IEEE Symposium on High-Performance Interconnects*, 2007.

[74] D. M. F. Mattos, N. C. Fernandes, V. T. da Costa, L. P. Cardoso, M. E. M. Campista, L. H. M. K. Costa, and O. Duarte, "OMNI: OpenFlow MaNagement Infrastructure," in *2011 International Conference on the Network of the Future (NOF)*, 2011.

[75] G. Gibb, H. Zeng, and N. McKeown, "Initial thoughts on custom network processing via waypoint services," in *3rd Workshop on Infrastructures for Software/Hardware Co-Design*, 2011.

[76] A. Voellmy, H. Kim, and N. Feamster, "Procera: a language for high-level reactive network control," in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks (HotSDN)*, 2012.

[77] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi, "Hierarchical policies for software defined networks," in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks (HotSDN)*, 2012.

[78] Y. Nakagawa, K. Hyoudou, and T. Shimizu, "A management method of IP multicast in overlay networks using OpenFlow," in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks (HotSDN)*, 2012.

[79] G. Gibb, H. Zeng, and N. McKeown, "Outsourcing network functionality," in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks (HotSDN)*, 2012.

[80] R. Raghavendra, J. Lobo, and K.-W. Lee, "Dynamic graph query primitives for SDN-based cloudnetwork management," in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks (HotSDN)*, 2012.

[81] D. Simeonidou, R. Nejabati, and S. Azodolmolky, "Enabling the future optical Internet with OpenFlow: A paradigm shift in providing intelligent optical network services," in *2011 13th International Conference on Transparent Optical Networks (ICTON)*, 2011.

[82] S. Das, G. Parulkar, and N. McKeown, "Unifying Packet and Circuit Switched Networks," in *Proceedings of IEEE GLOBECOM Workshops*, 2009.

[83] S. Das, G. Parulkar, N. McKeown, P. Singh, D. Getachew, and L. Ong, "Packet and circuit network convergence with OpenFlow," in *2010 Conference on (OFC/NFOEC) Optical Fiber Communication (OFC), collocated National Fiber Optic Engineers Conference*, 2010.

[84] S. Das, Y. Yiakoumis, G. Parulkar, N. McKeown, P. Singh, D. Getachew, and P. D. Desai, "Application-aware aggregation and traffic engineering in a converged packet-circuit network," in *Optical Fiber Communication Conference and Exposition (OFC/NFOEC) and the National Fiber Optic Engineers Conference*, 2011.

[85] S. Das, A. R. Sharafat, G. Parulkar, and N. McKeown, "MPLS with a simple OPEN control plane," in *Optical Fiber Communication Conference and Exposition (OFC/NFOEC), 2011 and the National Fiber Optic Engineers Conference*, 2011.

[86] O. El Ferkouss, S. Correia, R. Ben Ali, Y. Lemieux, M. Julien, M. Tatipamula, and O. Cherkaoui, "On the Flexibility of MPLS Applications over an OpenFlow-Enabled Network," in *2011 IEEE Global Telecommunications Conference (GLOBECOM 2011)*, 2011.

[87] J. Kempf, S. Whyte, J. Ellithorpe, P. Kazemian, M. Haitjema, N. Beheshti, S. Stuart, and H. Green, "OpenFlow MPLS and the open source label switched router," in *Proceedings of the 23rd International Teletraffic Congress*, 2011.

[88] A. R. Sharafat, S. Das, G. Parulkar, and N. McKeown, "MPLS-TE and MPLS VPNS with OpenFlow," in *Proceedings of the ACM SIGCOMM*, 2011.

[89] N. Handigol, S. Seetharaman, M. Flajslik, A. Gember, N. McKeown, G. Parulkar, A. Akella, N. Feamster, R. Clark, A. Krishnamurthy, V. Brajkovic, and T. Anderson, "Aster*x: Load-Balancing Web Traffic over Wide-Area Networks," in *Open Networking Summit Demo*, 2011.

[90] R. Wang, D. Butnariu, and J. Rexford, "OpenFlow-Based Server Load Balancing Gone Wild," in *Proceedings of the 11th USENIX conference on Hot topics in management of internet, cloud, and enterprise networks and services*, 2011.

[91] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester, "Enabling fast failure recovery in OpenFlow networks," in *8th International Workshop on the Design of Reliable Communication Networks (DRCN)*, 2011.

[92] A. Khurshid, W. Zhou, M. Caesar, and P. B. Godfrey, "VeriFlow: verifying network-wide invariants in real time," in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks (HotSDN)*, 2012.

[93] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu, "A security enforcement kernel for OpenFlow networks," in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks (HotSDN)*, 2012.

[94] S. Azodolmolky, R. Nejabati, E. Escalona, R. Jayakumar, N. Efstathiou, and D. Simeonidou, "Integrated OpenFlow–GMPLS control plane: an overlay model for software defined packet over optical networks," *Opt. Express*, vol. 19, no. 26, pp. B421–B428, December 2011.

[95] S. Das, G. Parulkar, and N. McKeown, "Why OpenFlow/SDN Can Succeed Where GMPLS Failed," in *European Conference and Exhibition on Optical Communication*. Optical Society of America, 2012, p. Tu.1.D.1.

[96] O. Baldonado. SDN, OpenFlow, and next-generation data center networks. [Online]. Available: http://www.eetimes.com/design/embedded/4371543/SDN--OpenFlow--and-next-generation-data-center-networks

[97] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: dynamic flow scheduling for data center networks," in *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, 2010.

[98] C. Rotsos, R. Mortier, A. Madhavapeddy, B. Singh, and A. W. Moore, "Cost, performance and flexibility in OpenFlow: Pick three," *Workshop on Software Defined Networks Co-located with the IEEE*

International Conference on Communications (ICC), 2012. [Online]. Available: http://www.cs.nott.ac.uk/~rmm/papers/pdf/iccsdn12-mirageof.pdf

[99] R. Bennesby, P. Fonseca, E. Mota, and A. Passito, "An inter-as routing component for software-defined networks," in *IEEE Network Operations and Management Symposium (NOMS)*, 2012.

[100] K.-K. Yap, M. Kobayashi, R. Sherwood, T.-Y. Huang, M. Chan, N. Handigol, and N. McKeown, "OpenRoads: empowering research in mobile networks," *SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 1, pp. 125–126, January 2010.

[101] OpenFlow Wireless. [Online]. Available: http://www.openflow.org/wk/index.php/OpenFlow_Wireless

[102] K.-K. Yap, M. Kobayashi, D. Underhill, S. Seetharaman, P. Kazemian, and N. McKeown, "The Stanford OpenRoads Deployment," in *Proceedings of the 4th ACM International Workshop on Experimental Evaluation and Characterization*, 2009.

[103] K.-K. Yap, S. Katti, G. Parulkar, and N. McKeown, "Delivering capacity for the mobile internet by stitching together networks," in *Proceedings of the 2010 ACM Workshop on Wireless of the Students, by the Students, for the Students*, 2010.

[104] K.-K. Yap, R. Sherwood, M. Kobayashi, T.-Y. Huang, M. Chan, N. Handigol, N. McKeown, and G. Parulkar, "Blueprint for introducing innovation into wireless mobile networks," in *Proceedings of the Second ACM SIGCOMM Workshop on Virtualized Infrastructure Systems and Architectures*, 2010.

[105] T.-Y. Huang, K.-K. Yap, B. Dodson, M. S. Lam, and N. McKeown, "PhoneNet: a phone-to-phone network for group communication within an administrative domain," in *Proceedings of the Second ACM SIGCOMM Workshop on Networking, Systems, and Applications on Mobile Handhelds*, 2010.

[106] M. Bansal, J. Mehlman, S. Katti, and P. Levis, "OpenRadio: a programmable wireless dataplane," in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks (HotSDN)*, 2012.

[107] L. Suresh, J. Schulz-Zander, R. Merz, A. Feldmann, and T. Vazao, "Towards programmable enterprise WLANS with Odin," in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks (HotSDN)*, 2012.

[108] L. Lu, Y. Xiao, and H. Du, "OpenFlow control for cooperating AQM scheme," in *2010 IEEE 10th International Conference on Signal Processing (ICSP)*, 2010.

[109] K.-K. Yap, T.-Y. Huang, B. Dodson, M. S. Lam, and N. McKeown, "Towards software-friendly networks," in *Proceedings of the First ACM Asia-Pacific Workshop on Systems*, 2010.

[110] M. R. Nascimento, C. E. Rothenberg, M. R. Salvador, and M. F. Magalhães, "QuagFlow: partnering Quagga with OpenFlow," in *Proceedings of the ACM SIGCOMM 2010 Conference*, 2010.

[111] M. R. Nascimento, C. E. Rothenberg, M. R. Salvador, C. N. A. Corrêa, S. C. de Lucena, and M. F. Magalhães, "Virtual routers as a service: the RouteFlow approach leveraging software-defined networks," in *Proceedings of the 6th International Conference on Future Internet Technologies*, 2011.

[112] C. E. Rothenberg, M. R. Nascimento, M. R. Salvador, C. N. A. Corrêa, S. Cunha de Lucena, and R. Raszuk, "Revisiting routing control platforms with the eyes and muscles of software-defined networking," in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks (HotSDN)*, 2012.

[113] H. E. Egilmez, B. Gorkemli, A. M. Tekalp, and S. Civanlar, "Scalable video streaming over OpenFlow networks: An optimization framework for QoS routing," in *18th IEEE International Conference on Image Processing (ICIP)*, 2011.

[114] OpenFlow Stanford Deployment. [Online]. Available: http://www.openflow.org/wp/stanford-deployment/

[115] Clemson OpenFlow Agregate. [Online]. Available: http://groups.geni.net/geni/wiki/GeniAggregate/ClemsonOpenFlow

[116] Georgia Tech OpenFlow Agregate. [Online]. Available: http://groups.geni.net/geni/wiki/GeniAggregate/GeorgiaTechOpenFlow

[117] Indiana OpenFlow Agregate. [Online]. Available: http://groups.geni.net/geni/wiki/GeniAggregate/IndianaOpenFlow

[118] KSU Lab OpenFlow Aggregate. [Online]. Available: http://groups.geni.net/geni/wiki/GeniAggregate/KansasStateOpenFlow

[119] Rutgers OpenFlow Agregate. [Online]. Available: http://groups.geni.net/geni/wiki/GeniAggregate/RutgersOpenFlow

[120] University of Washington OpenFlow Agregate. [Online]. Available: http://groups.geni.net/geni/wiki/GeniAggregate/WashingtonOpenFlow

[121] Winsconsin OpenFlow Agregate. [Online]. Available: http://groups.geni.net/geni/wiki/GeniAggregate/WisconsinOpenFlow

[122] GENI, Exploring networks of the future. [Online]. Available: http://www.geni.net/

[123] Internet2. [Online]. Available: www.internet2.edu

[124] National LambdaRail. [Online]. Available: http://www.nlr.net

[125] Testbed Networks: Provided by NLR. [Online]. Available: www.nlr.net/testbeds.php

[126] GENI. GENI OpenFlow Backbone Deployment at Internet2. [Online]. Available: http://groups.geni.net/geni/wiki/OFI2

[127] Internet2. Nation's First 100G Open, Nationwide, Software-Defined Network Launches for Education, Research, Industry and Innovators. [Online]. Available: http://internet2.edu/news/pr/2012.10.01.nations-first-100g-national-scale-network-launches.html

[128] Internet2. Internet2 Mailing List Service. [Online]. Available: https://lists.internet2.edu/sympa/arc/i2-news/2012-07/msg00002.html

[129] "Energy Sciences Network," https://www.es.net/.

[130] Advanced Networking Initiative (ANI). [Online]. Available: http://www.es.net/RandD/advanced-networking-initiative/

[131] IP8800 OpenFlow Networking. [Online]. Available: http://support.necam.com/pflow/legacy/ip8800/

[132] ESNet. 100G Testbed. [Online]. Available: http://www.es.net/RandD/100g-testbed/

[133] Energy Sciences Network ESNet. Proposal Process. [Online]. Available: http://www.es.net/RandD/100g-testbed/proposal-process

[134] Ofelia. [Online]. Available: http://www.fp7-ofelia.eu/news-and-events/press-releases/ofelia-openflow-facility-now-open-for-experiments/

[135] MRI-R2 Consortium: Development of Dynamic Network System (DYNES). [Online]. Available: http://www.internet2.edu/ion/dynes.html

[136] Open Networking Summit 2012 Program. [Online]. Available: http://opennetsummit.org/

[137] S. Levy. Going With the Flow: Google's Secret Switch to the Next Wave of Networking. [Online]. Available: http://www.wired.com/wiredenterprise/2012/04/going-with-the-flow-google/all/1

[138] M. Jarschel, S. Oechsner, D. Schlosser, R. Pries, S. Goll, and P. Tran-Gia, "Modeling and performance evaluation of an OpenFlow architecture," in *23rd International Teletraffic Congress (ITC)*, 2011.

[139] A. Bianco, R. Birke, L. Giraudo, and M. Palacin, "OpenFlow Switching: Data Plane Performance," in *IEEE International Conference on Communications (ICC)*, 2010.

[140] D. Levin, A. Wundsam, B. Heller, N. Handigol, and A. Feldmann, "Logically centralized?: state distribution trade-offs in software defined networks," in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks (HotSDN)*, 2012.

[141] B. Heller, R. Sherwood, and N. McKeown, "The controller placement problem," *SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 4, pp. 473–478, Sep. 2012.

[142] L. Liu, T. Tsuritani, I. Morita, H. Guo, and J. Wu, "Experimental validation and performance evaluation of OpenFlow-based wavelength path control in transparent optical networks," *Opt. Express*, vol. 19, no. 27, pp. 26 578–26 578, Sep. 2012.

[143] S. Hassas Yeganeh and Y. Ganjali, "Kandoo: a framework for efficient and scalable offloading of control applications," in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks (HotSDN)*, 2012.

[144] J. C. Mogul and P. Congdon, "Hey, you darned counters!: get off my ASIC!" in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks (HotSDN)*, 2012.

[145] G. Lu, R. Miao, Y. Xiong, and C. Guo, "Using CPU as a traffic co-processing unit in commodity switches," in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks (HotSDN)*, 2012.

[146] L. Vanbever, J. Reich, T. Benson, N. Foster, and J. Rexford, "HotSwap: Correct and efficient controller upgrades for Software-Defined Networks," in *ACM SIGCOMM HotSDN Workshop*, 2013.

[147] A. Akhunzada, E. Ahmed, A. Gani, M. Khan, M. Imran, and S. Guizani, "Securing software defined networks: taxonomy, requirements, and open issues," *Communications Magazine, IEEE*, vol. 53, no. 4, pp. 36–44, 2015.

[148] S. H. Yeganeh, A. Tootoonchian, and Y. Ganjali, "On scalability of software-defined networking," *Communications Magazine, IEEE*, vol. 51, no. 2, pp. 136–141, 2013.

[149] X. Guan, B.-Y. Choi, and S. Song, "Reliability and scalability issues in software defined network frameworks," in *Research and Educational Experiment Workshop (GREE), 2013 Second GENI*. IEEE, 2013, pp. 102–103.

[150] E. Hernandez-Valencia, S. Izzo, and B. Polonsky, "How will nfv/sdn transform service provider opex?" *Network, IEEE*, vol. 29, no. 3, pp. 60–67, 2015.

[151] S. Sezer, S. Scott-Hayward, P.-K. Chouhan, B. Fraser, D. Lake, J. Finnegan, N. Viljoen, M. Miller, and N. Rao, "Are we ready for sdn? implementation challenges for software-defined networks," *Communications Magazine, IEEE*, vol. 51, no. 7, pp. 36–43, 2013.

[152] R. Jain and S. Paul, "Network virtualization and software defined networking for cloud computing: a survey," *Communications Magazine, IEEE*, vol. 51, no. 11, pp. 24–31, 2013.

[153] B. Nunes, M. Mendonca, X.-N. Nguyen, K. Obraczka, T. Turletti *et al.*, "A survey of software-defined networking: Past, present, and future of programmable networks," *Communications Surveys & Tutorials, IEEE*, vol. 16, no. 3, pp. 1617–1634, 2014.

[154] W. Xia, Y. Wen, C. H. Foh, D. Niyato, and H. Xie, "A survey on software-defined networking," *Communications Surveys & Tutorials, IEEE*, vol. 17, no. 1, pp. 27–51, 2014.

[155] A. Lara, A. Kolasani, and B. Ramamurthy, "Simplifying network management using Software Defined Networking and OpenFlow," in *2012 IEEE 6th International Conference on Advanced Networks and Telecommunication Systems (ANTS)*, December. 2012, pp. 1 –5.

[156] H. Kim and N. Feamster, "Improving network management with software defined networking," *Communications Magazine, IEEE*, vol. 51, no. 2, pp. 114–119, February 2013.

[157] S. Shin and G. Gu, "Cloudwatcher: Network security monitoring using openflow in dynamic cloud networks (or: How to provide security monitoring as a service in clouds?)," in *2012 20th IEEE International Conference on Network Protocols (ICNP)*, Austin, Texas, October 2012, pp. 1–6.

[158] S. Shin, P. Porras, V. Yegneswaran, M. Fong, G. Gu, and M. Tyson, "FRESCO: Modular Composable Security Services for Software-Defined Networks," in *Internet Society Network and Distributed System Security Symposium (NDSS)*, San Diego, California, U.S.A., February 2013.

[159] A. K. Bandara, E. C. Lupu, and A. Russo, "Using event calculus to formalise policy specification and analysis," in *IEEE Workshop on Policies for Distributed Systems and Networks*, Lake Como, Italy, June 2003.

[160] A. K. Bandara, E. C. Lupu, J. Moffett, and A. Russo, "A goal-based approach to policy refinement," in *IEEE Workshop on Policies for Distributed Systems and Networks*, Yorktown Heights, NY, USA, June 2004.

[161] A. K. Bandara, A. Kakas, E. C. Lupu, and A. Russo, "Using argumentation logic for firewall policy specification and analysis," in *Large Scale Management of Distributed Systems*. Springer, 2006, pp. 185–196.

[162] J. Rubio-Loyola, J. Serrat, M. Charalambides, P. Flegkas *et al.*, "Using linear temporal model checking for goal-oriented policy refinement frameworks," in *IEEE Workshop on Policies for Distributed Systems and Networks*, Stockholm, Sweden, June 2005.

[163] J. Rubio-Loyola, J. Serrat, M. Charalambides, P. Flegkas, and G. Pavlou, "A methodological approach toward the refinement problem in policy-based management systems," *IEEE Communications Magazine*, vol. 44, no. 10, pp. 60–68, 2006.

[164] M. Charalambides, P. Flegkas, G. Pavlou, A. Bandara *et al.*, "Policy conflict analysis for quality of service management," in *IEEE International Workshop on Policies for Distributed Systems and Networks*, Stockholm, Sweden, June 2005.

[165] M. Charalambides, P. Flegkas, G. Pavlou, J. Rubio-Loyola *et al.*, "Dynamic policy analysis and conflict resolution for diffserv quality of service management," in *IEEE/IFIP Network Operations and Management Symposium (NOMS)*, Vancouver, Canada, April 2006.

[166] D. Agrawal, K.-W. Lee, and J. Lobo, "Policy-based management of networked computing systems," *Communications Magazine, IEEE*, vol. 43, no. 10, pp. 69–75, October 2005.

[167] R. Bhatia, J. Lobo, and M. Kohli, "Policy evaluation for network management," in *IEEE INFOCOM*, vol. 3, March 2000.

[168] E. Bertino, C. Brodie, S. Calo, L. Cranor, C. Karat *et al.*, "Analysis of privacy and security policies," *IBM Journal of Research and Development*, vol. 53, no. 2, pp. 3:1–3:18, March 2009.

[169] S. Lange, S. Gebert, T. Zinner, P. Tran-Gia *et al.*, "Heuristic approaches to the controller placement problem in large scale SDN networks," *Network and Service Management, IEEE Transactions on*, vol. 12, no. 1, pp. 4–17, March 2015.

[170] D. Tuncer, M. Charalambides, S. Clayman, and G. Pavlou, "Adaptive resource management and control in software defined networks," *Network and Service Management, IEEE Transactions on*, vol. 12, no. 1, pp. 18–33, March 2015.

[171] M. Wichtlhuber, R. Reinecke, and D. Hausheer, "An SDN-based CDN/ISP collaboration architecture for managing high-volume flows," *Network and Service Management, IEEE Transactions on*, vol. 12, no. 1, pp. 48–60, March 2015.

[172] M. Bari, S. Chowdhury, R. Ahmed, and R. Boutaba, "Policycop: An autonomic qos policy enforcement framework for software defined networks," in *Future Networks and Services (SDN4FNS), 2013 IEEE SDN for*, November 2013, pp. 1–7.

[173] V. Sekar, S. Ratnasamy, M. K. Reiter, N. Egi, and G. Shi, "The middlebox manifesto: enabling innovation in middlebox deployment," in *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, 2011.

[174] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella, "Towards an elastic distributed sdn controller," *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 7–12, August 2013.

[175] A. Tootoonchian and Y. Ganjali, "Hyperflow: A distributed control plane for openflow," in *Proceedings of the 2010 Internet Network Management Conference on Research on Enterprise Networking*, ser. INM/WREN'10, 2010.

[176] N. Farrington, G. Porter, S. Radhakrishnan, H. H. Bazzaz, V. Subramanya, Y. Fainman, G. Papen, and A. Vahdat, "Helios: a hybrid electrical/optical switch architecture for modular data centers," *SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 4, August 2010.

[177] F. Yonghong, B. Jun, W. Jianping, C. Ze, W. Ke, and L. Min, "A dormant multi-controller model for software defined networking," *Communications, China*, vol. 11, no. 3, pp. 45–55, March 2014.

[178] Carnegie Mellon University. Residence Hall and Dedicated Remote Access. [Online]. Available: http://www.cmu.edu/computing/network/guidelines/network-res.html

[179] GENI, "GENI Portal," http://portal.geni.net.

[180] A. Sperotto, R. Sadre, D. F. van Vliet, and A. Pras, "A labeled data set for flow-based intrusion detection," in *Proceedings of the 9th IEEE International Workshop on IP Operations and Management, IPOM 2009, Venice, Italy*, ser. Lecture Notes in Computer Science, vol. 5843. Springer Verlag, October 2009, pp. 39–50.

[181] The Bro Network Security Monitor. [Online]. Available: www.bro.org

[182] nDPI: Open and Extensible LGPLv3 Deep Packet Inspection Library. [Online]. Available: http://www.ntop.org/products/ndpi/

[183] nmap security scanner. [Online]. Available: www.nmap.org

[184] Scapy. [Online]. Available: http://www.secdev.org/projects/scapy/

[185] R. Hofstede, L. Hendriks, A. Sperotto, and A. Pras, "SSH Compromise Detection using NetFlow/IPFIX," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 5, pp. 20–26, 2014.

[186] E. Dart, L. Rotman, B. Tierney, M. Hester, and J. Zurawski, "The Science DMZ: A Network Design Pattern for Data-intensive Science," in *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 85:1–85:10.

[187] J. Zurawski, S. Balasubramanian, A. Brown, E. Kissel, A. Lake, M. Swany, B. Tierney, and M. Zekauskas, "perfsonar: On-board diagnostics for big data," in *Proceedings of the 1st Workshop on Big Data and Science: Infrastructure and Services Co-located with IEEE International Conference on Big Data 2013*, ser. IEEE BigData 2013, 2013.

[188] Internet2 AL2S. [Online]. Available: http://www.internet2.edu/products-services/advanced-networking/layer-2-services/

[189] M. Casado, T. Koponen, R. Ramanathan, and S. Shenker, "Virtualizing the network forwarding plane," in *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow*, ser. PRESTO '10. New York, NY, USA: ACM, 2010, pp. 8:1–8:6. [Online]. Available: http://doi.acm.org/10.1145/1921151.1921162

[190] J. Matias, E. Jacob, D. Sanchez, and Y. Demchenko, "An OpenFlow Based Network Virtualization Framework for the Cloud," in *IEEE Third International Conference on Cloud Computing Technology and Science (CloudCom)*, 2011, pp. 672–678.

[191] NEC. NEC ProgrammableFlow: Redefining Cloud Network Virtualization with OpenFlow. [Online]. Available: http://www.nec.com/en/global/prod/pflow/images_documents/NEC_ProgrammableFlow_Redefining_Cloud_Network_Virtualization_with_OpenFlow.pdf

[192] D. Drutskoy, E. Keller, and J. Rexford, "Scalable network virtualization in software-defined networks," *Internet Computing, IEEE*, vol. 17, no. 2, pp. 20–27, 2013.

[193] P. Skoldstrom and K. Yedavalli, "Network virtualization and resource allocation in OpenFlow-based wide area networks," in *IEEE International Conference on Communications (ICC)*, 2012, pp. 6622–6626.

[194] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving high utilization with software-driven wan," *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 15–26, August 2013.

[195] A. Mahimkar, A. Chiu, R. Doverspike, M. D. Feuer, P. Magill, E. Mavrogiorgis, J. Pastor, S. L. Woodward, and J. Yates, "Bandwidth on demand for inter-data center communication," in *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*. ACM, 2011, p. 24.

[196] R. Doverspike, G. Clapp, P. Douyon, D. M. Freimuth, K. Gullapalli, B. Han, J. Hartley, A. Mahimkar, E. Mavrogiorgis, J. OConnor *et al.*, "Using SDN technology to enable cost-effective bandwidth-on-demand for cloud services [Invited]," *Journal of Optical Communications and Networking*, vol. 7, no. 2, pp. A326–A334, 2015.

[197] D. Drutskoy, E. Keller, and J. Rexford, "Scalable network virtualization in software-defined networks," *Internet Computing, IEEE*, vol. 17, no. 2, pp. 20–27, 2013.

[198] M. R. Nascimento, C. E. Rothenberg, M. R. Salvador, C. N. Corrêa, S. C. de Lucena, and M. F. Magalhães, "Virtual routers as a service: the routeflow approach leveraging software-defined networks," in *Proceedings of the 6th International Conference on Future Internet Technologies*. ACM, 2011.

[199] NEC. NEC ProgrammableFlow: Redefining Cloud Network Virtualization with OpenFlow. [Online]. Available: http://www.nec.com/en/global/prod/pflow/images_documents/NEC_ProgrammableFlow_Redefining_Cloud_Network_Virtualization_with_OpenFlow.pdf

[200] B. Heller, R. Sherwood, and N. McKeown, "The controller placement problem," in *Proceedings of the first workshop on Hot topics in software defined networks*. ACM, 2012, pp. 7–12.

[201] D. Hock, M. Hartmann, S. Gebert, M. Jarschel, T. Zinner, and P. Tran-Gia, "Pareto-optimal resilient controller placement in SDN-based core networks," in *IEEE 25th International Teletraffic Congress (ITC)*, 2013.

[202] Y.-N. Hu, W.-D. Wang, X.-Y. Gong, X.-R. Que, and S.-D. Cheng, "On the placement of controllers in software-defined networks," *The Journal of China Universities of Posts and Telecommunications*, vol. 19, pp. 92–171, 2012.

[203] Y. Hu, W. Wang, X. Gong, X. Que, and S. Cheng, "On reliability-optimized controller placement for software-defined networks," *Communications, China*, vol. 11, no. 2, pp. 38–54, 2014.

[204] Energy Sciences Network (ESNet). Science DMZ. [Online]. Available: https://fasterdata.es.net/science-dmz/science-dmz-security/

[205] P. Lin, J. Hart, U. Krishnaswamy, T. Murakami, M. Kobayashi, A. Al-Shabibi, K.-C. Wang, and J. Bi, "Seamless Interworking of SDN and IP," *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 475–476, August 2013.

[206] P. Lin, J. Bi, Z. Chen, Y. Wang, H. Hu, and A. Xu, "WE-bridge: West-east Bridge for SDN Inter-domain Network Peering," in *INFOCOM Workshops*, April 2014.

[207] V. Kotronis, X. Dimitropoulos, and B. Ager, "Outsourcing the Routing Control Logic: Better Internet Routing Based on SDN principles," in *ACM Workshop on Hot Topics in Networks*, November 2012.

[208] I. Monga, E. Pouyoul, and C. Guok, "Software Defined Networking for big-data science: Architectural models from campus to the WAN," in *Proceedings of the Supercomputing Conference*, 2012.

[209] On.Lab, "OpenSource SDN Stack," http://onlab.us/tools.html.

[210] "Openfire," http://www.igniterealtime.org/projects/openfire/.

[211] "sFlow," http://www.sflow.org/.

[212] "Controlling large flows with sFlow," http://blog.sflow.com/2013/05/controlling-large-flows-with-openflow.html.

[213] "Basic ONOS tutorial," https://wiki.onosproject.org/display/ONOS/Basic+ONOS+Tutorial.

[214] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard, "Networking named content," in *Proceedings of the 5th international conference on Emerging networking experiments and technologies*, Rome, Italy, December 2009, pp. 1–12.

[215] D. G. Andersen, H. Balakrishnan, N. Feamster, T. Koponen, D. Moon, and S. Shenker, "Accountable Internet Protocol (AIP)," in *Proc. ACM SIGCOMM*, Seattle, WA, U.S.A., August 2008.

[216] "Host Identity Protocol (HIP)," http://tools.ietf.org/html/rfc5201, 2008.

[217] "New Generation Networks," http://www2.nict.go.jp/w/w100/index-e.

[218] "Networking Technology and Systems: Future Internet Design (FIND), NSF program solicitation," 2007.

[219] M. Lemke, "Position Statement: FIRE, NSF/OECD Workshop on Social and Economic Factors Shaping the future of the Internet," January 2007.

[220] "FP7 Information and Communication Technologies: Pervasive and Trusted Network and Service Infrastructures, European Commission," 2007.

[221] Cisco, "Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2011-2016."

[222] S. C. Nelson, G. Bhanage, and D. Raychaudhuri, "GSTAR: Generalized Storage-aware Routing for Mobilityfirst in the Future Mobile Internet," in *Proc. of MobiArch*, August 2011.

[223] A. Krishnamoorthy, "Implementation and Evaluation of the MobilityFirst Protocol Stack on Software-Defined Network Platforms. M. Sc. Thesis, WINLAB, Rutgers University," http://dx.doi.org/doi:10.7282/T3G44N97.

[224] S. Melle, D. Perkins, and C. Villamizar, "Network Cost Savings from Router Bypass in IP over WDM Core Networks," in *IEEE/OSA Conference on Optical Fiber Communication/National Fiber Optic Engineers Conference*, February 2008.

[225] Y. Lui, G. Shen, and W. Shao, "Design for Energy-efficient IP over WDM Networks with Joint Lightpath Bypass and Router-card Sleeping Strategies," *IEEE/OSA Journal of Optical Communications and Networking*, vol. 5, no. 11, pp. 1122–1138, November 2013.

[226] M. Karol, "A Distributed Algorithm for Optimal (Optical) Bypass of IP Routers to Improve Network Performance and Efficiency," in *IEEE Conference on Information Sciences and Systems (CISS)*, March 2011.

[227] Internet Engineering Task Force, "VXLAN: A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks." [Online]. Available: http://datatracker.ietf.org/doc/draft-mahalingam-dutt-dcops-vxlan/

[228] U. Lakshman and L. Lobo, "MPLS Traffic Engineering," http://www.ciscopress.com/articles/article.asp?p=426640.

[229] A. Tootoonchian and Y. Ganjali, "HyperFlow: A Distributed Control Plane for OpenFlow," in *USENIX Conference on Research on Enterprise Networking Internet Network Management* , April 2010.

[230] N. Feamster, J. Rexford, S. Shenker, R. Clark, R. Hutchins, D. Levin, , and J. Bailey, "SDX: A Software-Defined Internet Exchange," in *Open Networking Summit Research Track*, April 2013.

[231] Open Networking Foundation. OpenFlow Switch Specification 1.4.0. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf

[232] A. Gämperli, V. Kotronis, and X. Dimitropoulos, "Evaluating the Effect of Centralization on Routing Convergence on a Hybrid BGP-SDN Emulation Framework," in *Proceedings of ACM SIGCOMM*, August 2014.

[233] X. Yang, C. Tracy, J. Sobieski, and T. Lehman, "GMPLS-Based Dynamic Provisioning and Traffic Engineering of High-Capacity Ethernet Circuits in Hybrid Optical/Packet Networks," in *IEEE INFOCOM*, April 2006.

[234] C. Guok, "ESnet On-Demand Secure Circuits and Advance Reservation System (OSCARS)," in *Internet2 Joint Techs Workshop*, February 2005.

[235] Energy Sciences Network, "OSCARS," https://www.es.net/engineering-services/oscars/.

[236] A. Tomaszewski, M. Pióro, and M. Mycek, "Distributed inter-domain link capacity optimization for inter-domain IP/MPLS routing," in *Proceedings of IEEE GLOBECOM*, 2007.

[237] M. Roughan and Y. Zhang, "GATEway: symbiotic inter-domain traffic engineering," *Telecommunication Systems*, vol. 47, no. 1-2, pp. 3–17, 2011.

[238] M. Chamania, M. Caria, and A. Jukan, "Effective usage of dynamic circuits for IP routing," in *Proceedings of IEEE ICC*, 2010.

[239] J. Rivillo, J. Hernandez, and I. Phillips, "On the Efficient Detection of Elephant Flows in Aggregated Network Traffic," in *London Communication Symposium*, September 2005.

[240] T. Mori, M. Uchida, R. Kawahara, J. Pan, and S. Goto, "Identifying Elephant Flows Through Periodically Sampled Packets," in *ACM SIGCOMM Conference on Internet Measurement*, August 2004.

[241] K. Psounis, A. Ghosh, B. Prabhakar, and G. Wang, "SIFT: A Simple Algorithm for Tracking Elephant Flows, and Taking Advantage of Power Laws," in *Annual Allerton Conference on Control, Communication and Computing*, September 2005.

[242] "sFlow-RT," http://www.inmon.com/products/sFlow-RT.php.

[243] "Label Distribution Protocol RFC," https://tools.ietf.org/html/rfc5036.

[244] M. Davy, G. Parulkar, J. van Reijendam, D. Schmiedt, R. Clar, C. Tengi, I. Seskar, P. Christian, I. Cote, and G. China. A Case for Expanding OpenFlow/SDN Deployments On university Campuses. [Online]. Available: http://www.openflow.org/wp/wp-content/uploads/2011/07/GENI-Workshop-Whitepaper.pdf

[245] Software-defined Networking. [Online]. Available: http://www.ieee-infocom.org/2009/keynotes.html

[246] H. Kim and N. Feamster, "Improving network management with software defined networking," *Communications Magazine, IEEE*, vol. 51, no. 2, pp. 114–119, February 2013.

[247] A. Voellmy, H. Kim, and N. Feamster, "Procera: A language for high-level reactive network control," in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, ser. HotSDN '12, New York, NY, USA, 2012, pp. 43–48.

[248] D. Kreutz, F. M. Ramos, and P. Verissimo, "Towards secure and dependable software-defined networks," in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '13, New York, NY, USA, 2013.

[249] S. Shin and G. Gu, "Attacking software-defined networks: A first feasibility study," 2013, poster presented at HotSDN 2013, Hong Kong.

[250] K. Benton, L. Camp, and C. Small, "Openflow vulnerability assessment," 2013, poster presented at HotSDN 2013, Hong Kong.

[251] Arachni: A web application security scanner framework. [Online]. Available: http://www.arachni-scanner.com/

[252] A. J. Bennieston. NMAP - A Stealth Port Scanner. [Online]. Available: http://nmap.org/bennieston-tutorial/

[253] GENI. GENI Network Stitching Sites. [Online]. Available: http://groups.geni.net/geni/wiki/GeniNetworkStitchingSites

[254] GENI. Scaling Up: How to Grow the Topology of an Existing Experiment. [Online]. Available: http://groups.geni.net/geni/wiki/GEC21Agenda/ScalingUp/Procedure

[255] H. Rodriguez and I. Monga and A. Sadasivarao and S. Sayed and C. Guok and E. Pouyoul and C. Liou and T. Rosing, "Traffic Optimization in Multi-Layered WANs using SDN," in *Proceedings of IEEE Hot Interconnects*, 2014.

[256] S. Paul, R. Yates, D. Raychaudhuri, and J. Kurose, "The cache-and-forward network architecture for efficient mobile content delivery services in the future internet," *ITU T Kaleidoscope: Innov. in NGN*, 2008.

[257] "IP mobility support for IPv4," http://tools.ietf.org/html/rfc3344.

[258] "Mobility support in IPv6," http://www.ietf.org/rfc/rfc3775.txt.

[259] A. Lara, B. Ramamurthy, K. Nagaraja, A. Krishnamoorthy, and D. Raychaudhuri, "Using OpenFlow to Provide Cut-Through Switching in MobilityFirst," *Photonic Network Communications*, vol. 28, no. 2, pp. 165–177, 2014.

[260] Y. Wang, I. Avramopoulos, and J. Rexford, "Design for Configurability: Rethinking Interdomain Routing Policies from the Ground Up," *IEEE Journal on Selected Areas in Communications*, vol. 27, no. 3, pp. 336–348, April 2009.

[261] R. Doverspike, G. Clapp, P. Douyon, D. Freimuth, K. Gullapalli, B. Han, J. Hartley, A. Mahimkar, E. Mavrogiorgis, J. O'Connor, J. Pastor, K. K. Ramakrishnan, M. Rauch, M. Stadler, A. Von Lehmen, B. Wilson, and S. Woodward, "Using SDN Technology to Enable Cost-effective Bandwidth-on-demand for Cloud Services [Invited]," *IEEE/OSA Journal of Optical Communications and Networking*, vol. 7, no. 2, pp. A326–A334, February 2015.

[262] S. Van Dongen, "Graph Clustering Via a Discrete Uncoupling Process," *SIAM Journal on Matrix Analysis and Applications*, vol. 30, no. 1, pp. 121–141, 2008.

[263] S. Kent, C. Lynn, and K. Seo, "Secure Border Gateway Protocol (S-BGP)," *IEEE Journal on Selected Areas in Communications*, vol. 18, no. 4, pp. 582–592, 2000.

[264] A. Lara, B. Ramamurthy, K. Nagaraja, A. Krishnamoorthy, D. Raychaudhuri, "Cut-Through Switching Options in a MobilityFirst Network with OpenFlow," in *IEEE 7th International Conference on Advanced Networks and Telecommunication Systems (ANTS)*, December 2013.

[265] T. Vu, A. Baid, Y. Zhang, T. D. Nguyen, J. Fukuyama, R. P. Martin, and D. Raychaudhuri, "DMap: A Shared Hosting Scheme for Dynamic Identifier to Locator Mappings in the Global Internet," in *IEEE Distributed Computing Systems (ICDCS)*, June 2012.

[266] A. Sharma, X. Tie, H. Uppal, A. Venkataramani, D. Westbrook, and A. Yadav, "A Global Name Service for a Highly Mobile Internetwork," in *ACM SIGCOMM*, August 2014.

[267] N. Feamster, H. Balakrishnan, and J. Rexford, "Some Foundational Problems in Interdomain Routing," in *ACM Workshop on Hot Topics in Networks*, November 2004.

[268] H. Rodrigues, I. Monga, A. Sadasivarao, S. Syed, C. Guok, E. Pouyoul, C. Liou, and T. Rosing, "Traffic Optimization in Multi-Layered WANs using SDN," in *IEEE High-Performance Interconnects (HOTI)*, August 2014.

[269] A. Sadasivarao, S. Syed, P. Pan, C. Liou, A. Lake, C. Guok, and I. Monga, "Open transport switch: a software defined networking architecture for transport networks," in *Proceedings of ACM SIGCOMM Hot SDN*, August 2013.

[270] J. Pan, S. Paul, R. Jain, and M. Bowman, "MILSA: A Mobility and Multihoming Supporting Identifier Locator Split Architecture for Naming in the Next Generation Internet," in *Proceedings of IEEE GLOBECOM*, December 2008.

[271] Z. Gao, A. Venkataramani, J. F. Kurose, and S. Heimlicher, "Towards a Quantitative Comparison of Location-independent Network Architectures," in *ACM SIGCOMM*, August 2014.

[272] Z. Fu, P. Zerfos, H. Luo, and S. Lu, "The impact of multihop wireless channel on TCP throughput and loss," in *IEEE INFOCOM 2003*, March 2003.

[273] A. Anand, F. Dogar, D. Han, B. Li, H. Lim, M. Machado, W. Wu, A. Akella, D. G. Andersen, J. W. Byers *et al.*, "XIA: An Architecture for an Evolvable and Trustworthy Internet," in *ACM Workshop on Hot Topics in Networks*, November 2011.

[274] N. Basher, A. Mahanti, A. Mahanti, C. Williamson, and M. Arlitt, "A comparative analysis of web and peer-to-peer traffic," in *International World Wide Web Conference*, April 2008.

[275] J. Rexford, J. Wang, Z. Xiao, and Y. Zhang, "BGP routing stability of popular destinations," in *ACM SIGCOMM Workshop on Internet measurement*, 2002.

[276] F. Bronzino, K. Nagaraja, I. Seskar, and D. Raychaudhuri, "Network Service Abstractions for a Mobility-centric Future Internet Architecture," in *ACM international workshop on Mobility in the evolving internet architecture*, October 2013.

[277] R. Moskowitz, P. Nikander, P. Jokela, and T. Henderson, "Host Identity Protocol," https://tools.ietf.org/html/rfc5201, 2008.

[278] D. Farinacci, V. Fuller, D. Meyer, and D. Lewis, *The Locator/ID Separation Protocol (LISP)*, https://tools.ietf.org/html/rfc6830, 2013.

[279] Y. Hu, F. Zhang, K. K. Ramakrishnan, and D. Raychaudhuri, "GeoTopo: A PoP-level Topology Generator for Future Internet Study," WINALB Technical Report, Rutgers University, Tech. Rep., 2014.