An Evaluation of SDN Based Network Virtualization Techniques

by

Felipe Stall Rechia

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved April 2016 by the
Graduate Supervisory Committee:

Violet R. Syrotiuk, Chair
Gail-Joon Ahn
Dijiang Huang

ARIZONA STATE UNIVERSITY

May 2016

ABSTRACT

With the software-defined networking trend growing, several network virtualization controllers have been developed in recent years. These controllers, also called network hypervisors, attempt to manage physical SDN based networks so that multiple tenants can safely share the same forwarding plane hardware without risk of being affected by or affecting other tenants. However, many areas remain unexplored by current network hypervisor implementations. This thesis presents and evaluates some of the features offered by network hypervisors, such as full header space availability, isolation, and transparent traffic forwarding capabilities for tenants. Flow setup time and throughput are also measured and compared among different network hypervisors. Three different network hypervisors are evaluated: FlowVisor, VeRTIGO and OpenVirteX. These virtualization tools are assessed with experiments conducted on three different testbeds: an emulated Mininet scenario, a physical single-switch testbed, and also a remote GENI testbed. The results indicate that network hypervisors bring SDN flexibility to network virtualization, making it easier for network administrators to define with precision how the network is sliced and divided among tenants. This increased flexibility, however, may come with the cost of decreased performance, and also brings additional risks of interoperability due to a lack of standardization of virtualization methods.

TABLE OF CONTENTS

Page

LIST OF TABLES v

LIST OF FIGURES vi

CHAPTER

1 INTRODUCTION 1

2 LITERATURE REVIEW 5

2.1 Software Defined Networking and Network Virtualization 5

2.2 Virtual Local Area Network 9

2.3 Q-in-Q 11

2.4 MAC-in-MAC 13

2.5 Multiprotocol Label Switching 14

2.6 Virtual eXtensible Local Area Network 15

2.7 Virtual Private Networks 16

2.8 FlowVisor 18

2.9 VeRTIGO 23

2.10 OpenVirteX 25

2.11 FlowN 30

2.12 AutoSlice 31

2.13 AutoVFlow 32

2.14 Summary 32

3 EXPERIMENTS 33

3.1 Experimental Setup 33

3.1.1 Mininet Topology 34

3.1.2 Physical Topology 36

CHAPTER Page

3.1.3 GENI Topology 37
3.1.4 Floodlight Setup 38
3.1.5 FlowVisor Setup 38
3.1.6 VeRTIGO Setup 39
3.1.7 OVX Setup 40
3.1.8 Scapy 40
3.2 Functional Experiments 41
3.2.1 Network and Topology Isolation 42
3.2.2 Autonomous Rerouting 44
3.2.3 Transparent Traffic Forwarding 46
3.2.4 Compliance with Addressing Standards 48
3.3 Performance Experiments 50
3.3.1 Flow Setup Time 51
3.3.2 Throughput 53
3.4 Summary 54
4 RESULTS AND ANALYSIS 55
4.1 Functional Test Results 55
4.1.1 Network and Topology Isolation 56
4.1.2 Autonomous Rerouting 60
4.1.3 Transparent Traffic Forwarding 63
4.1.4 Compliance with Addressing Standards 65
4.1.4.1 Unicast IPv4 Header Rewriting 65
4.1.4.2 ARP and Multicast Header Rewriting 67
4.2 Performance Test Results 69

CHAPTER Page

4.2.1 Flow Setup Time 69
4.2.2 Throughput 72
4.3 Results Summary and Analysis 76
5 CONCLUSION 80
5.1 Future Work 82
5.1.1 Development of Network Hypervisors 82
5.1.2 IPv6 Support 83
5.1.3 Improved Experiments 83
REFERENCES 85
APPENDIX
A EXPERIMENTAL SETUP DETAILS 92

LIST OF TABLES

Table Page

1 Host Connectivity as Observed by Tenant Hosts in Mininet 59

2 Summarized Results of Traffic Forwarded with Network Virtualization. 63

LIST OF FIGURES

Figure Page

1 Simplified SDN Architecture. 6

2 Topology Discovery Using LLDP Flooding. 8

3 Network Hypervisor in an SDN Architecture. 8

4 Original Ethernet Frame and Different Encapsulation Options. 12

5 Use of Duplicate VLAN Tags Enabled by Q-In-Q. 12

6 MAC-In-MAC Frame. 13

7 The VXLAN Frame Format. 16

8 Network Slicing by FlowVisor. 19

9 FlowVisor Packet Flow Processing. 21

10 Physical Network at the Bottom and Corresponding Virtual Network. 24

11 OVX Simplified Architecture. 26

12 OVX Topology Virtualization through Discovery Manipulation. 27

13 OVX Packet Flow Processing. 29

14 Mininet Test Scenario. 35

15 Physical Test Scenario for Performance Tests. 37

16 GENI Test Scenario for Throughput Test. 38

17 Topology Isolation Expected with FlowVisor Virtualization. 43

18 Topology Isolation Expected with OVX and VeRTIGO Virtualization. 43

19 OVX Fast Reroute Feature. 45

20 Transparent Traffic Forwarding Test Reference. 47

21 Round Trip Time of the First Ping with Network Hypervisor. 51

22 Network A Discovered by Floodlight Instance A Using FlowVisor as the Network Hypervisor. 56

Figure Page

23 Network B Discovered by Floodlight B through FlowVisor. 57

24 Network C Discovered by Floodlight C through FlowVisor. 57

25 Network A Discovered by Floodlight Instance A Using OVX as the Network Hypervisor. 58

26 Network B Discovered by Floodlight B through OVX. 58

27 Network C Discovered by Floodlight C through OVX. 58

28 IP Header Rewriting Done by OVX. 66

29 IP Header Rewriting Done by OVX with Same IP Addresses. 66

30 ARP, IP and MAC Multicast Flooded by OVX. 68

31 Round Trip Time of First Ping. 70

32 Average round Trip Time of subsequent Pings. 71

33 Throughput Measured with Different Controllers, in Bits per Second. 72

34 Throughput Results in GENI Testbed, in Bits per Second 74

35 Throughput Results in GENI Testbed, for OVX, in Bits per Second 75

Chapter 1

INTRODUCTION

The virtualization of computers is becoming increasingly popular and widespread in the context of cloud computing. Users can set up their own private virtual machines hosted by a service provider in a matter of minutes, using services such as Amazon EC2, Microsoft Azure, or Google's Cloud Platform. Virtualization and isolation of the CPU, memory and storage elements in these environments is straightforward and highly automated. However, dynamic configuration, virtualization, and isolation of network resources is still considered a "missing link that will interconnect all other virtualized appliances" [11].

Chowdhury and Boutaba [11] define network virtualization as a service that provides several multiple logical networks decoupled from one single physical network, allowing different users to share the same physical network resources while still being entirely isolated from one another. In order to understand network virtualization it is important to have a clear definition of what are the resources offered by computer networks [33], [60]:

- The topology, which is comprised of the set of network devices – that is, switches and routers, and their respective ports and links;
- The address space, which is determined by the kind of addressing used in the network, such as IP;
- The network devices' resources, namely the CPU, memory, flow tables, etc.;
- The bandwidth of links.

The concept of network virtualization is not new. Virtual local area networks

(VLANs), virtual private networks (VPNs), and several other techniques allow several *tenants* (or *customers*) – sets of users of a network infrastructure – to share the same network hardware without interfering with other tenants. However, these traditional virtualization techniques rely on a distributed network control plane, and, thus, frequently cannot split network resources evenly among tenants. Furthermore, the provisioning of traditional virtual networks is made on a box-by-box basis. Without centralized management, the manual network configuration process can take months [33].

In the computing domain, hypervisors are regarded as the controlling entities that enable virtual appliances to share abstracted computing and storage resources of a host machine among several tenants. What hypervisors assign to one tenant cannot be used by another [33]. The same requirements have been addressed in the networking domain by the development of *network hypervisors*. Their purpose is analogous – to allow several hosts to share network resources while minimizing the risk of having one instance interfering with another [11].

Network hypervisors leverage software defined networking (SDN) concepts to implement network virtualization. SDN decouples the control and data forwarding planes of a network, and allows the network devices to be controlled by a single centralized controller. This enables quick provisioning and fair sharing of network resources among tenants [33]. Additionally, the network wide view provided by SDN gives the means to explore new research areas and develop innovative network applications.

Several different SDN-based network hypervisors have been developed in recent years, such as FlowVisor [60], OpenVirteX (OVX) [57] and VeRTIGO [15]. These network hypervisors aim to provide isolation between network tenants, and in some

cases they go further and offer additional features. OVX makes the full IP header space available to all tenants through use of a complex header rewriting technique [57]. Both OVX and VeRTIGO propose automatic handling of network failure and/or congestion [15], [57].

However, network hypervisors often neglect to evaluate basic network functions. Are network hypervisors capable of forwarding any kind of Ethernet or IP protocols without blocking tenant traffic? Do the header rewriting techniques employed by OVX comply with current IP standards?

In an attempt to answer these questions and many more, this thesis evaluates network hypervisors with both qualitative and quantitative experimental approaches. Functional experiments are conducted to verify network topology isolation, resiliency, transparent traffic forwarding, and compliance with addressing standards. Performance experiments are also conducted to measure the impacts of introducing a network virtualization layer between the control and data planes of the network. Flow setup time and throughput are measured and compared to that of non-virtualized networks.

The results show that network hypervisors enable efficient sharing of network resources while keeping tenants isolated. However, they still have problems handling IPv6 or multicast traffic, and in some cases the virtualization layer introduces great performance impacts to the network.

Chapter 2 presents both the traditional and the SDN network virtualization technologies. The purpose is to give the reader a background on how virtualization is done in traditional distributed networks and on how it can be done with SDN. Chapter 3 introduces and explains the experimental scenarios, procedures, and methods used to evaluate three network hypervisors. The results of the experiments are analyzed and discussed in Chapter 4. Conclusions are drawn in Chapter 5, outlining the

major differences between the traditional and SDN-based virtualization techniques and discussing advantages and disadvantages of each. Future work possibilities are also presented in Chapter 5.

Chapter 2

LITERATURE REVIEW

Recent works about network virtualization tend to ignore fully-developed virtualization techniques, such as VLANs and VPNs, in favor of new software defined networking (SDN) techniques. However, these traditional methods are already fully standardized and implemented in vendors' equipment, so they are prepared to deal with many real world problems faced by network users today.

This chapter introduces background concepts about SDN and explains how it differs from traditional networking concepts. It then presents a brief explanation of how some of the traditional network virtualization standards and technologies work, followed by an introduction to recent SDN-based network virtualization tools, the network hypervisors.

2.1 Software Defined Networking and Network Virtualization

Traditional networks are distributed systems composed of network devices that are responsible for handling both the *control plane* and the *data forwarding plane* of the network. The control plane function decides how the network traffic should be handled; it is responsible for discovering neighboring devices and networks, and deciding where traffic should be forwarded. Topology discovery, spanning-tree, and routing protocols are examples of control plane functions [33]. The data forwarding plane only consults forwarding tables that were previously calculated by the control plane, and uses this information to quickly forward traffic [33].

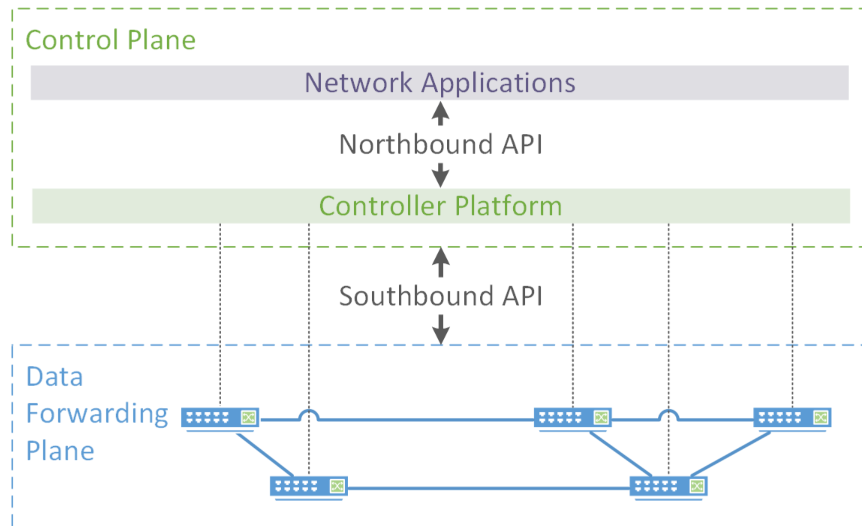

Figure 1. Simplified SDN architecture.

SDN is a new paradigm which aims to take control plane functions out of the network devices and place them in a specialized, logically-centralized controller of the network. This has the benefit of simplifying network configuration and policy enforcement. It also reduces the system resources required from the network devices, as they are no longer required to run control plane functions [33].

In an SDN architecture, the point of reference is usually the centralized *controller platform*, depicted at the center of Figure 1 [33]. *Network applications* are program modules that communicate with the controller platform through a *northbound application programming interface* (API). Typical network applications are MAC learning, load balancers, and routing algorithms. The northbound APIs are not very well standardized and there are many options to choose from [33]. Therefore, the controller platform and the network applications are often integrated into a single piece of software which is called the *controller*. POX [50], Floodlight [19] and Beacon [17] are examples of controllers which offer a standard MAC learning application. The data forwarding plane is composed of simple *forwarding devices*, usually called switches,

which communicate with the controller through the *southbound API*. The OpenFlow protocol [40] is the most well known, standardized and actively used southbound API for SDN [33].

OpenFlow allows the controller to manipulate the traffic in the data forwarding plane through the installation of *flow rules* in the OpenFlow *flow tables* of the switches. Each flow rule is composed of *match* and *action* pairs. For example, a packet can be matched by destination MAC address, source IP address, and/or many other common header fields. If packets match the flow rule, then they can be forwarded to the controller, dropped, have their headers rewritten, flooded out of all switch ports, etc. [33].

A key function for network virtualization in SDN is *topology discovery*. The controller needs to know the data forwarding plane topology – how the network devices are interconnected physically – in order to successfully implement traffic forwarding. Typically OpenFlow controllers discover the data forwarding plane by first sending OpenFlow PACKET OUT messages asking network devices to flood link layer discovery protocol (LLDP) frames out of all ports. The controller then waits for OpenFlow PACKET IN messages carrying LLDP messages. The PACKET IN messages contain information about which port received which LLDP messages, thereby allowing the controller to build a graph of the network. This process is explained by Pakzad *et al.* [47] and depicted in Figure 2. The controller fully discovers the topology by repeating this process for all network devices.

Device identification also plays an important role in SDN-based network virtualization. Each OpenFlow capable switch in a network is uniquely identified by a datapath ID (DPID), a 64-bit field composed of 48 bits of the network device's real unique MAC address plus 16 bits that are left as an additional identification field. Vendors

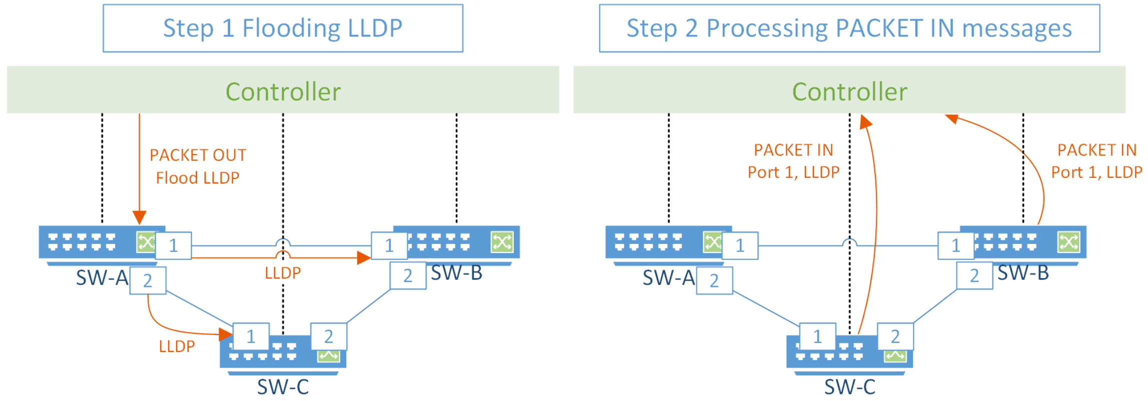

Figure 2. Topology discovery using LLDP flooding.

can use this additional 16-bit field in any way they desire [8]. For instance, in the Hewlett-Packard switch used in this work, each OpenFlow instance is associated with a VLAN number in the switch. The extra 16-bit field in the DPID is used to carry the VLAN number which corresponds to an OpenFlow instance. This allows easy distinction of which ports of the switch can be controlled through a particular DPID – any ports belonging to the given VLAN. This concept is called slicing of resources and will be further explained in Section 2.8.

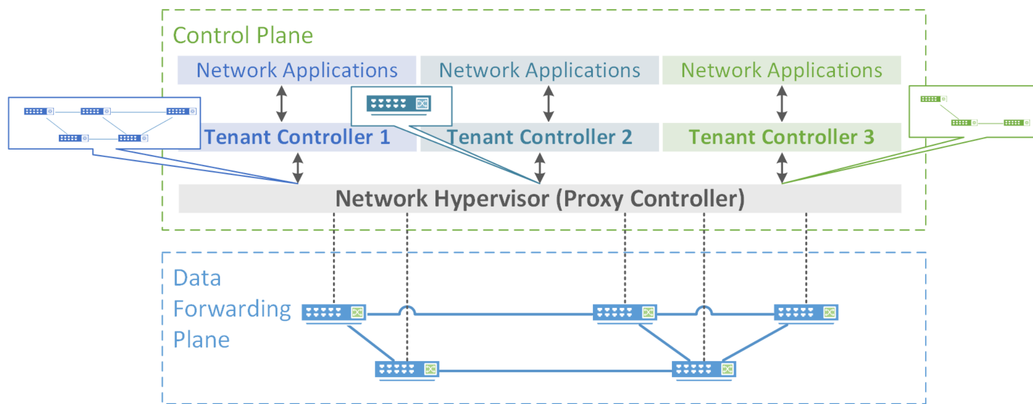

Figure 3. Network hypervisor in an SDN architecture.

SDN-based virtualization is usually implemented with the introduction of *proxy*

controllers that act as **network hypervisors** between the data forwarding plane and the many *tenant controllers* that are sharing control over the same infrastructure. As depicted in Figure 3, network hypervisors can change how the tenant controllers see the network. Tenant controller 1 has control over all the network devices transparently; Tenant controller 2 sees an abstraction of the whole network represented by a single big switch; Tenant controller 3 sees only a portion of the network containing three of the five switches.

Examples of network hypervisors that work using the proxy controller principle are: FlowVisor [60], OpenVirteX [57], VeRTIGO [15], AutoVFlow [66] and AutoSlice [9]. FlowN [16] is an exception and does not use the proxy architecture, instead being a modified controller which spawns containers of the network applications for virtualization. More details about each of these SDN virtualization approaches will be given in later sections of this chapter.

It is important to first understand at least some of the most basic traditional virtualization technologies before further exploring SDN-based network virtualization. The traditional methods, some of which will be briefly covered in the next few sections, can be employed to further enhance the novel methods implemented with SDN.

2.2 Virtual Local Area Network

The virtual local area network (VLAN) technology is currently standardized by the IEEE standard 802.1Q-2014 document [24]. VLANs provide traffic isolation and a reduction of broadcast domains within LANs, and it is one of the most basic forms of network virtualization. Basic functionality is realized by the introduction of a VLAN tag (802.1Q tag) in the middle of Ethernet frames. For example, switches that support

VLANs do not allow frames tagged with a certain VLAN tag X to be forwarded to ports which belong only to VLAN Y. The fields in a VLAN enabled Ethernet II Layer 2 frame are [24]:

- MAC destination (6 bytes)
- MAC source (6 bytes)
- **VLAN tag** (4 bytes)
- Ethertype (2 bytes)
- Payload (variable length)
- CRC check (4 bytes)

The 4 bytes of the VLAN tag are divided in the following ways [24]:

- Tag Protocol Identifier (TPID, 2 bytes)
  - 0x8100 for basic VLAN functionality.
  - 0x88a8 for backbone component addressing (Q-in-Q).
  - 0x88e7 for service encapsulation (MAC-in-MAC).
- Tag Control Information (TCI, 2 bytes)
  - Priority Code Point (PCP) – a 3-bit field that can be used to convey priority information about the current frame.
  - Drop Eligible Indicator (DEI) – a 1-bit field that can be used to indicate whether this frame is eligible to be dropped in case of congestion.
  - VLAN Identifier (VID) – a 12-bit field used to identify to which VLAN this frame belongs.

Additionally, the use of VLAN tags also allows traffic to be marked with different priorities. For instance, time sensitive frames generated by an IP phone could have its

PCP field configured to a higher value, indicating that this frame should be forwarded with greater priority in a network.

Some problems arise from the use of frames with a single VLAN tag. For instance, only 2^{12} virtual networks may be created, a number which can be easily reached by current data center networks with thousands of virtual machines. Furthermore, each customer may only use a subset of the 2^{12} VIDs, so administration of which VIDs are available to which customers can become troublesome. This problem is partially addressed by Q-in-Q, explained in the next section.

2.3 Q-in-Q

The use of two VLAN tags per frame allows customers to employ the full VLAN header space, overcoming the global limit of VIDs. Using two VLAN tags in the same frame is usually called Q-in-Q by network equipment vendors [31], and was originally specified by IEEE standard 802.1ad-2005 [26]. The standard describes the isolation of multiple customer networks within a single provider by assigning one VID to each customer in the first VLAN tag of the frame, and the customer is free to use any of the 2^{12} VIDs in the second VLAN tag. The name Q-in-Q is a reference to the fact that a 802.1Q tag is used again within an 802.1Q frame. The original 802.1ad standard was later incorporated into the IEEE standard 802.1Q-2014 [24].

Figure 4 [38] shows different types of frames for illustration and comparison of the different tagging and encapsulation options possible with VLANs – a pure Ethernet frame, a frame with a with single VLAN tag – allowing 2^{12} different VIDs – and a frame with two VLAN tags – allowing $2^{12} \times 2^{12} = 2^{24}$ VID combinations. However, the

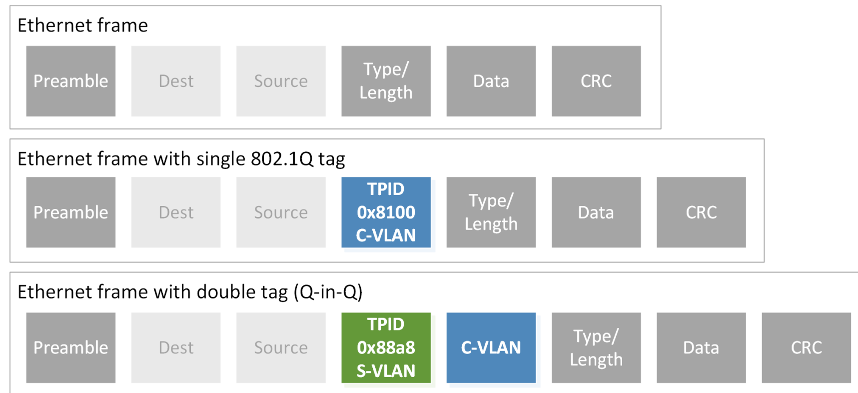

Figure 4. Original Ethernet frame and different encapsulation options.

customer's MAC address is always exposed in these frames, regardless of the number of VLAN tags used for the payload.

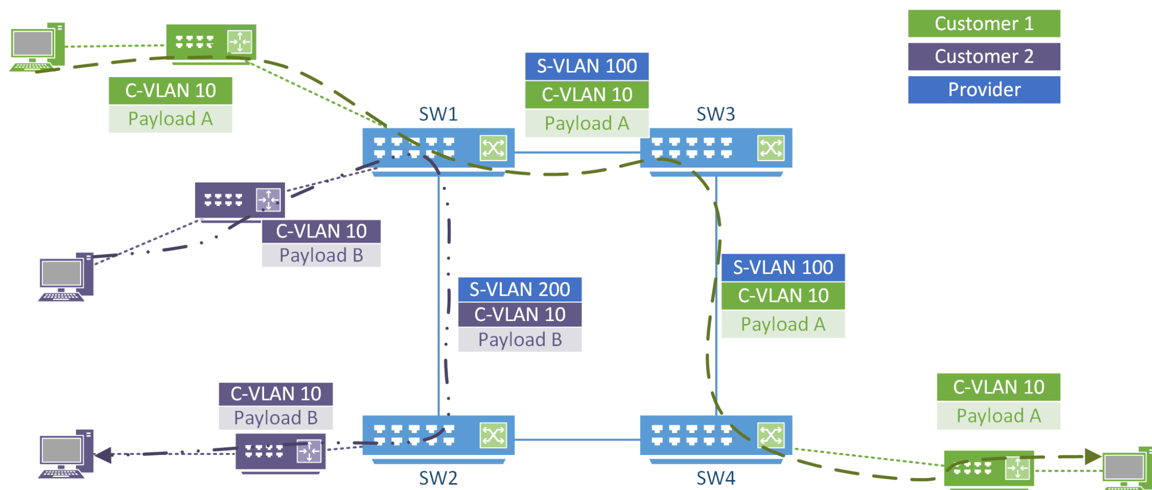

Figure 5. Use of duplicate VLAN tags enabled by Q-in-Q.

In Figure 5 [38], an example of Q-in-Q usage shows two different customers that can use the same VLAN tag – the *customer VLAN* (C-VLAN) tag number 10 – and remain isolated from one another while sharing the same network infrastructure. This works because the provider supplies a different *service VLAN* (S-VLAN) tag for each customer. For customer 1 the provider encapsulates all traffic with S-VLAN tag 100,

while, customer's 2 traffic is encapsulated with S-VLAN tag 200. Q-in-Q allows both customers to have 2^{12} VIDs available. However, it still does not provide complete separation between customer and provider domains due to the following possible problems [7]:

- Customers MAC addresses travel through the entire provider backbone and are learned in every switch along the way. This affects the scalability of the service as the provider's MAC tables may easily become full;
- The number of service VLAN tags is still limited to 2^{12};
- There is no clear demarcation point between customer and provider networks regarding fault and performance management.

MAC-in-MAC can be used to overcome these problems, and it will be explained in the next section.

2.4 MAC-in-MAC

The provider backbone bridges (PBB) functionality is typically called "MAC-in-MAC" by networking equipment vendors and extends the concept of Q-in-Q by allowing complete encapsulation of the customer's traffic, including the customer MAC address (C-MAC). The IEEE 802.1ah-2008 standard [27] introduced the concept of PBB, which was later incorporated into the IEEE Std 802.1Q-2014 standard [24].

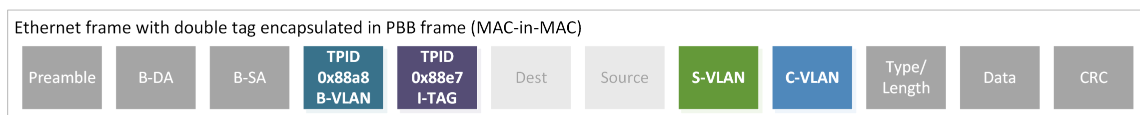

Figure 6. MAC-in-MAC frame.

Figure 6 shows the MAC-in-MAC frame, in which there are backbone destination and source addresses (B-DA and B-SA, respectively). Figure 6 shows that in the PBB frame the customer MAC addresses (in lighter gray color) cannot be learned by switches in the provider backbone. This becomes clear when comparing the PBB frame to the single and double tagged frames of Figure 4. The backbone VLAN (B-VLAN) represents a VLAN in the backbone, which is independent from the other VLAN tags in the customer frame. The backbone service instance tag (I-TAG) contains a 24-bit field called the backbone service instance identifier (I-SID) that is used to identify a unique customer within the provider backbone bridged network. This allows 2^{24} different customers to use 2^{24} different combinations of VLAN tags each [7].

Therefore, the MAC-in-MAC solution addresses the main problems which occur with Q-in-Q by:

- Providing more identifiers for different network customers (2^{24});
- Preventing switches in the backbone from learning all the customer MAC addresses, learning only the backbone destination and source addresses; and,
- Introducing a clear demarcation point between customer and provider networks.

2.5 Multiprotocol Label Switching

The multiprotocol label switching (MPLS) architecture introduces the concept of partitioning sets of packets into forwarding equivalence classes (FECs) and mapping each to a specific set of hops in the network. The FECs are then mapped into labels, or label switched paths (LSPs) across the network. Labels can then be easily used by label switching routers (LSRs) to make the forwarding decisions, because the network layer is analyzed just once – at the network ingress router [54].

The paths that FECs take in the network are usually assigned by routing algorithms that already run in the network, such as shortest path first. The mapping of LSPs to FECs is then typically performed by the label distribution protocol (LDP) [4] or resource reservation protocol (RSVP) with extensions for LSP tunnels [6] that allows different LSRs to negotiate the meaning of labels and how to forward them across the network.

Among other things, MPLS enables a packet-switched network to operate almost as a circuit-switched network. Many network services are based on MPLS, such as virtual pseudo wire services (VPWS) and virtual private LAN services (VPLS) [5].

2.6 Virtual eXtensible Local Area Network

Virtual eXtensible local area network (VXLAN) is yet another framework that was created to solve the problem of network virtualization, although specifically oriented to data center networks [35].

The main purpose of VXLAN is to solve a few major issues that arise from grouping a large amount of tenants and VMs in a single shared layer 2 network [35]. These include:

- Using network infrastructure shared by potentially hundreds of thousands of VMs;
- Limitations of VLAN tags while still dealing with possible duplicate assignments of MAC addresses and VLAN IDs;
- Preferred use of IP networks and routing to take advantage of equal cost multipath (ECMP) and improve network utilization.

The VXLAN framework solves these issues by building a layer 2 network overlay

on top of a layer 3 network, therefore “stretching” layer 2 segments to reach remote networks. VXLANs are identified by a 24-bit segment ID called a VXLAN network identifier (VNI) [35].

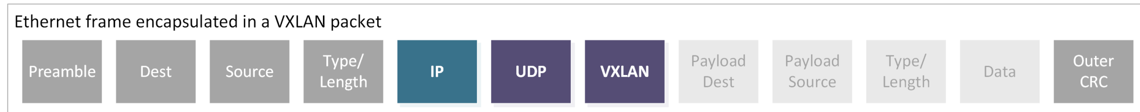

Figure 7. The VXLAN frame format.

The tenant’s frames are encapsulated with a VXLAN header and a user datagram protocol (UDP) which can then be forwarded through any IP network. The UDP destination port is always 4789, and the Ethernet CRC check is only preserved for the outer frame. Figure 7 shows the frame format for VXLAN [35]. The tenant’s payload, in lighter gray color, is completely isolated from the provider’s network because it stays in the application layer.

2.7 Virtual Private Networks

Virtual private networks (VPNs) achieve virtualization in such a way that the network client sees typically just a portion of the whole network, which could be the Internet or any other shared network infrastructure. Only participating users can send and receive traffic within the VPN [30], [34].

Although VPNs are often considered network virtualization with added security mechanisms, such as encryption and authentication, there is no precise definition of what a VPN needs to have. In some cases encryption and authentication are not used at all (e.g., the network provider is trusted in a MPLS VPLS VPN). Lewis [34] and

Jaha *et al.* [30] attempted to classify some of the existing types of VPN in several different ways. A few of these criteria are summarized in the following paragraphs.

In *trusted VPNs*, the client trusts that the provider's network is secure and not accessible by the public. No authentication or encryption is necessary [30], [34]. Examples of trusted VPNs are MPLS-based layer 2 VPNs such as virtual pseudo wire service (VPWS), and virtual private LAN service (VPLS) [5]. In *secure VPNs*, client data must be authenticated and encrypted over the provider's network [30], [34]. Examples of trusted VPNs are application based secure socket layer (SSL) (e.g., OpenVPN [42]), and Internet protocol security (IPSec) VPNs [32].

Connection-oriented VPNs use virtual circuits or tunnels to transport data [30], [34]. For instance, MPLS layer 2 VPN or generic routing encapsulation (GRE) are connection-oriented VPNs [18]. *Connectionless VPNs* rely on partitioning of client data at the provider edge (PE) [30], [34]. Using VLAN tags to achieve layer 2 connectivity between two customer sites can be considered connectionless, as it basically relies on pure Ethernet for forwarding [23].

Overlay VPNs imply that the client is not aware of the network topology used for the VPN because it does not exchange routing information with the provider. Overlay VPNs include MPLS layer 2 VPN, GRE, or IPSec tunnels [30], [34]. *Peer VPNs* require the client to exchange routing information with the provider, such as in a virtual routing and forwarding (VRF) VPN [34].

Provider provisioned VPNs must be configured and deployed entirely by the network provider. This allows the provider to control more precisely how customer traffic is handled. VPWS, VPLS and VRF are all provider provisioned VPNs [30], [34]. *Customer provisioned VPNs* are configured and deployed entirely by the network

client, and the provider may not even be aware of the existence of these VPNs. GRE and application based SSL are both customer provisioned VPNs [30], [34].

VPNs can provide the same virtualization benefits that were discussed in Chapter 1, topology, address space and resource isolation. The next few sections will introduce the SDN approach to network virtualization – network hypervisors.

2.8 FlowVisor

FlowVisor [60] is one of the first SDN network virtualization approaches to be implemented based on OpenFlow. It consists of a transparent proxy that acts between the OpenFlow controller and switches, allowing multiple controllers to share the same network infrastructure. Sherwood *et al.* [60] suggest the division of network resources in five dimensions: bandwidth, topology, traffic, device CPU, and forwarding tables. The network is then divided in *slices*, each of which contain a subset of these resources, and is controllable only by the owner of that slice.

To isolate the topology, FlowVisor may restrict the ports the tenant controllers see from the real physical topology, to hide restricted areas of the network from the users [60]. Figure 8 shows an example in which only Tenant 1 sees switches SW1 and SW2 from the network, and Tenant 2 sees only a fraction of the ports from switches SW3 and SW4.

Flow space isolation is implemented by rewriting the flow rules sent from the tenant controller to the switch, restricting the flow space they affect. For instance, if a controller attempts to create a rule restricting all traffic and FlowVisor knows that this controller has access only to install rules for TCP port 80, then the OpenFlow

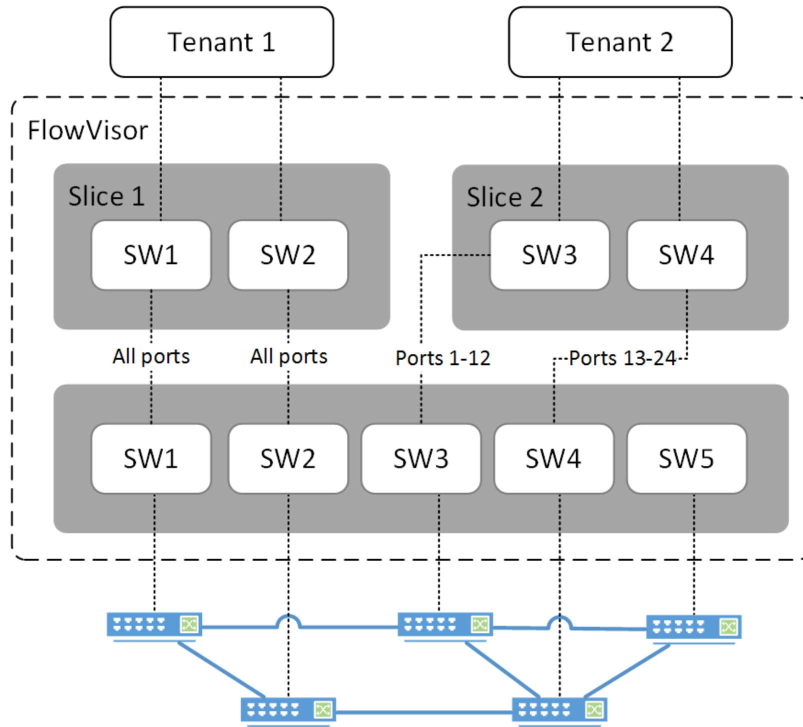

Figure 8. Network slicing by FlowVisor.

rule is rewritten in FlowVisor before being sent to the switches and affect only TCP port 80 traffic [60].

To protect the network devices' CPU, FlowVisor acts by rate limiting the number of messages exchanged between tenant controllers and switches. If there are too many table-miss OpenFlow PACKET IN messages coming from a network device to a controller, FlowVisor also acts by installing a temporary drop rule in the corresponding network device while forwarding the first table-miss PACKET IN to the controller. This protects not only the CPU of the network device, but also the CPU of the controller, as it gives time for the controller to process the new table-miss PACKET IN request without being disturbed by further repeated PACKET IN messages coming from the same network device [60].

FlowVisor also provides protection of the flow tables of the switches, to ensure

that one single controller does not exhaust all the flow entries in a device. This is achieved by counting the flow rules installed by each tenant controller and limiting the number of authorized flow rules to a predefined value [60].

To isolate network bandwidth, OpenFlow does not support a direct way of controlling bandwidth or QoS. Therefore, it may employ the use of different VLAN tags configured with different PCP bits for this purpose. Although this technique relies on at least some degree of manual queue configuration in the switch, FlowVisor may rewrite flows with new VLAN PCP bits to adjust the amount of bandwidth allowed for installed flow rules [60].

Even though FlowVisor is still not perfectly capable of isolating the five network dimensions, the SDN concept allows it to divide and monitor the network in ways that were not possible before. FlowVisor experiments show that it leverages SDN concepts to introduce centralized policy enforcement, that is, FlowVisor has a global point of view of the network and may drop or rewrite OpenFlow messages according to configured policies [59]. Furthermore, FlowVisor has the ability to slice the flow space with more flexibility than the traditional coarse-grained traditional network virtualization techniques. For instance, VLANs allow the traffic to be sliced only "by input port or explicit tag", while FlowVisor's slicing mechanism can split the network by ports, protocol, address ranges, etc. [59].

Several experiments demonstrate FlowVisor's ability to run an experiment in a slice, side-by-side with other experiments or even production traffic in a network, with guaranteed isolation among slices, fine-grained network control, and still providing hardware forwarding speeds to each slice [59].

Figure 9 shows an example of how FlowVisor interacts with the switches and the tenant controllers when A1 starts an attempt to communicate with A2 by sending an

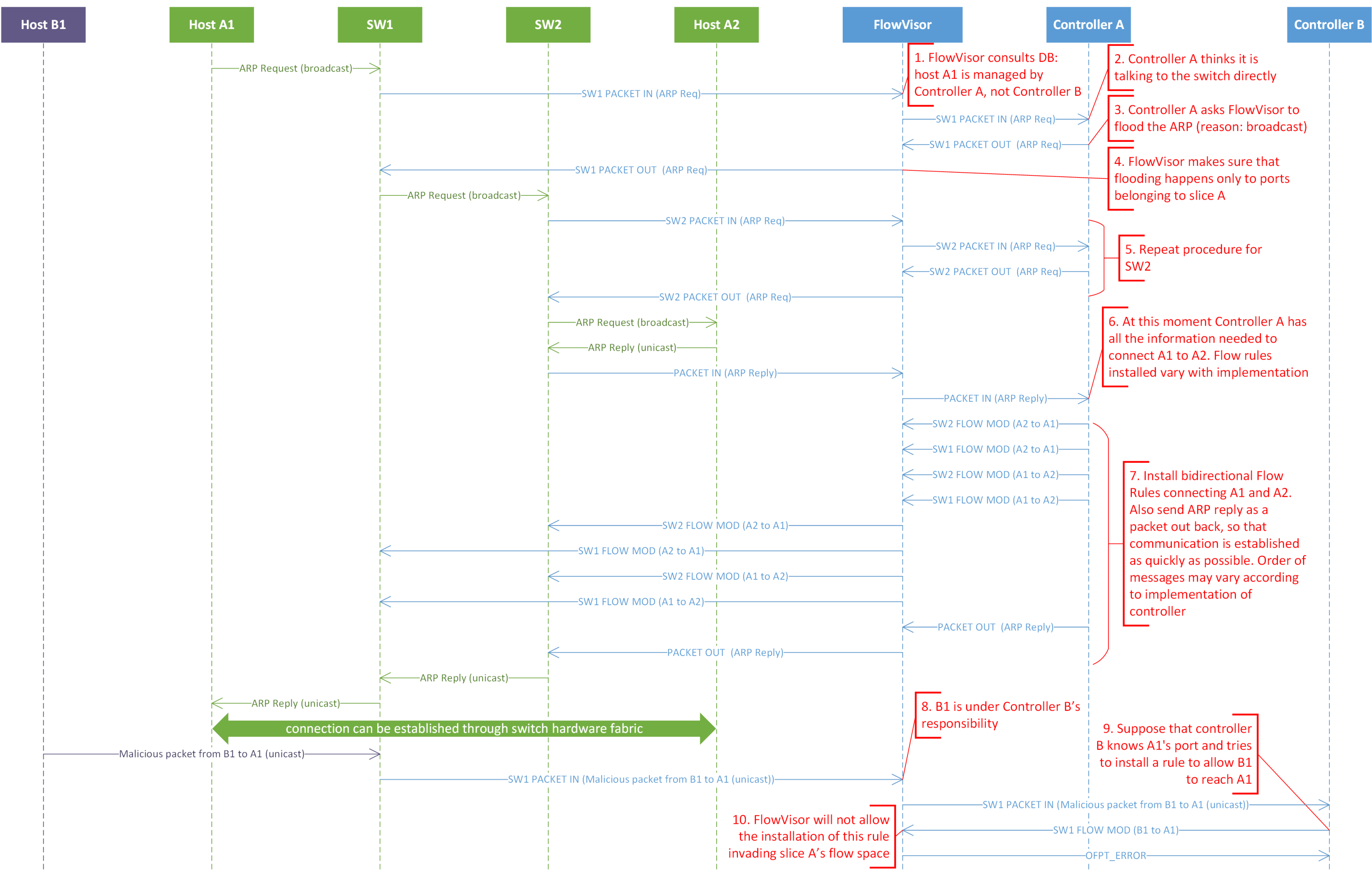

Figure 9. FlowVisor packet flow processing.

ARP Request. SW1 forwards this request to FlowVisor inside an OpenFlow PACKET IN message. In Step 1, FlowVisor analyzes the PACKET IN message and forwards this ARP Request to controller A (Step 2), because host A1's MAC address belongs to slice A flow space (this is configured by the network administrator in FlowVisor's database). The controller, running a simple MAC Learning application, sends back to FlowVisor a PACKET OUT flood with the ARP Request (Step 3). If necessary, FlowVisor rewrites the PACKET OUT message before sending it back to the network (Step 4), ensuring that the ARP Request is flooded only to members of slice A. SW1 received the PACKET OUT, forwards the ARP request to SW2, which, in turn, repeats the same Steps 1 through 4 to forward the ARP Request to host A (Step 5). A2 receives the ARP request, replies to it, and the reverse process of reaching A1 is started. When the controller A gets the PACKET IN ARP Reply message from A2, the controller has enough information to instruct the switches to start forwarding traffic between A1 and A2 independently (Step 6). In Step 7, the controller A sends FLOW MOD messages to FlowVisor. FlowVisor checks if these messages are not violating any flow spaces not belonging to controller A, and then programs SW1 and SW2 with the new Flow Rules. Additionally, the controller A finishes sending the PACKET OUT ARP Reply back to SW1 to make sure that the address resolution phase finishes. The programming of SW1 and SW2 hardware fabric is done, and hosts A1 and A2 can communicate through the data plane without further need from the control plane. Steps 8-10 show how FlowVisor would handle a case when both a host and a tenant controller try to "invade" part of the flow space which does not belong to their slice. FlowVisor blocks attempts from controller B to install flows in controller A's slice.

2.9 VeRTIGO

VeRTIGO [15] extends FlowVisor with the concept of *abstract nodes*, or abstract network devices, which consist of two basic building blocks: *virtual links* and *virtual ports*. Virtual links bundle together several physical nodes and links and abstract them to the tenant controller as a single link between any two ports. One or more virtual ports are mapped to a physical port according to the number of virtual links that need to use the same physical port. Figure 10 shows how VeRTIGO would use virtual links 1 and 2 between SW-A and SW-D to create redundant connections between physical ports A and B. The four physical switches are represented as a single abstract node to the tenant controller, with virtual ports X and Y mapped to physical ports A and B, respectively.

VeRTIGO is built on top of FlowVisor, so it automatically provides all of the network slicing features offered by FlowVisor. New modules that were designed specifically for VeRTIGO are briefly described here.

The *classifier* module determines which messages coming from the network are handled by the tenant OpenFlow controllers. Some of the messages are directly handled by VeRTIGO's *internal controller*, so that it can hide details of the network from tenant controllers, which only control a portion exposed to them. For instance, in Figure 10, OpenFlow messages generated by traffic entering SW-A from virtual port X are handled by the tenant controller, while any OpenFlow messages related to traffic between SW-B and SW-D – an internal portion of the virtual network, hidden by VeRTIGO – are handled and routed by the internal controller [15].

A *node virtualizer* module is responsible for grouping multiple physical nodes in the network and making them seem as only one abstract node for the tenant controller,

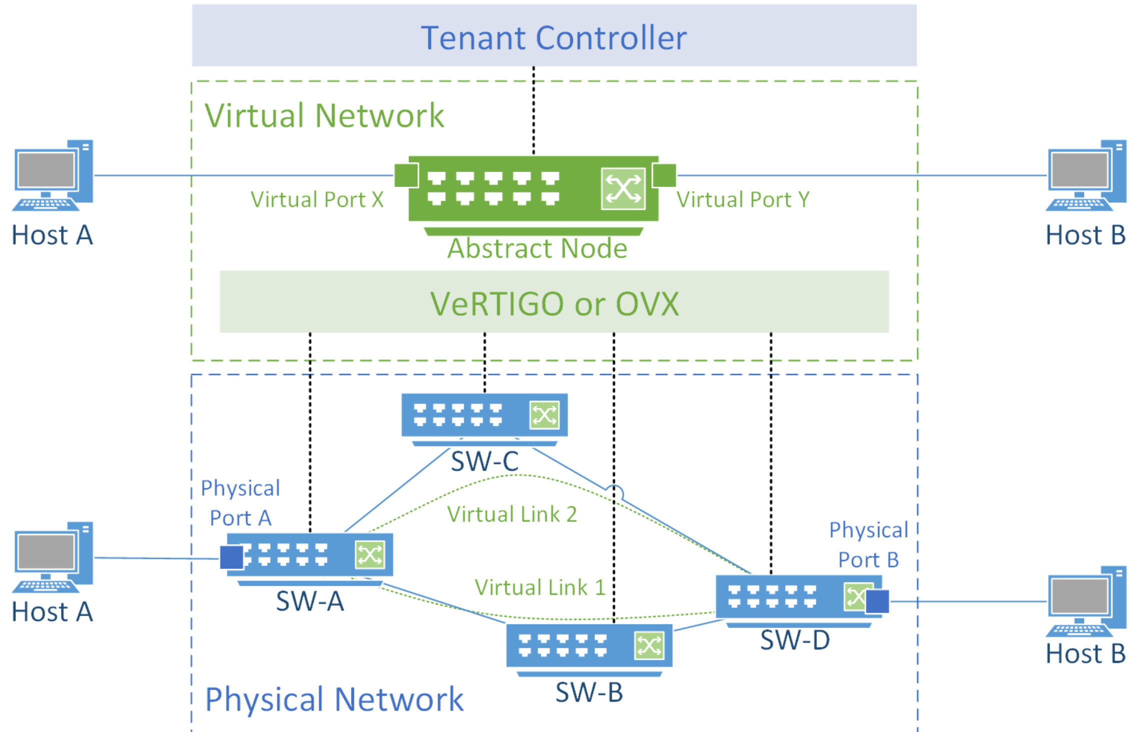

Figure 10. Physical network at the bottom and corresponding virtual network.

as depicted in Figure 10. VeRTIGO multiplexes the messages from several physical nodes in the network and maps real data path IDs (DPIDs) to virtual DPIDs [15] of the abstract node.

When a single physical port is in use by many different virtual links, the *port mapper* handles the mapping of virtual to physical ports. This also occurs for abstract node ports, which need to have physical port numbers remapped to virtual port numbers [15].

The *virtual topology (VT) planner* is in charge of associating virtual network instances with real network resources. For instance, when there are multiple paths to connect points A and B, this module is responsible for finding the best path inside an abstract node. The best path depends on application requirements and may be chosen based on available throughput or total latency. The VT planner monitors network

statistics to determine current throughput and latency to judge which is the best path for the current application.

2.10 OpenVirteX

OpenVirteX [57] (OVX) brings the virtualization of networks one step closer to the infrastructure as a service (IaaS) concept. OVX provides a framework which makes it possible for tenant controllers to instantiate, snapshot, migrate or delete virtual networks, which is analogous to hypervisors handling virtual machines (VMs) in cloud computing environments. Similar to FlowVisor, OVX stands as a proxy between the network operating system (NOS) and the OpenFlow capable network. The main improvements claimed by OVX are that it provides:

- A complete custom address space (or flow space) to each of the network slices created with no risk of overlapping addresses;
- A fully virtualized network topology that can be specified by the slice tenant.

The sample virtualized *big switch* network shown in Figure 10 can also be realized with OVX, with the difference that OVX can provide an almost full MAC and IPv4 header space to each of its tenants. This allow the tenants to use potentially overlapping addresses. This is accomplished by using header rewriting techniques at the edges of the network, rewriting the MAC or IP addresses in the packet headers with additional information including globally unique tenant IDs. If the tenant controller pushes layer 2 flow rules to the network devices, then OVX resorts to rewriting the MAC header. If layer 3 flow rules are installed, OVX uses IP rewriting [57].

Al-Shabibi *et al.* [57] claim that the link virtualization could also be implemented

by MPLS labels instead of header rewriting, though this is not done in the current OVX version.

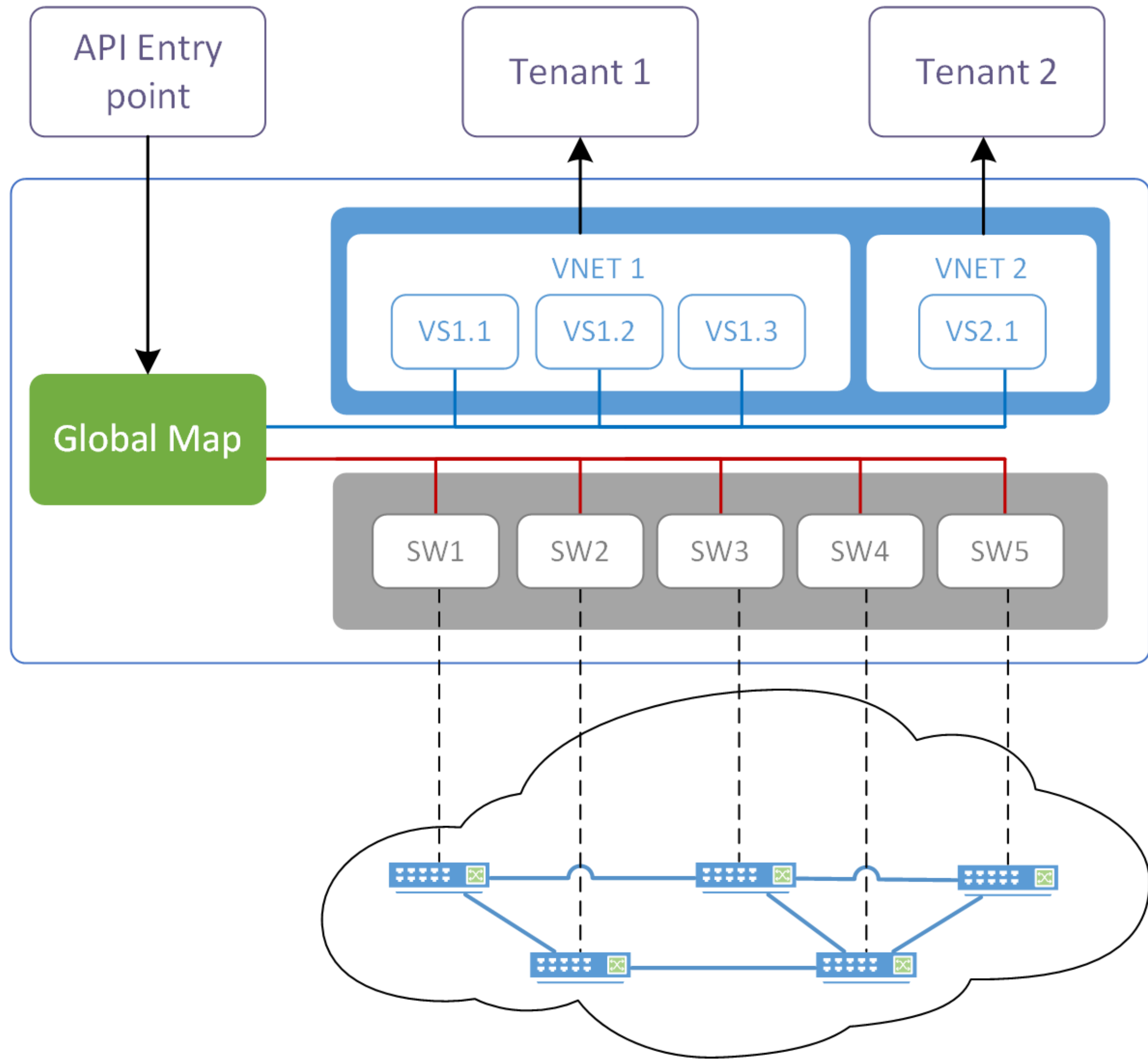

Figure 11. OVX simplified architecture.

OVX works by keeping two separate logical representations of the network as seen in Figure 11 [45]. One is the physical network, illustrated by the switches inside the cloud, and their virtual counterparts stored in OVX's database – SW1-SW5 in the gray box. The virtual networks are represented by the blue boxes on the upper portion

of the architecture. Both the physical and virtual representations of the network are stored in a *global map*, represented by the green box in Figure 11, which currently uses MongoDB as the database backend [46].

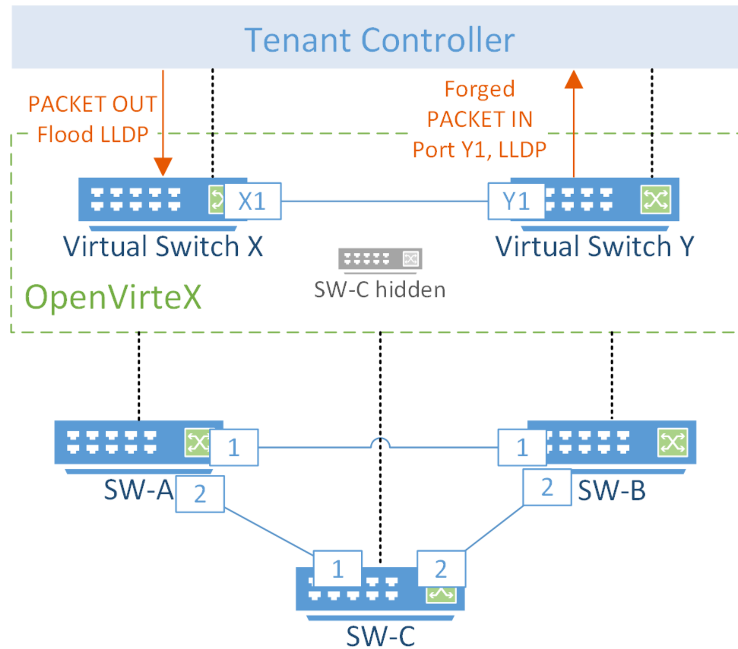

Figure 12. OVX topology virtualization through discovery manipulation.

To enable the representation of different virtual topologies, OVX intercepts LLDP messages coming from the tenant controllers and creates LLDP responses designed to represent the underlying physical network topology in different ways. This topology discovery manipulation is depicted in Figure 12, where virtual switch X is mapped to switch A, virtual switch Y is mapped to switch B and switch C is hidden from the tenant controller. OVX receives a PACKET OUT from the tenant controller, requesting to flood LLDP frames to discover the underlying topology. Instead of flooding the LLDP frame out through ports 1 and 2 of SW-A, OVX creates a fake LLDP frame as a response, to provide the tenant controller with the illusion that

there are only two switches, X and Y, connected to each other through ports X1 and Y1, respectively. Al-Shabibi *et al.* [57] describe the creation of this fake LLDP frame as *forging* an LLDP response. In addition to providing topology virtualization, this method ensures that OVX never allows tenant controllers to flood the physical network with LLDP messages. Therefore, the number of LLDP frames travelling through the physical network does not depend on the number of virtual networks [57].

An additional feature that OVX provides is *resiliency* through the use of backup routes within the network. When building abstract nodes with multiple paths between hosts, OVX can plan direct and backup routes. Flow rules are then installed to forward traffic through the direct routes by default. If a link within a direct route goes down, OVX detects this and automatically installs the backup route rules in the switches [57].

Figure 13 shows how OVX operates to provide network virtualization. The example uses the same network as shown in Figure 9, but outlining some major differences between OVX and FlowVisor. With OVX, the two switches SW1 and SW2 are represented as a single big switch (BIGSW) to the tenant controllers. In Step 1, OVX consults its database (previously configured by an administrator) to determine which hosts belong to which virtual networks. OVX then forwards PACKET IN messages to the tenant controller A based on that database. In step 2, OVX uses buffers to store the PACKET IN messages locally, thus avoiding sending unnecessary payload data to the tenant controllers. A buffer ID is used to identify buffers so that the same data can be returned to the network, as seen in Step 3. In Step 4 note that, unlike FlowVisor, the PACKET OUT message is immediately delivered at SW2 instead of going from SW1 to SW2 and only then delivered to host A2. This reduces the number of messages that need to travel through the data plane during the address resolution phase. OVX

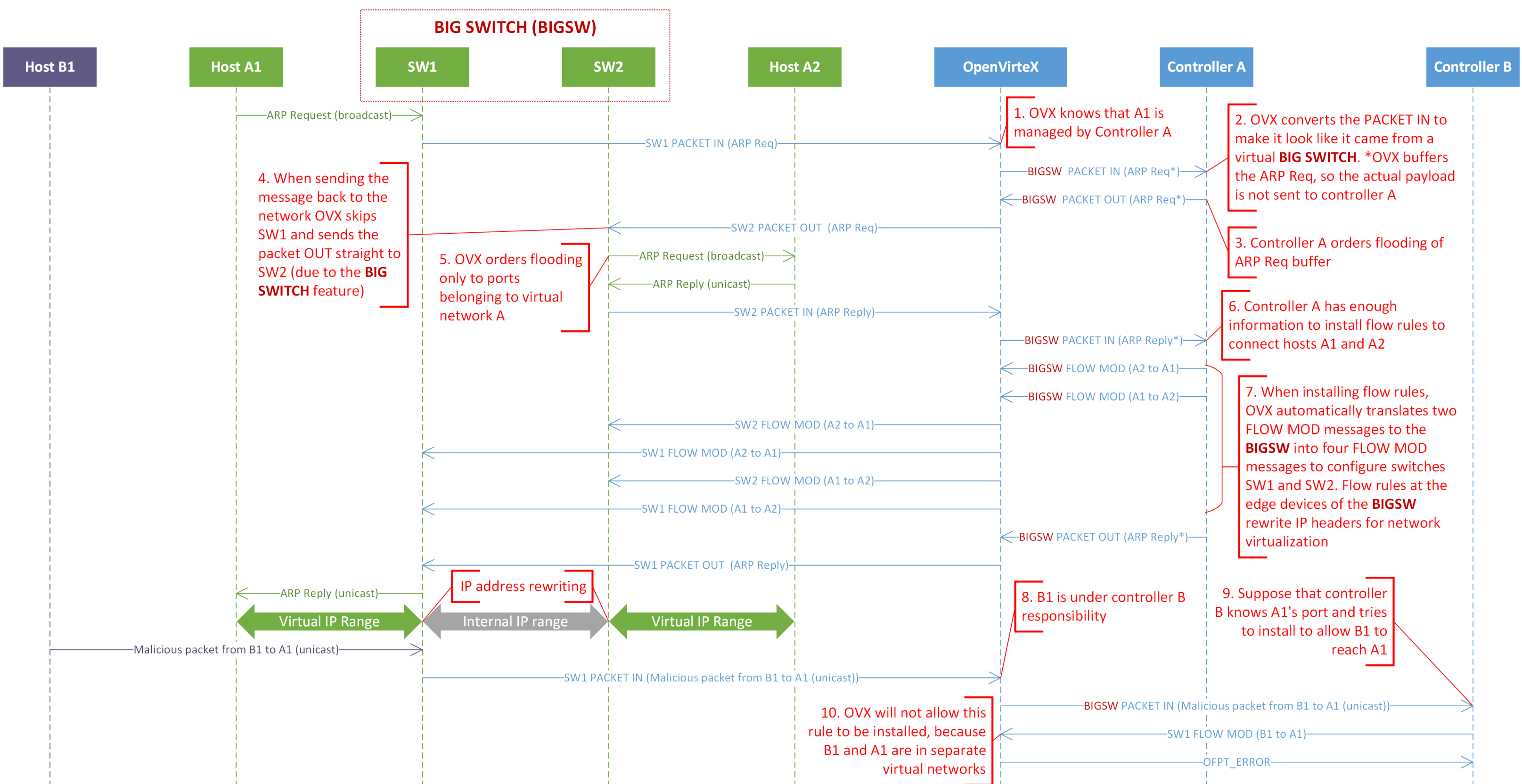

Figure 13. OVX packet flow processing.

then floods the ARP Request to all hosts belonging to virtual network A (see Step 5, only host A2 in this case). After receiving the second PACKET IN corresponding to the ARP Reply message from A2 to A1, OVX forwards it to controller A. At this point (Step 7), controller A sends two FLOW MOD messages to configure the big switch to interconnect hosts A1 and A2. OVX translates the two FLOW MOD messages to four FLOW MOD messages to program SW1 and SW2. This step is essential for the big switch implementation to work, as controller A sees and programs the network as if it were a single switch. Also in this step it is important to note that the FLOW MOD rules not only forward traffic, but also rewrite source and destination IP addresses at the edges of the big switch network. Steps 8-10 show what happens when host B1 and tenant controller B try to "invade" controller A's virtual network: OVX returns an OFPT ERROR message to controller B's unauthorized FLOW MOD.

2.11 FlowN

FlowN [16] proposes a *database* and lightweight *container based* virtualization as an extension to the NOX controller [39]. The database maintains a mapping of the physical and virtual networks, while each virtual container runs one tenant application with an independent address space. To ensure traffic isolation, FlowN adds VLAN headers to traffic entering the network and removes these headers as traffic leaves the network, allowing the tenants to reuse the same IP address space multiple times.

Instead of working as a proxy controller, FlowN is a modified version of the NOX controller. Therefore, it does not need to map OpenFlow protocol messages between the physical network and the virtual representation of the network that is exposed to tenant controllers. This significantly reduces the memory overhead of running

multiple controllers, because the virtualization is realized within NOX itself, while different tenants are applications running on the same controller, but in namespace containers. The FlowN authors compare it to FlowVisor and show that FlowN has superior performance in the presence of a great number of virtual networks (100 or more). The improved results are attributed to the use of a relational database to store the virtual to physical topology mapping, which is more scalable than FlowVisor's custom data structure mapping [16].

2.12 AutoSlice

AutoSlice [9] presents a system that automates the task of splitting the network's data forwarding plane among several tenants in the control plane. Its main objective is to minimize manual reconfiguration of the network so that substrate providers can resell slices of their networks to tenants. The implementation consists of a distributed hypervisor architecture composed of one management module (MM) and several controller proxies (CPX). The MM maps the virtual SDN (vSDN) topologies into the physical network, and assigns each CPX a fraction (a domain) of the network resources. The CPX is responsible for rewriting control messages from the control network to the data forwarding plane, using traffic tagging where needed to ensure isolation. AutoSlice also attempts to exploit traffic properties such as mouse and elephant flows to optimize caching of flow entries.

2.13 AutoVFlow

AutoVFlow [66] works similarly to AutoSlice, but proposes to give more control to the administrators of each vSDN, shifting the responsibility of handling flow space virtualization to the administrator of each network. It also claims to adopt the MAC rewriting technique in the network edges to ensure that the full header space is available to each of the tenants.

2.14 Summary

This section introduced key concepts required to understand the typical SDN architecture and operation, such as separation of the control and data planes, and how controllers control and discover network devices. VLAN, MPLS and VXLAN and VPNs were briefly explained, and it was shown that some of the SDN network hypervisors may still employ some of these traditional technologies to implement network virtualization. The next chapter will propose experiments to evaluate some of these network hypervisors.

Chapter 3

EXPERIMENTS

The experiments in this chapter were designed to evaluate and compare a few of the network hypervisors introduced in the last chapter. This chapter is divided into three main sections. Section 3.1 explains the experimental setup, the software used, and testbed design choices used for all the experiments; Section 3.2 describes procedures that can be used to verify virtual topology isolation, resiliency, traffic forwarding, and compliance with addressing standards; and Section 3.3 describes procedures that can be used to verify flow setup time and throughput achieved in virtualized networks.

3.1 Experimental Setup

Experiments are carried out in two different types of testbeds, a virtual Mininet-based testbed [36] to evaluate functional aspects of network virtualization, and a physical single-switch testbed to analyze basic performance aspects of network virtualization. A remote GENI testbed [21] is used to confirm some of the experimental results.

In all experiments the OpenFlow controller in use is Floodlight [19], because it is an easy to set up controller and contains all the functions needed to experiment with the networks under test, namely topology discovery and MAC learning.

Tutorials and source code are available to the public for FlowVisor [1], VeRTIGO

[12] and OpenVirteX [2]. AutoSlice, AutoVFlow and FlowN source code were not found online and, thus, only the first three approaches are covered in this work.

3.1.1 Mininet Topology

Mininet is a *network emulator*, and it works by managing instantiation of networks of virtual hosts, switches, and links. The virtual hosts are created as separate network namespaces in Linux and the virtual switches are typically Open vSwitches (OVS) [49]. Mininet allows the user to use an external OpenFlow controller to control OpenFlow enabled OVS switches [36].

The Mininet based topology depicted in Figure 14 is used for all of the functional experiments. Each switch has 3 hosts connected to it, though the hosts connected to switches SW2-SW4 are not shown for clarity. Additionally, the DPIDs and MAC addresses were shortened by using double colons to express a sequence of zeros. For instance, the DPID of SW3 (00:00:00:00:00:00:00:03) is represented by 00::03, and the MAC address of host C1 (02:00:00:00:03:01) is shown as 02::03:01. The addressing scheme of the virtual devices used in this network was devised to make experiments easier to configure, understand, and run:

- Hosts were assigned to three different networks, A, B, and C. Networks A and B (10.0.0.0/8) overlap with the purpose of testing network isolation benefits. Network C (3.0.0.0/8) is meant to verify possible conflicts with the header rewriting method, which is specific to OVX;
- Switch DPIDs are matched to the switch numbers 00::NN. For instance, switch SW3 ($NN = 03$) has DPID 00::03, to make it easier to troubleshoot and analyze results;

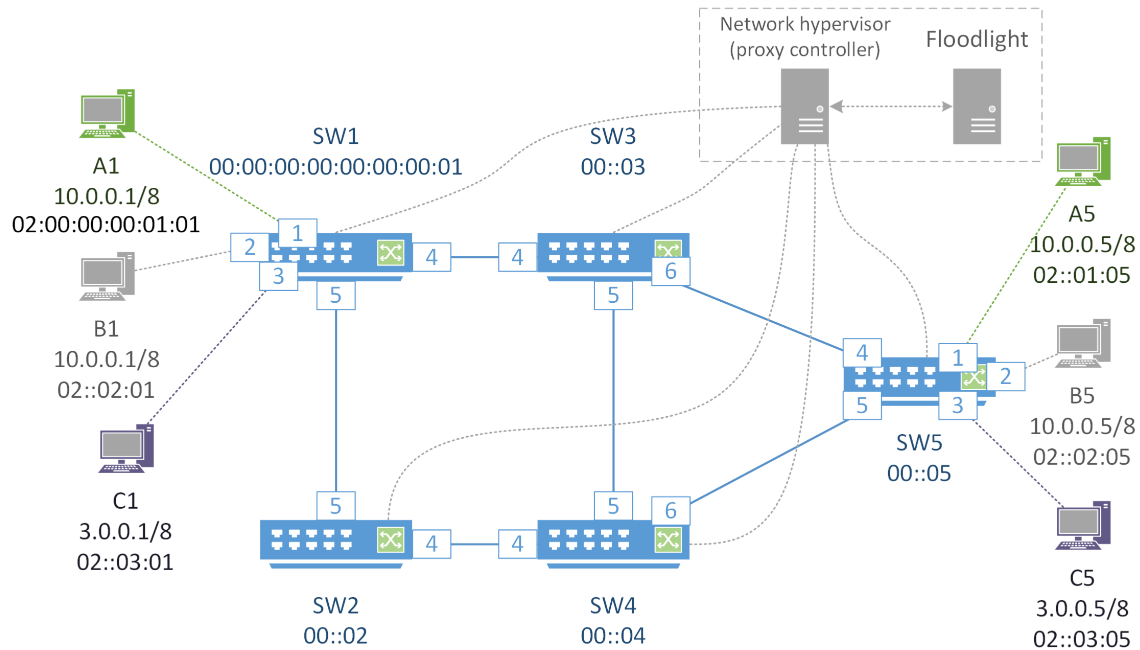

Figure 14. Mininet test scenario.

- Host MAC addresses were configured to reflect the network they belong to and the switch they are connected to by following the template 02::XX:YY, where XX is the network and YY is the switch. For instance, host B3 is in network B (represented by $XX = 02$) and switch SW3 ($YY = 03$), so its MAC address is 02::02:03.

Figure 14 also shows that the network hypervisor – OVX, VeRTIGO, or FlowVisor – is connected to the Open vSwitches through the loopback interface. The same loopback interface is used to connect the network hypervisor to Floodlight.

Five switches are used in this emulated topology to make sure that the different virtualization methods and features of each network hypervisor can be evaluated. This network can be used to demonstrate FlowVisor's slicing, OVX's resiliency with redundant routes, and both OVX's and VeRTIGO's topology virtualization capabilities. More details about this Mininet topology are available in the Appendix.

3.1.2 Physical Topology

The physical topology used for testing of flow setup time and throughput between hosts is depicted in Figure 15. Computers are connected to an HP OpenFlow Switch, which is controlled by the Floodlight controller plus any of the network hypervisors under test – OVX, FlowVisor or VeRTIGO. The testbed computers A to D have two network interfaces each, one connected to the HP OpenFlow switch, used in the performance tests, and another interface connected to a standard layer 2 switch to facilitate management of the lab equipment. VLAN 10 was configured to be controlled by the Floodlight controller. VLAN 40 was configured to be controlled by the network hypervisor plus Floodlight, to measure the performance impacts of adding this network virtualization layer to the control plane. Therefore computers A and B are used when only Floodlight performance needs to be tested, and computers C and D are used when the network hypervisor and Floodlight performance is tested.

For experiments conducted in this local physical topology, the OpenFlow hardware switch is connected to the network hypervisor, which, in turn, connects to a single Floodlight tenant controller instance at TCP port 10000.

The main purpose of this physical test topology is to run performance tests while not being limited by Mininet's consumption of CPU resources, which could happen for higher throughput traffic. It is a much simpler topology because it is intended to be used only to measure the throughput of traffic crossing the hardware switch and the time to install new flow rules in the switch. These experiments will be described later in this chapter.

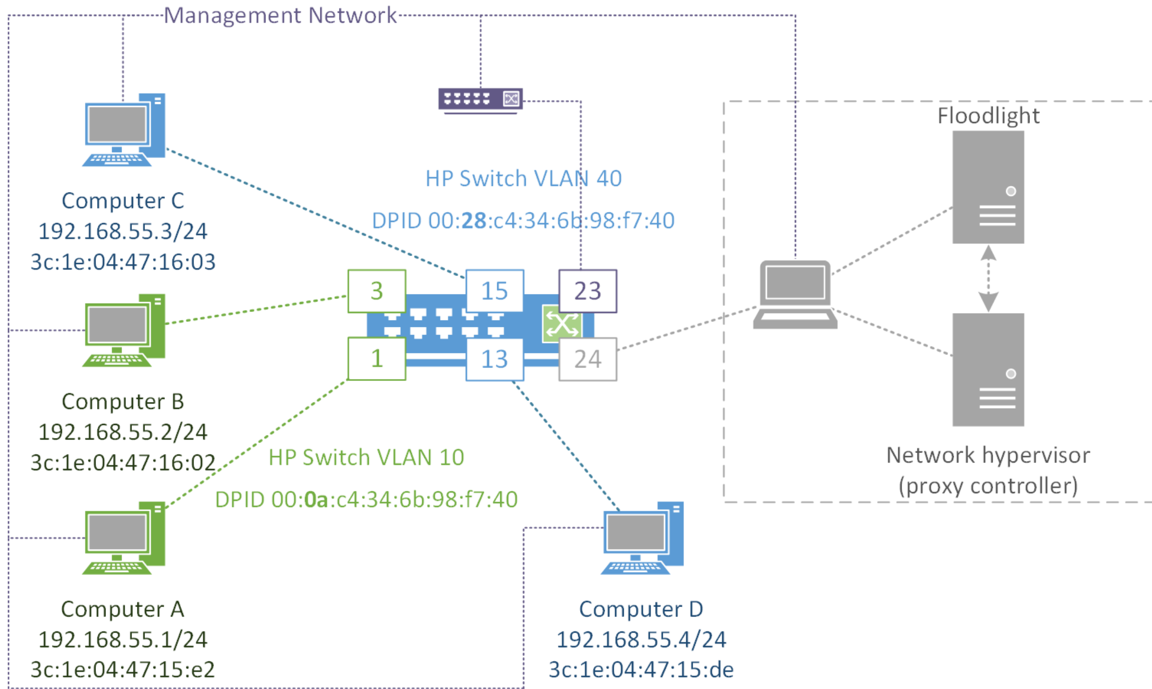

Figure 15. Physical test scenario for performance tests.

3.1.3 GENI Topology

For comparison purposes, a test topology was also set up using the Global Environment for Network Innovations (GENI) [21]. An OpenFlow hardware switch was reserved using an adapted resource specification from a GENI tutorial [22]. The hardware switch at the GENI testbed connects via the Internet to the local network hypervisor instance in a local computer. The network hypervisor, in turn, connects to a single instance of the Floodlight tenant controller. This is depicted in Figure 16.

The reason to have this third test topology is to confirm experimental performance results from the physical topology described in Section 3.1.2. Hosts 1 and 3 are configured with iperf and are used to measure throughput through the GENI hardware switch.

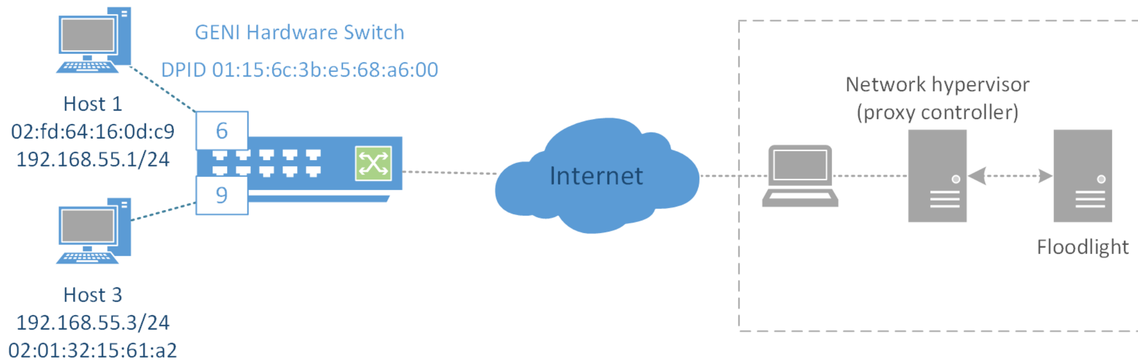

Figure 16. GENI test scenario for throughput test.

3.1.4 Floodlight Setup

Due to some port conflicts, Floodlight [19] requires some changes to its default configuration in order to be successfully set up in the same machine as the other network hypervisors. Some of its default ports, such as 8080 for the REST API, conflict with other applications. Additionally, its OpenFlow version negotiation for versions 1.1 to 1.4 needs to be turned off, because none of the network hypervisors tested here support OpenFlow versions higher than 1.0.

There are three different configuration files for Floodlight. These are used to instantiate up to three instances of Floodlight at ports 10000, 20000, and 30000, according to the experimental needs. More details about the configuration can be found in the Appendix.

3.1.5 FlowVisor Setup

To set up FlowVisor the standard installation procedure from [1] is used. After completing the installation, FlowVisor is configured through a command line interface control tool called *fvctl*. The configuration done in FlowVisor can be output to a

JSON file and loaded later. Configuration details for all experiments can be found in the Appendix.

The original Mininet network from Figure 14 cannot be set up with FlowVisor because it does not work with overlapping IP addresses. Therefore, the topology tested in FlowVisor has a slight variation from Figure 14: network B is changed from 10.0.0.0/8 to 11.0.0.0/8 to avoid overlapping with network A. The network is sliced based on source MAC addresses to make the configuration as similar as possible with the standard OVX slicing, which is done by specifying which MAC addresses belong to each network.

3.1.6 VeRTIGO Setup

VeRTIGO setup and installation is very similar to FlowVisor's and it is described in VeRTIGO's repository [12]. However, instead of using a JSON file for configuration, VeRTIGO relies on an XML file to store its configuration. Although Doriguzzi Corin *et al.* [15] claim that VeRTIGO has capabilities to create virtual links, ports and nodes, their latest published version in [12] does not provide all of that functionality to the user through command line configuration. The only extra function available to users is the virtual link, which apparently does not work by itself. More details of the final configuration used for experiments, plus start and stop scripts for the VeRTIGO test scenario can be found in the Appendix.

3.1.7 OVX Setup

Installing and configuring OVX is described in [2]. Although it is possible to configure OVX through a command line interface, the quickest and most efficient way to configure it is to use their *network embedder*, a module that automatically maps a virtual topology onto the physical topology based on a configuration received in JSON format.

The configuration used in the experiments of this work, as well as start and stop scripts for experiments are detailed in the Appendix.

3.1.8 Scapy

Scapy is a Python-based program which can be used to create packets and inject them into the network. It uses Python's object oriented capabilities to provide the user with simple interfaces to create packets by calling and stacking functions together. It can handle many different protocols, and also allows the user to create their own protocol [56]. Scapy can be downloaded from Github [55].

A command line packet generation tool was created with the purpose of testing different types of traffic that can be handled by the network or equipment under test. It uses a variety of sample frames and payloads extracted from real tcpdump captures, as well as packets assembled from scratch using only Scapy to generate a series of packets. The tool contains options to generate more than 40 different types of packets and allows the user to change source/destination MAC and IP addresses, insert VLAN tags and/or MPLS labels in the packet, and other small changes. The code for this

tool is available at [51], and the protocols that the tool can generate are listed in the Appendix.

This tool is used by injecting frames into one interface in the network and monitoring other network interfaces with tcpdump [61] or Wireshark [65] to observe if the frames reach their intended destination. The network is expected to *transparently* forward most frames, which means that the frames should not be dropped or altered by the network.

3.2 Functional Experiments

Functional experiments include verifying the network hypervisor's capabilities to isolate network topologies; to provide autonomous rerouting inside abstract nodes for added resiliency; to allow transparent traffic forwarding; and to comply with IP and MAC addressing standards. The experiments in this section are inspired by trying to answer the following questions about network hypervisors:

- Are the virtual networks they provide truly isolated? Is it possible for a tenant controller to control networks belonging to other tenants?
- Can they transparently handle traffic with reserved MAC addresses [28], MAC/IP multicast addresses [28], [29], IPv6 [14] or VLAN tagging?
- Do the header rewriting techniques respect current MAC [28] and IPv4 [53] addressing standards defined by Internet Assigned Numbers Authority (IANA) and the Internet Engineering Task Force (IETF)?
- What kind of applications would benefit from the network virtualization methods presented here? Can network hypervisors be used for wide area network VPN applications or just data center network virtualization?

For all the functional experiments in this section the OpenFlow switches are connected to the network hypervisor at port 6633. The network hypervisor, in turn, connects to the three Floodlight tenant controller instances A, B, and C at ports 10000, 20000, and 30000, respectively.

3.2.1 Network and Topology Isolation

For this experiment, the Mininet topology illustrated in Figure 14 and described in Section 3.1.1 is used to verify network isolation among tenants. First, the network hypervisor under test – FlowVisor, OVX or VeRTIGO – must be configured to split this network into three separate virtual networks, each controlled by a different instance of Floodlight.

FlowVisor is expected to split the original network as illustrated in Figure 17. For OVX and VeRTIGO, network isolation should look like what is shown in Figure 18, with a big switch representing the five switches. Network isolation can be verified by observing that the hosts belonging to different slices cannot communicate. For instance, host A1 should not be able to send any packets to any of the C1-C5 hosts. Additionally, the network hypervisor must ensure that the tenant controllers are authorized to control only their network portions.

The network virtualization must be guaranteed both from a client and from a controller perspective. To verify this, one possible procedure is to:

1. Try to reach C5 from A1 by sending a packet straight to the destination with *test_packets.py* (should not work);
2. Try to reach A5 from A1 using the same method (should work);

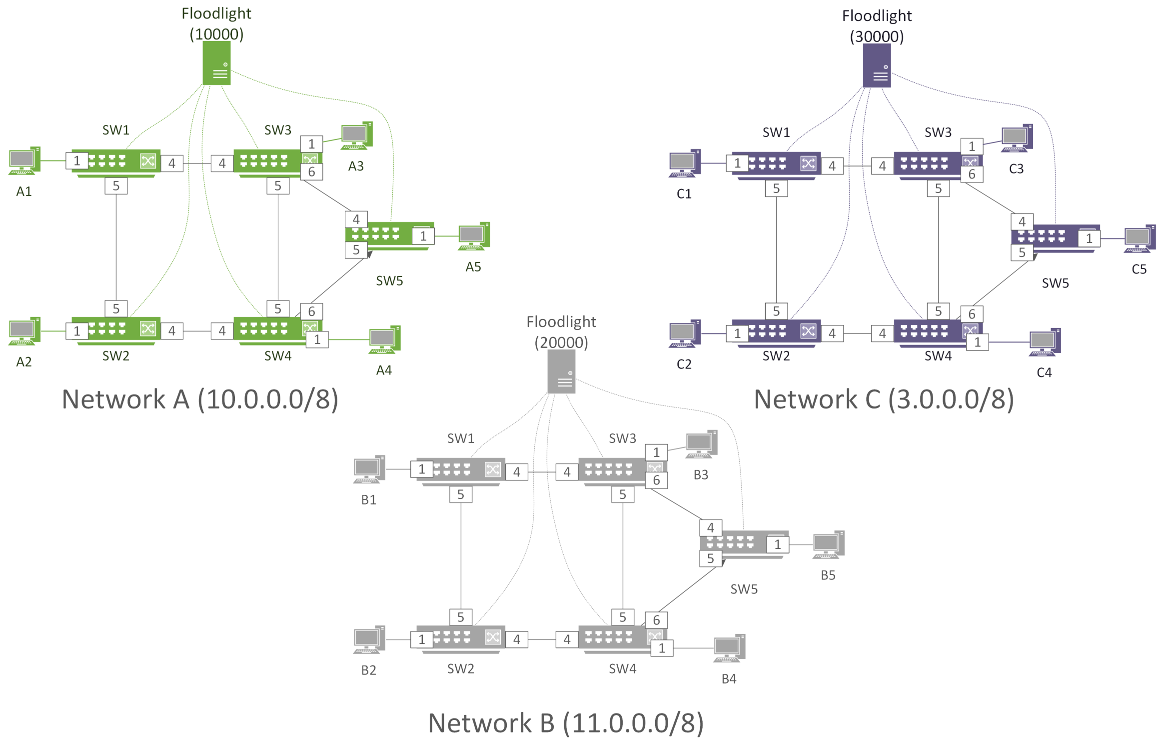

Figure 17. Topology isolation expected with FlowVisor virtualization.

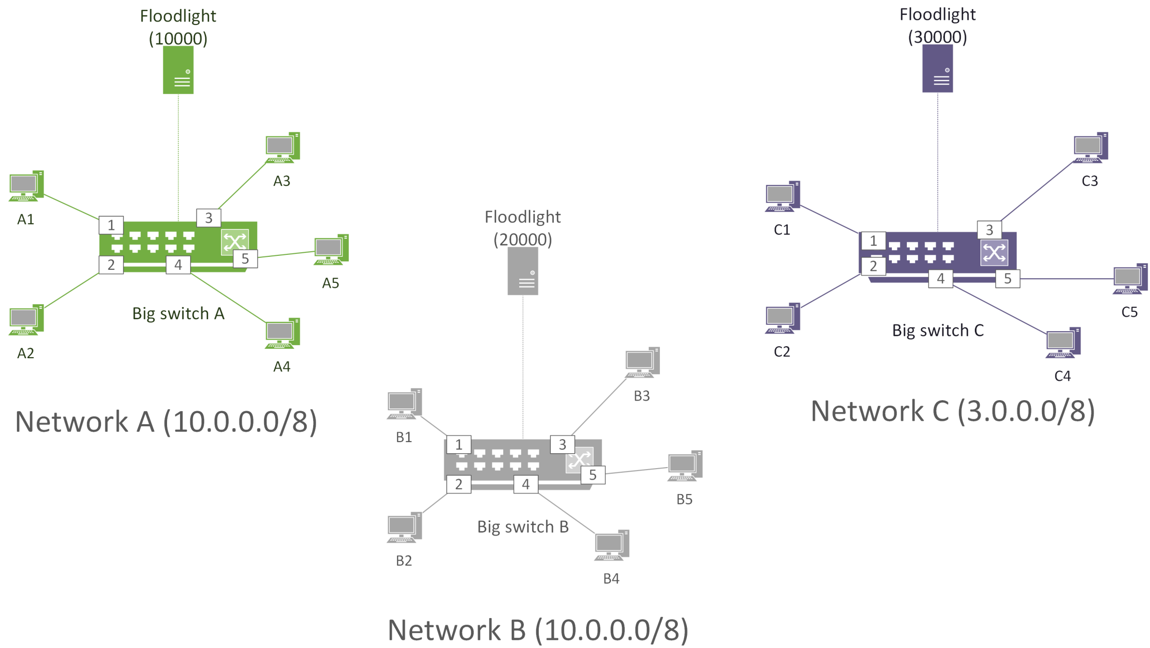

Figure 18. Topology isolation expected with OVX and VeRTIGO virtualization.

3. Try to reach C5 from C1 using the same method (should work);
4. Try to reach every node from every node using Mininet's *pingall*. This should show that hosts in network A can only reach other hosts in network A, and the same applies for networks B and C;
5. Verify from the controllers' interfaces that each controller (A, B and C) sees only their own network hosts.

This sequence is repeated with each different network hypervisor running between the physical network and the Floodlight instances.

3.2.2 Autonomous Rerouting

Both OVX and VeRTIGO claim to be able to provide autonomous rerouting inside abstract nodes. This idea is similar to what is provided by the fast reroute extensions for RSVP-TE for LSP Tunnels [48]. Only OVX and VeRTIGO should support this experiment, and it is conducted in the Mininet topology described in Section 3.1.1.

Suppose that the network hypervisor routes traffic between hosts A1 and A2 through switches SW1 and SW2, as represented by the arrow with a solid line in Figure 19. If the link between SW1 and SW2 fails, the network hypervisor should keep the network functional by rerouting the traffic via a backup route, perhaps through switches SW1-SW3-SW4-SW2.

One possible procedure to verify support for autonomous rerouting is:

1. Create a single virtual switch network and establish test traffic between a pair of network hosts (should work);

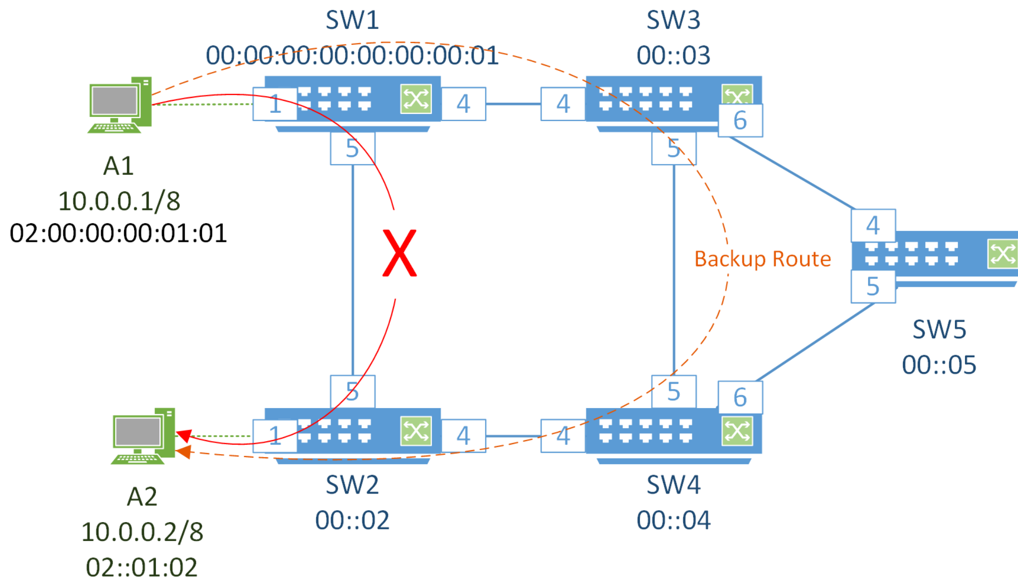

Figure 19. OVX Fast Reroute feature.

2. Determine which links are used by the active and backup routes by inspecting the switches flow tables;
3. Shutdown a link belonging to the active route;
4. Traffic should then be forwarded through the backup route, perhaps suffering a minor interruption;
5. Bring the link from step 3 back up;
6. Traffic should be forwarded back through the original route.

To test this, it is important to enable the "backup routes" option in the JSON configuration file:

```
"routing": {
```

```
"algorithm": "spf", // shortest path first
"backup_num": 1 }   // number of backup routes
```

Traffic interruption time can be measured by using a traffic generator, such as iperf. The traffic generator has to be configured to send UDP packets, to make sure that the application layer will not resend lost packets. By sending packets with known size and at a fixed rate, it is possible to measure the interruption time by counting the number of lost packets. Equation 3.1 shows how to calculate *bandwidth* based on the volume of data received ($data_{rx}$) and *time* elapsed.

$$bandwidth = \frac{data_{rx}}{time} \tag{3.1}$$

Considering that $data_{rx} = N_{packets} \times packet\ size$, and known *packet size*, *bandwidth*, and *time*, Equation 3.1 can be rewritten as Equation 3.2.

$$N_{packets} = \frac{time \times bandwidth}{packet\ size} \tag{3.2}$$

For instance, using a *packet size* of 125 bytes (1000 bits), a *bandwidth* of 1 Mbps and sending data for 10 seconds should result in $N_{packets} = 10000$ packets. Therefore, each packet lost corresponds to $10/10000 = 1$ millisecond. It is straightforward to estimate how much time the network remains unavailable by counting the number of lost packets.

3.2.3 Transparent Traffic Forwarding

Network tenants may want to run traditional distributed network applications in their own virtual network, such as the LLDP discovery mechanism, or the open

shortest path first (OSPF) routing protocol. Both LLDP and OSPF rely on the use of multicast frames to communicate with neighboring network devices. Network virtualization is usually well tested for standard unicast IP traffic, but some multicast frames and other special frame types are hardly ever verified for *transparent traffic forwarding* in experimental network hypervisors. Therefore, it is important to verify that several different types of traffic can be forwarded by virtual networks.

This experiment is conducted in the Mininet topology described in Section 3.1.1. The results are compared with a standard Ethernet layer 2 switch, and an Open vSwitch in standalone bridge mode. Both are used as a reference for the experiment, depicted in Figure 20. The Ethernet switch used for testing is a NETGEAR Fast Ethernet FS108, but any other standard Ethernet layer 2 switch should yield the same results.

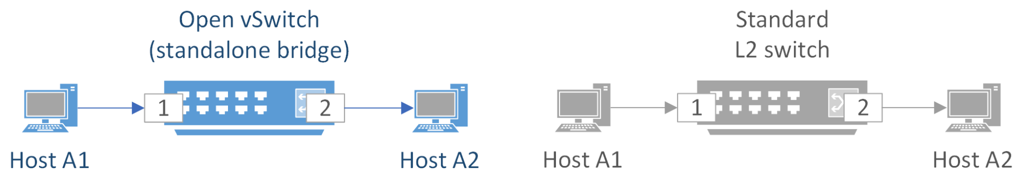

Figure 20. Transparent traffic forwarding test reference.

One way of verifying transparent traffic forwarding consists of using the *test_packets.py* tool [51], described earlier in Section 3.1.8. The user has to inject frames into one network port and then expect the frames to be received at any other destination host in the network. The network is expected to forward frames to ports belonging to the correct slice or virtual network without dropping or altering them. The following set of steps can be used to verify transparent traffic forwarding:

1. Set up the *test_packets.py* tool to send traffic from host A1 to A2;

2. Run Wireshark at hosts A1 and A2;
3. Send a predefined set of packets from host A1 to A2 through an Open vSwitch configured in standalone bridge mode. The predefined set of packets is documented in the Appendix;
4. With Wireshark, save the capture files including the frames sent from A1 and the frames received at A2;
5. Compare the capture files and take note of which frames were dropped or altered by the network;
6. Run steps 3-5 using the hardware Ethernet layer 2 switch;
7. Repeat steps 3-5 using Floodlight to control the network;
8. Repeat steps 3-5 using FlowVisor and Floodlight to control the network;
9. Repeat steps 3-5 using OpenVirteX and Floodlight to control the network;
10. Repeat steps 3-5 using VeRTIGO and Floodlight to control the network.

The different network hypervisors should not further restrict the type of traffic which can be normally forwarded by the Floodlight controller alone.

3.2.4 Compliance with Addressing Standards

FlowVisor and VeRTIGO do not change the traffic in the network. They merely slice the available header space so that the different tenants are handled by different tenant controllers. OVX, however, works by rewriting IP headers to make the full IP header space available to each tenant [57].

The purpose of this experiment is to observe how the network hypervisor changes IP and MAC headers as they enter the network. The traffic flowing through internal links of the network is captured in the Mininet topology described in Section 3.1.1

(illustrated in Figure 14). Then the captured frames are analyzed while bearing in mind compliance with IP and MAC addressing standards and reserved ranges of addresses, as specified by:

- IANA IP multicast [29] and MAC address allocation [28];
- IETF address allocation for private networks [53];
- IEEE bridge control protocols, such as the spanning-tree protocol (STP) or generic attribute registration protocol (GARP) [25];
- Other general addresses already assigned to vendors or reserved for specific purposes. Wireshark maintains an updated database of Ethernet vendor codes and well-known MAC addresses [64].

For instance, the address range 01:00:0C:xx:xx:xx is used by several of Cisco's proprietary protocols [64]. The easiest method to verify if a specific MAC address is taken is to consult Wireshark's database [64].

IPv4 address compliance means that the IP addresses generated by the network hypervisor are valid and within the ranges assigned to private networks. For instance, 1.0.0.1 is a public IP address assigned to an organization in Asia [63], and 224.0.0.5 belongs to the IP multicast range reserved by IANA and is specifically used for exchange of OSPF messages [53]. Neither of these addresses should be used by network hypervisors when rewriting unicast IP headers, as they could cause many problems if there is ever a need to interoperate with other IP networks. Private IP packets could leak to the Internet and travel all the way to Asia, or they could be interpreted as IP multicast addresses and flooded.

The *test_packets.py* tool is used to generate the required frames for address compliance checks:

1. Ensure that hosts A1 and A2 from the same virtual network can communicate;
2. Send IPv4 packets from A1 to A2 and capture the traffic in the link between SW1 and SW2;
3. Send ARP packets from A1 to A2 and capture the traffic in the link between SW1 and SW2;
4. Send IPv4 multicast packets from A1 and capture the traffic in the link between SW1 and SW2;
5. Send frames with reserved MAC multicast frames from A1 and capture the traffic in the link between SW1 and SW2;
6. Execute the previous steps for FlowVisor, VeRTIGO, and OpenVirteX.

FlowVisor and VeRTIGO should not affect the IP and MAC headers in any way, because they do not work by modifying packets. Layer 2 and Layer 3 traffic rewritten by OVX should comply with the addressing standards discussed in this section.

3.3 Performance Experiments

Flow setup time is already recognized as one of OpenFlow's (and SDN) disadvantages of having a centralized controller taking care of forwarding decisions [62]. Additionally, it is important to confirm that the virtualization techniques employed by network hypervisors do not affect network throughput significantly. For instance, Al-Shabibi *et al.* [57] claim that OVX introduces a negligible drop in performance due to the header rewriting technique. The experiments described in this section intend to measure:

- The flow setup time overhead introduced by the virtualization element in the network;

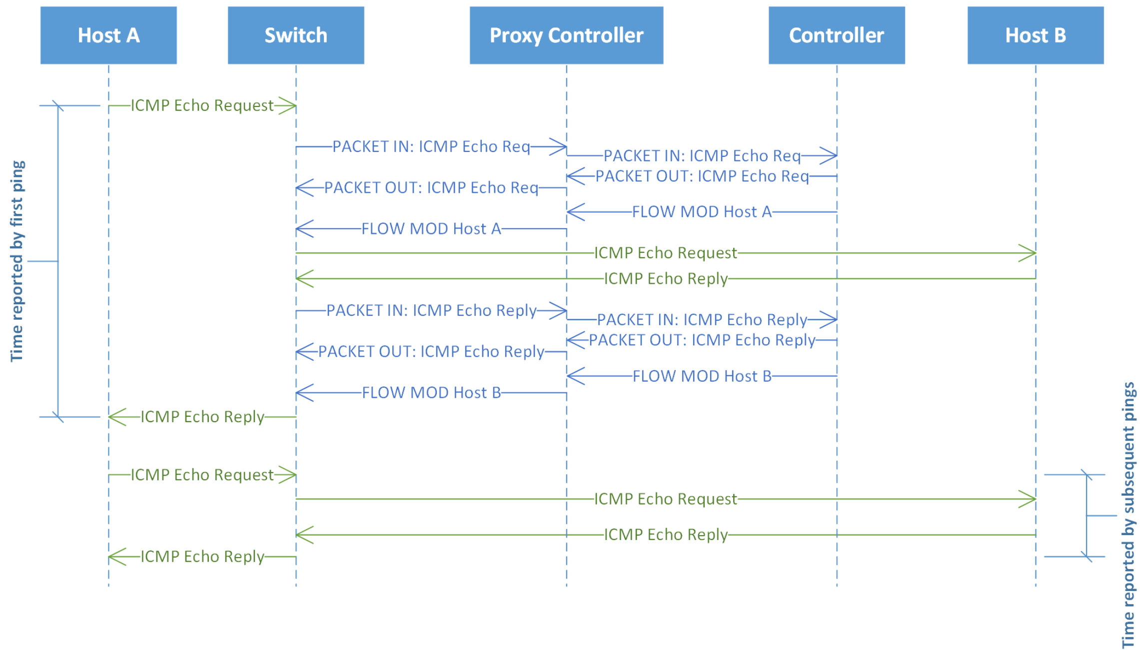

Figure 21. Round trip time of the first ping with network hypervisor.

- Decreased data throughput caused by traffic isolation techniques, such as rewriting of packet headers.

3.3.1 Flow Setup Time

The objective of this experiment is to compare the latency of pure Floodlight established flow rules versus the latency with the added network hypervisor between Floodlight and the OpenFlow switch. This experiment is conducted in the physical topology described in Section 3.1.2 and illustrated in Figure 15. The ARP entries on all hosts were manually added to their tables to avoid additional variance introduced by the address resolution phase.

The procedure consists of measuring the round trip time (RTT) of a ping between two hosts. As depicted in Figure 21, when the switch has no rules for the traffic

between hosts A and B, the Internet control message protocol (ICMP) echo request and reply messages must be processed by the controller. By comparing the RTT of the first ping with subsequent RTTs, it is possible to determine how much time the network hypervisor and the tenant controller add to the whole RTT Equation 3.3.

$$RTT_{firstping} = t_{processing} + t_{transmission} + t_{queuing} + t_{propagation} \tag{3.3}$$

The propagation, queuing and transmission delays are negligible in this small network, which is tested without any significant load. Therefore, the time it takes for the flow to be setup bidirectionally depends almost exclusively on the processing time added by the network hypervisor proxy and the Floodlight controller, as seen in Equation 3.4:

$$RTT_{firstping} \approx t_{processing_{proxy}} + t_{processing_{floodlight}} \tag{3.4}$$

A script was written to perform the following steps to implement the flow setup time test:

1. Start a ping with count 1 from host A to B and record its RTT;
2. Start a ping with count 3 from host A to B and record the average RTT;
3. Record the results in a comma separated values (CSV) file;
4. Sleep for enough time so that the flows in the switch expire. Floodlight's flow rules are configured to expire after spending 5 seconds unused. Sleeping for 8 seconds gives enough time to let the flow rules expire;
5. Repeat the previous steps 50 times.

The RTT recorded in step 1 represents the bidirectional flow setup time from the network user's perspective, and the average RTT recorded in step 2 represents the

network's actual latency after the flow rules are installed and the network ready to be used. The script used for this test can be found in the Appendix.

3.3.2 Throughput

Throughput loss may be introduced by the traffic isolation techniques used in network virtualization, such as VLAN tagging or rewriting of packet headers. The objective of this experiment is to determine how much throughput is lost due to network virtualization.

This experiment is conducted twice with two different hardware switches: Once in the scenario depicted in Figure 15 and once in the GENI testbed illustrated in Figure 16. Hardware switches are used to evaluate throughput, as they are expected to install flow rules in hardware flow tables and forward traffic at line rate. Since this experiment depends on hardware capabilities offered by vendors, two different testbeds are used to confirm experimental results.

One way of doing this is to determine the maximum throughput between two hosts with the *iperf* tool for each case. Scripts are used to do the following:

1. Using the physical test scenario (Figure 15) start a ping with a low count from hosts A to B to get the flows installed by Floodlight in the switch;
2. Immediately start an *iperf* TCP test through the same path;
3. Record the results in a CSV file;
4. Sleep for enough time so that the flows in the switch expire. Floodlight's flow rules are configured to expire after spending 5 seconds unused. Sleeping for 8 seconds gives enough time to let the flow rules expire;
5. Repeat the previous steps 30 times;

6. Repeat steps 1-5 using hosts C and D to measure throughput achieved by flows installed by FlowVisor and Floodlight;
7. Repeat steps 1-5 using hosts C and D to measure throughput achieved by flows installed by VeRTIGO and Floodlight;
8. Repeat steps 1-5 using hosts C and D to measure throughput achieved by flows installed by OVX and Floodlight;
9. Repeat all the steps using the GENI testbed (Figure 16), using hosts 1 and 3 to measure throughput for all cases.

The effect of the network virtualization layer on throughput is expected to be negligible. The throughput achieved with the use of network hypervisors should approach the throughput measured when using Floodlight alone.

3.4 Summary

This section described in detail the experimental setups used for evaluation of network hypervisors, as well as some of the tools used to test them. Several experimental procedures were suggested and detailed. The next chapter will present and analyze the results of these experiments.

Chapter 4

RESULTS AND ANALYSIS

This chapter presents the results that were collected by following the experimental procedures from the last chapter. Functional test results are presented first in Section 4.1, followed by the performance test results in Section 4.2, and a general analysis of all the results in Section 4.3.

4.1 Functional Test Results

Functional experimental results are inherently qualitative and consist of determining whether the result is acceptable or not. This section only attempts to present and briefly analyze the results. A deeper overall analysis of the results is later presented in Section 4.3.

The following subsections present results regarding network and topology isolation; support of autonomous rerouting; transparent traffic forwarding; and compliance with addressing standards.

Although this section was originally intended to evaluate the three network hypervisors, most of the results in this section concern only FlowVisor and OVX. Unfortunately, VeRTIGO's implementation was not complete and stable enough to be tested by the experimental procedures described in Section 3.2.

4.1.1 Network and Topology Isolation

The Floodlight GUI was used to verify that the network hypervisors were exposing the correct virtual network representations to tenant controllers. Three instances of the Floodlight tenant controller were spawned in ports 10000, 20000 and 30000 to control networks A, B, and C respectively.

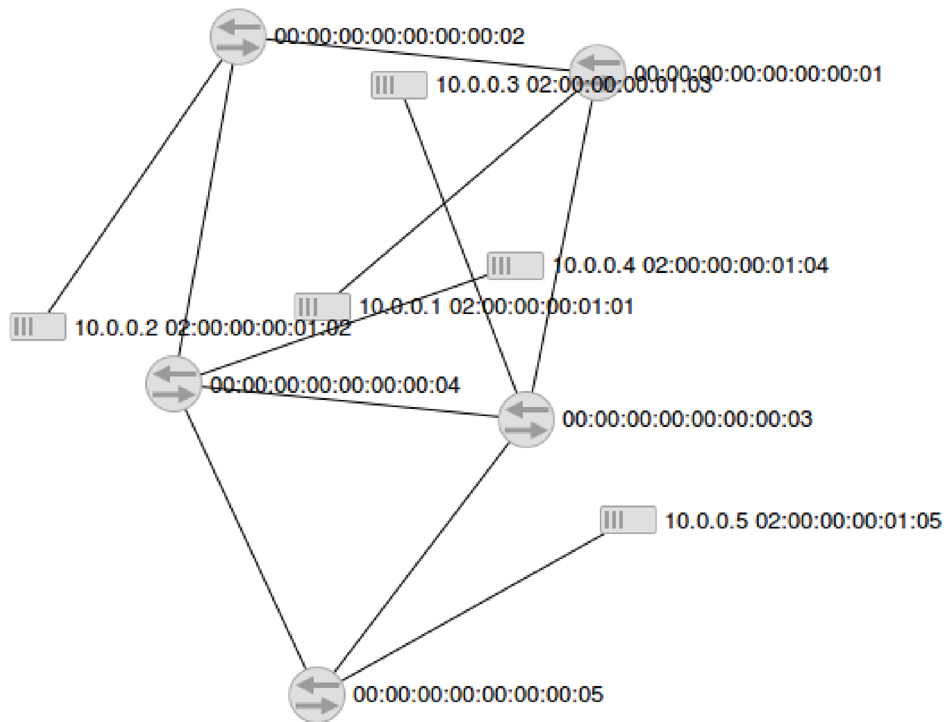

Figure 22. Network A discovered by Floodlight instance A using FlowVisor as the network hypervisor.

Figure 22 shows network A from Floodlight's perspective as presented by FlowVisor, the network hypervisor. Five switches with DPIDs ranging from 00::01-05 are shown, which correspond to switches SW1-SW5 in the Mininet topology presented in Section 3.2.1, depicted in Figure 17. All the hosts in this network have MAC addresses

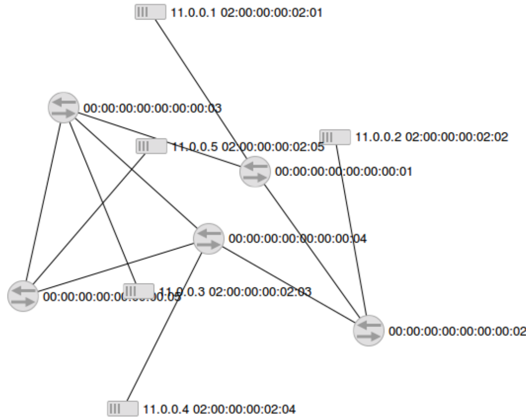

Figure 23. Network B discovered by Floodlight B through FlowVisor.

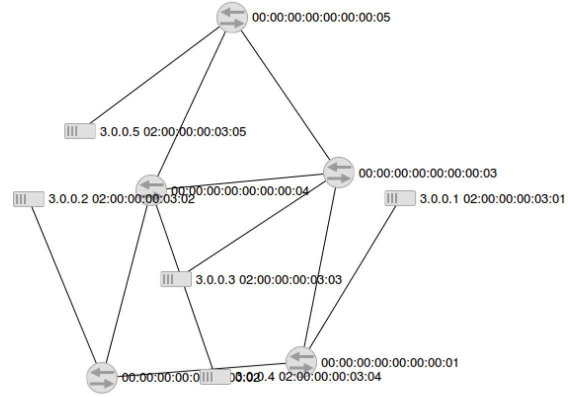

Figure 24. Network C discovered by Floodlight C through FlowVisor.

02::01:XX, and these correctly correspond to hosts A1-A5. Analogous results may be observed in Figures 23 and 24, which contain hosts B1-B5 and hosts C1-C5 respectively. This shows that FlowVisor works as expected, allowing network isolation by slicing a single network into three different networks that operate independently.

As observed in Figure 25, the Floodlight instance A “sees” only a single virtual switch presented by OVX. This virtual switch has DPID 00:a4:05::01 and is only a representation of the five switches from Figure 14. The same analysis used for FlowVisor is repeated, comparing which hosts are connected to each network. From this analysis it is confirmed that hosts with MAC address 02::01:XX appear only in network A. The same can be confirmed from Figures 26 and 27, where hosts 02::02:XX appear only in network B and hosts 02::03:XX only in network C. Therefore, OVX also works as expected and allows network isolation by virtualizing a single network into three abstract instances that operate independently.

Network isolation was also verified from the network hosts perspective. Using Mininet’s pingall utility for all hosts shows that hosts A1-A5 cannot reach hosts B1-B5

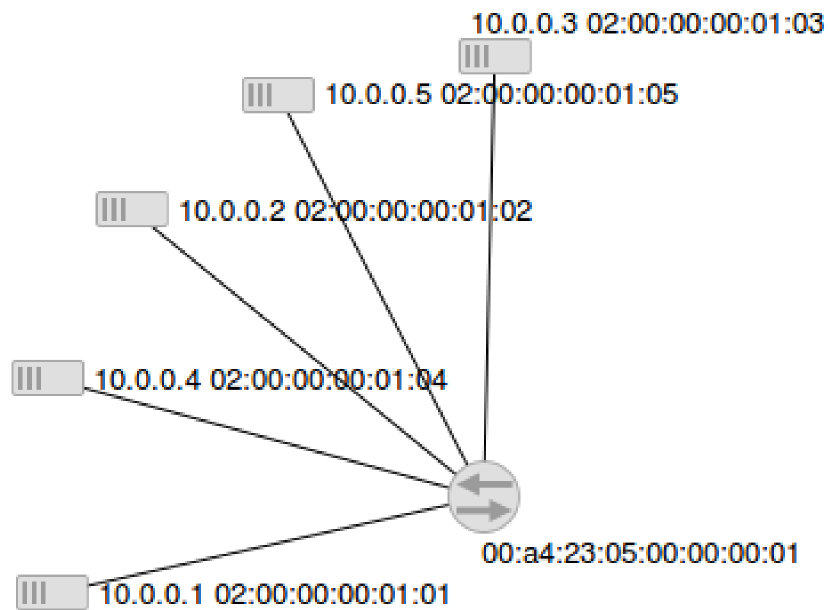

Figure 25. Network A discovered by Floodlight instance A using OVX as the network hypervisor.

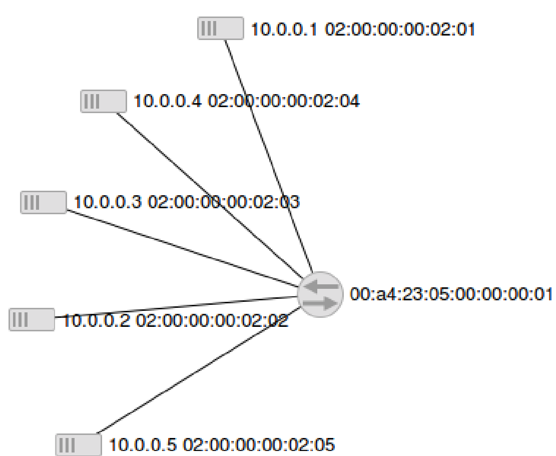

Figure 26. Network B discovered by Floodlight B through OVX.

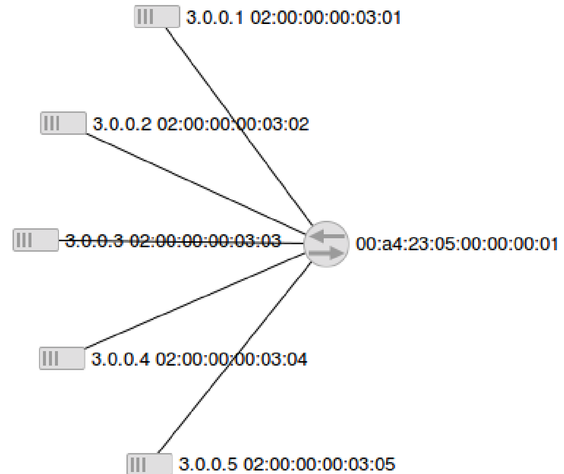

Figure 27. Network C discovered by Floodlight C through OVX.

nor C1-C5. Table 1 shows this reachability in matrix form. For instance, row B3 intersection with column C1 has an "N", indicating that B3 cannot reach C1. Row C5 intersection with column C1 has a "Y", which means that C5 can reach C1. Both OVX and FlowVisor network hypervisors produced correct connectivity results.

Table 1. Host connectivity as observed by tenant hosts in Mininet

	A1	A2	A3	A4	A5	B1	B2	B3	B4	B5	C1	C2	C3	C4	C5
A1	–	Y	Y	Y	Y	N	N	N	N	N	N	N	N	N	N
A2	Y	–	Y	Y	Y	N	N	N	N	N	N	N	N	N	N
A3	Y	Y	–	Y	Y	N	N	N	N	N	N	N	N	N	N
A4	Y	Y	Y	–	Y	N	N	N	N	N	N	N	N	N	N
A5	Y	Y	Y	Y	–	N	N	N	N	N	N	N	N	N	N
B1	N	N	N	N	N	–	Y	Y	Y	Y	N	N	N	N	N
B2	N	N	N	N	N	Y	–	Y	Y	Y	N	N	N	N	N
B3	N	N	N	N	N	Y	Y	–	Y	Y	N	N	N	N	N
B4	N	N	N	N	N	Y	Y	Y	–	Y	N	N	N	N	N
B5	N	N	N	N	N	Y	Y	Y	Y	–	N	N	N	N	N
C1	N	N	N	N	N	N	N	N	N	N	–	Y	Y	Y	Y
C2	N	N	N	N	N	N	N	N	N	N	Y	–	Y	Y	Y
C3	N	N	N	N	N	N	N	N	N	N	Y	Y	–	Y	Y
C4	N	N	N	N	N	N	N	N	N	N	Y	Y	Y	–	Y
C5	N	N	N	N	N	N	N	N	N	N	Y	Y	Y	Y	–

VeRTIGO could not be evaluated because of stability issues. While bringing up the Mininet topology, VeRTIGO would restart the connection with Floodlight indefinitely. The problem seems to be a bug in VeRTIGO, which is very likely to be related to the encapsulation of LLDP packets from the Open vSwitches. It was not possible to solve this issue with configuration alone.

4.1.2 Autonomous Rerouting

During topology configuration, OVX presents information about the virtual switches and ports that map to the physical topology. It is essential to understand what OVX does during network configuration phase, in order to verify that the autonomous rerouting results are correct.

During network A configuration, OVX logs show that it creates a big-switch X, corresponding to the switch shown in Figure 25, a representation of the network shown in Figure 14. Then it creates virtual ports 1 and 2 in switch X and connects hosts A1 and A2 to them. This is shown in the following section of edited log output from OVX:

```
Set routing algorithm spf for big-switch X in virtual network A
Created virtual port 1 on virtual switch X in virtual network A
Connected host A1 to virtual port 1 on virtual switch X
Created virtual port 2 on virtual switch X in virtual network A
Connected host A2 to virtual port 2 on virtual switch X
```

OVX then proceeds to calculate direct routes. This is observed in the following section of edited log output, where the direct route between hosts A1 and A2 is configured bidirectionally through the link between SW1 and SW2:

```
Add route for big-switch X between virtual ports (1,2) with
      priority: 64
      path: [SW1 port 5 <-> SW2 port 5]
Add route for big-switch X between virtual ports (2,1) with
      priority: 64
```

```
        path: [SW2 port 5 <-> SW1 port 5]
```

And then backup routes are calculated, as observed in the following output which adds a secondary route with lower priority between hosts A1 and A2 that goes through the path SW1-SW3-SW4-SW2 bidirectionally:

```
  Add backup route for big-switch X between ports (1,2) with
        priority: 63
        path: [SW1 port 4 <-> SW3 port 4, SW3 port 5 <-> SW4 port 5,
               SW4 port 4 <-> SW2 port 4]
  Add backup route for big-switch X between ports (2,1) with
        priority: 63
        path: [SW2 port 4 <-> SW4 port 4, SW4 port 5 <-> SW3 port 5,
               SW3 port 4 <-> SW1 port 4]
```

Based on the log output, link failure between SW1 and SW2 is expected to cause traffic to be automatically rerouted through SW1-SW3-SW4-SW2 as depicted in Figure 19, in Section 3.2.2. The link failure causes connectivity between hosts A1 and A2 to go down and a new route is apparently established by OVX, as observed in the edited logs below:

```
Try recovery for virtual network A big-switch X between ports(1,2)
    in virtual network A switching all existing flow-mods crossing
    the big-switch X route 1 between ports (1,2) to the new path:
        [SW1/4 <-> SW3/4, SW3/5 <-> SW4/5, SW4/4 <-> SW2/4]
Try recovery for virtual network A big-switch X between ports(2,1)
    in virtual network A switching all existing flow-mods crossing
    the big-switch X route 1 between ports (2,1) to the new path:
```

```
        [SW2/4 <-> SW4/4, SW4/5 <-> SW3/5, SW3/4 <-> SW1/4]
virtual network A, switch X, route 1 between ports 1-2:
    flow-mod switched to the new path
virtual network A, switch X, route 1 between ports 2-1:
    flow-mod switched to the new path
Removing physical link between SW2/5 and SW1/5
Removing physical link between SW1/5 and SW2/5
```

However, when the link between SW1 and SW2 is deactivated, connectivity is not recovered and the experiment fails. The *dpctl dump-flows* command executed in Mininet reveals that one of the old flow rules using the link between SW1 and SW2 was not flushed from SW1's flow table:

```
mininet> dpctl dump-flows
*** SW1 -------------------------------------------------------------
[...],in_port=1,dl_src=02:00:00:00:01:01,dl_dst=02:00:00:00:01:02,
        nw_src=10.0.0.1,nw_dst=10.0.0.2
        actions=mod_nw_dst:1.0.0.2,mod_nw_src:1.0.0.1,
        output:5
```

This rule means that SW1 keeps trying to use the link at port 5, which was deactivated. The rule never expires by itself from the flow table because host A1 keeps sending traffic and trying to reach A2. In order to have traffic successfully routed through the backup route, OVX was expected to flush the flow rules of the original route and install new ones through the backup route. The only way to work around this is to stop traffic from A1 to A2, wait until the remaining flow rule expires, and

start it again. This allows the new flow to be processed by OVX and use the backup route.

Therefore, this experiment is considered a failure. Network connectivity should not be interrupted indefinitely if the hosts continue attempting to use the network. It is clear that OVX failed to flush the flow tables completely in order to install the new backup route flow rules.

As mentioned before, autonomous rerouting tests with VeRTIGO were not concluded because of stability issues.

4.1.3 Transparent Traffic Forwarding

Using the *test_packets.py* tool, many types of traffic were tested to evaluate whether the virtualization controllers would block specific frames. The summarized results are shown in Table 2 and classified into six different categories. All the categories are self explanatory, and more detail about which packets were used for testing can be found in [51]. Table 2 compares the results of traffic forwarding using a standard layer 2 switch (L2 column), Open vSwitch (OVS), Floodlight (FL), FlowVisor (FV) and OpenVirteX (OVX). Note that VeRTIGO results are not shown here because VeRTIGO's application would run into exceptions and stop working during tests.

Table 2. Summarized results of traffic forwarded with network virtualization.

Type of traffic	**L2**	**OVS**	**FL**	**FV**	**OVX**
IPv4 unicast	OK	OK	OK	OK	OK
IPv6 unicast	OK	OK	OK	No	No
IPv4 multicast	OK	OK	OK	OK	OK
L2 multicast	Some	No	Some	Some	Some
IPv4 w/ VLAN	OK	OK	OK	OK	OK
IPv4 w/ MPLS	OK	OK	OK	OK	OK

Both OVX and FlowVisor network hypervisors managed to forward everything, except:

- IPv6 packets;
- Two types of L2 Multicast packets: link layer discovery protocol (LLDP) and spanning tree protocol (STP).

The Floodlight controller supported IPv6 forwarding, though it had to be configured with OpenFlow 1.3, which is not supported by FlowVisor and OpenVirteX. However, Floodlight still cannot forward LLDP and STP frames. This limitation is expected, as these frames are intercepted by the controller's application for network discovery purposes or processed/dropped by the Open vSwitches.

Open vSwitch configured as a traditional layer 2 switch successfully manages to forward all types of traffic, except L2 multicast packets such as LLDP, STP, link aggregation protocol (LACP), and Cisco discovery protocol (CDP). These frames use reserved multicast MAC destination addresses in the ranges 01:80:c2:00:00:xx and 01:00:0c:cc:cc:xx, and should be normally processed by Ethernet bridges. With OVS configured as a standalone bridge, it is acceptable to have these frames processed or dropped instead of flooded to the rest of the network.

A standard layer 2 switch – a NETGEAR Fast Ethernet FS108 switch – was also used as a baseline to compare against the other results. It managed to forward all types of traffic except a few of the L2 multicast frames:

- Operations, Administration, and Maintenance (OAM), a protocol used to detect connectivity problems in layer 2 networks;
- LACP;

In general, the tested network hypervisors were surprisingly good in forwarding several types of traffic. The only general problems observed happen with IPv6, LLDP and STP traffic. IPv6 is not supported by OpenFlow 1.0, which is used by both OVX and FlowVisor. LLDP and STP are both used as control protocols for Ethernet networks. Although this limitation is expected because these protocols are meant to be processed by network hypervisors, this is a potentially serious limitation for customers who wish to run their own layer 2 control protocols in their virtual networks.

The resulting capture files gathered during this experiment are available in [52] in the pcapng format, which can be opened by Wireshark [65].

4.1.4 Compliance with Addressing Standards

4.1.4.1 Unicast IPv4 Header Rewriting

Regarding the header rewriting technique used by OVX, the results show that the network hypervisor replaces IPv4 addresses with addresses in the ranges X.0.0.0/8 for each tenant host, where X is the tenant ID assigned during OVX configuration. Figure 28 shows a screenshot of a packet capture of an ICMP echo request from host A1 to A2, and a reply from A2 to A1. By monitoring network traffic at port 5 of SW2, we can observe that the IP address 10.0.0.1 is rewritten as 1.0.0.1 and the IP address 10.0.0.2 as 1.0.0.2.

It is interesting to notice that if internal and external IP addresses are the same, OVX still installs flow rules that rewrite the IP header. For example, when host C1 (3.0.0.1) is sending ICMP echo requests to host C2 (3.0.0.2), OVX rewrites their

Figure 28. IP header rewriting done by OVX.

Figure 29. IP header rewriting done by OVX with same IP addresses.

addresses as 3.0.0.1 and 3.0.0.2 respectively, as shown in Figure 29. This is done because hosts C1 and C2 belong to tenant ID 3 and their internal IP addresses lie in the range 3.0.0.0/8. This pointless address rewriting can be confirmed by the following flow rules installed by OVX in SW1's flow table:

```
[...],in_port=3,dl_src=02:00:00:00:03:01,dl_dst=02:00:00:00:03:02,
                  nw_src=3.0.0.1,nw_dst=3.0.0.2
                  actions=mod_nw_dst:3.0.0.2,mod_nw_src:3.0.0.1,
                  output:5
```

```
[...],in_port=5,dl_src=02:00:00:00:03:02,dl_dst=02:00:00:00:03:01,
                nw_src=3.0.0.2,nw_dst=3.0.0.1
                actions=mod_nw_src:3.0.0.2,mod_nw_dst:3.0.0.1,
                output:3
```

The repeated address spaces in the internal and external networks managed by OVX could make troubleshooting much harder in case packets leak from the internal network, but they actually do not cause any problems in the traffic forwarding capabilities of the virtual network. However, the internal addressing scheme of OVX goes against Internet addressing best practices [53] and cannot be used outside of completely private IPv4 networks. If any traffic leaks from the internal network to the Internet, it could potentially carry sensitive data all the way to an unknown destination.

FlowVisor does not have any issues with addressing standards compliance because it does not modify tenant's traffic, but rather ensures that each tenant owns a limited portion of the flow space.

4.1.4.2 ARP and Multicast Header Rewriting

Regarding ARP, IPv4 multicast packets and MAC multicast frames, OVX did not present any addressing compliance problems to forward these frames across the virtual network. In fact, OVX did not even forward these packets across the data forwarding plane of the network. All ARP frames, IPv4 multicast packets (destination address 224.0.0.5 tested) and MAC multicast addresses (destination address 01:00:0c:cc:cc:cc tested) are always forwarded through the control plane of the network. This is shown in Figure 30, where each PACKET_IN of type ARP or with multicast destinations is

automatically flooded to all switches through OVX by using PACKET_OUT messages. This means that the header rewriting method employed by OVX does not apply to these situations.

```
 6633 17.369519000 02:00:00:00:01:01 Broadcast             OpenFlow   126   Type: OFPT_PACKET_IN
44009 17.371246000 02:00:00:00:01:01 Broadcast             OpenFlow   132   Type: OFPT_PACKET_OUT
44010 17.371546000 02:00:00:00:01:01 Broadcast             OpenFlow   132   Type: OFPT_PACKET_OUT
44013 17.371586000 02:00:00:00:01:01 Broadcast             OpenFlow   132   Type: OFPT_PACKET_OUT
44011 17.371595000 02:00:00:00:01:01 Broadcast             OpenFlow   132   Type: OFPT_PACKET_OUT
 6633 17.707480000 10.0.0.1          224.0.0.5             OpenFlow   182   Type: OFPT_PACKET_IN
44009 17.710168000 10.0.0.1          224.0.0.5             OpenFlow   188   Type: OFPT_PACKET_OUT
44011 17.710202000 10.0.0.1          224.0.0.5             OpenFlow   188   Type: OFPT_PACKET_OUT
44013 17.710208000 10.0.0.1          224.0.0.5             OpenFlow   188   Type: OFPT_PACKET_OUT
44010 17.710551000 10.0.0.1          224.0.0.5             OpenFlow   188   Type: OFPT_PACKET_OUT
 6633 17.978632000 02:00:00:00:01:01 CDP/VTP/DTP/PAgP/UDLD OpenFlow   472   Type: OFPT_PACKET_IN
44009 17.980015000 02:00:00:00:01:01 CDP/VTP/DTP/PAgP/UDLD OpenFlow   478   Type: OFPT_PACKET_OUT
44013 17.980001000 02:00:00:00:01:01 CDP/VTP/DTP/PAgP/UDLD OpenFlow   478   Type: OFPT_PACKET_OUT
44010 17.980035000 02:00:00:00:01:01 CDP/VTP/DTP/PAgP/UDLD OpenFlow   478   Type: OFPT_PACKET_OUT
44011 17.980405000 02:00:00:00:01:01 CDP/VTP/DTP/PAgP/UDLD OpenFlow   478   Type: OFPT_PACKET_OUT

▸ Frame 875: 182 bytes on wire (1456 bits), 182 bytes captured (1456 bits) on interface 0
▸ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
▸ Internet Protocol Version 4, Src: 127.0.0.1 (127.0.0.1), Dst: 127.0.0.1 (127.0.0.1)
▸ Transmission Control Protocol, Src Port: 44012 (44012), Dst Port: 6633 (6633), Seq: 3699, Ack: 4559, Len: 116
▾ OpenFlow 1.0
    .000 0001 = Version: 1.0 (0x01)
    Type: OFPT_PACKET_IN (10)
    Length: 116
    Transaction ID: 0
    Buffer Id: 0xffffffff
    Total length: 98
    In port: 1
    Reason: No matching flow (table-miss flow entry) (0)
    Padding: 0
  ▸ Ethernet II, Src: 02:00:00:00:01:01 (02:00:00:00:01:01), Dst: IPv4mcast_05 (01:00:5e:00:00:05)
  ▸ Internet Protocol Version 4, Src: 10.0.0.1 (10.0.0.1), Dst: 224.0.0.5 (224.0.0.5)
  ▸ Open Shortest Path First
```

Figure 30. ARP, IP and MAC multicast flooded by OVX.

Although OVX and FlowVisor do not have any compliance problems with the ARP and IP/MAC multicast frames tested, the results show that both of them may cause problems in multicast networks. Every single multicast frame sent to the network is handled by the network hypervisor alone, without ever forwarding it to the Floodlight tenant controller.

4.2 Performance Test Results

Using the physical topology (Section 3.1.2), performance tests were severely affected by the switch's inability to install required FLOW MOD rules in hardware tables. Due to hardware limitations, the OpenFlow switch could use only software flow tables. This causes a huge performance impact in test results, especially in throughput tests. Better results were achieved with the GENI testbed described in Section 3.1.3.

For all the box plots shown in the following sections, outliers (data points represented by circles) were discarded when computing statistics. Outliers are defined as data points that fall outside the boundaries defined by Equations 4.1 and 4.2 for each data set. Q_1 represents the first quartile, Q_3 represents the third quartile, and IQR represents the inter quartile range of each data set.

$$Lower\ boundary = Q_1 - 1.5 \times IQR \tag{4.1}$$

$$Upper\ boundary = Q_3 + 1.5 \times IQR \tag{4.2}$$

4.2.1 Flow Setup Time

Latency tests were only possible for FlowVisor and OVX. VeRTIGO caused the first packets of a flow to be dropped during flow installation. Therefore the ping test could not measure flow setup time for VeRTIGO.

This experiment was conducted in the physical topology described in Section 3.1.2, and verified flow setup time cases for:

- Floodlight;

- FlowVisor and Floodlight;
- No controller, using the switch in traditional layer 2 forwarding mode;
- OVX and Floodlight.

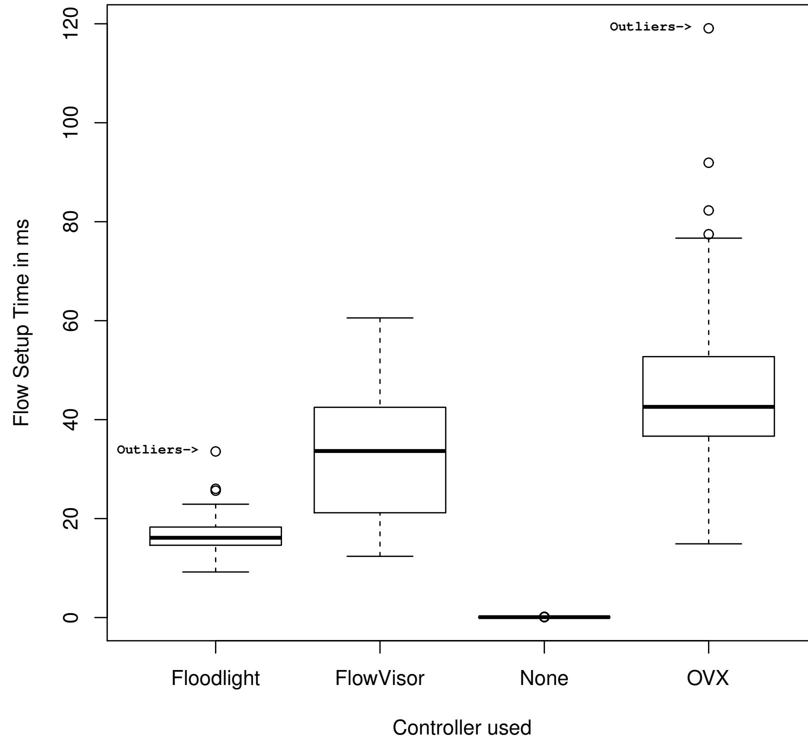

Figure 31. Round trip time of first ping.

As shown in Figure 31, flow setup time with Floodlight alone was on average under 20ms, while FlowVisor flow setup time floated around 20-40ms and OVX flow setup time around 40-50ms. The variance of the flow setup time was higher in OVX case, and this result is probably due to the fact that OVX relies on complex flow rules to rewrite traffic as it enters or exits the network. A standalone layer 2 switch yielded, on

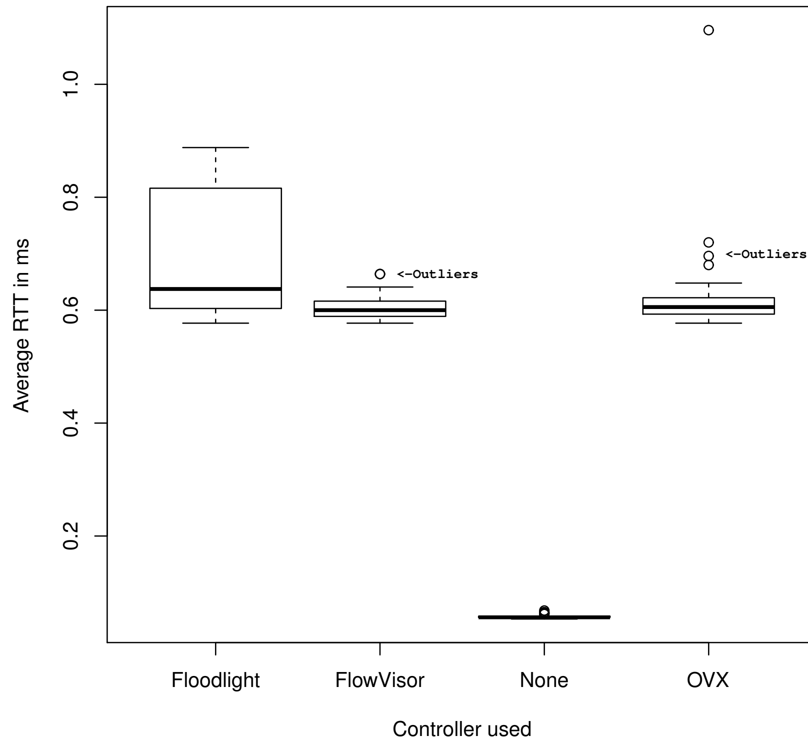

Figure 32. Average round trip time of subsequent pings.

average, 0.08ms of flow setup time. This is represented by the “None” controller. These results were according to expectations: the flow setup time is much higher when using network hypervisors between the network and the tenant controllers, and software defined networks show a significant overhead in flow setup time when compared with traditional networks.

Figure 32 shows that the average RTT of pings initiated after flow setup does not depend on the controller at all. This is expected after the flows are installed, as the controller no longer has any effect on the latency between hosts. Furthermore, it is in this plot that the difference between the software and hardware forwarding modes can be observed. The standalone forwarding mode of the switch (Controller = None)

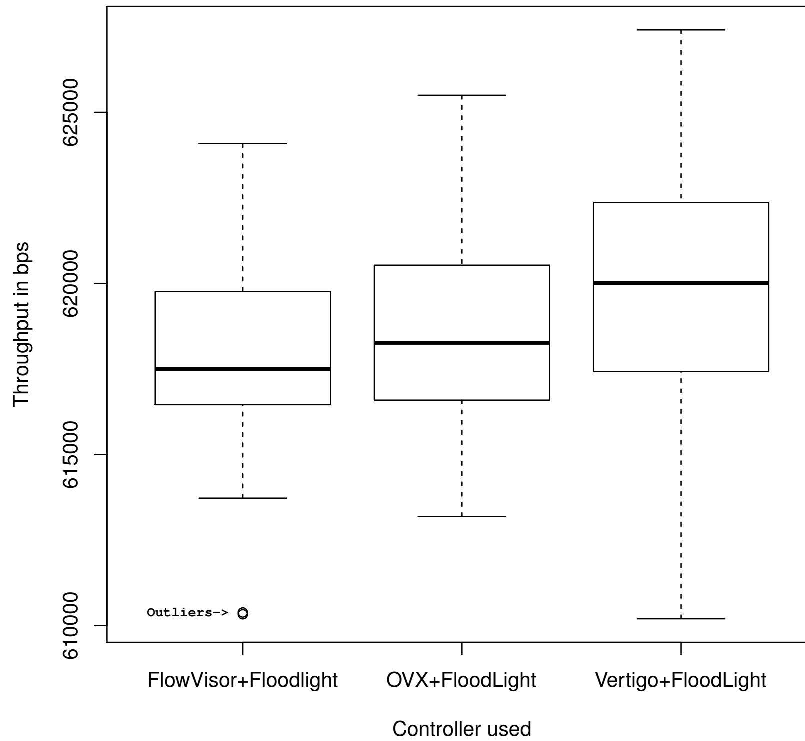

Figure 33. Throughput measured with different controllers, in bits per second.

takes advantage of hardware line rate forwarding, which results in approximately 10 times faster RTTs and a much lower variance.

4.2.2 Throughput

The throughput test is based on iperf in TCP mode. TCP is a best-effort protocol and the throughput achieved depends on network latency, frame size, packet loss, inter-frame gap, and other network conditions [10]. The expected throughput in a low latency local area network should be within the 90-100% range of the auto-negotiated

link capacity of 100Mbps. However, test results were affected by the poor performance of the HP-3500yl-24G switch, and measured throughput was less than 1 Mbps.

The results of the experiments conducted in the physical topology are summarized in Figure 33. The bandwidth with FlowVisor averages to ≈617.9kbps, with OVX ≈618.6kbps and VeRTIGO ≈620.0kbps. However, the variance in all the results is quite large, and statistically there is no significant difference in the resulting bandwidth measured between two hosts when changing to different network hypervisors. This result was expected, since the bottleneck of this experiment was the switch processor. High switch CPU load during the experiment is the most likely reason to have an outlier data point on the "FlowVisor + Floodlight" data set. Simple tasks during the experiment, such as opening a telnet session to the switch, could affect throughput, because they require the CPU to stop handling traffic temporarily.

Additional research and tests confirmed that the performance limit was imposed by the HP-3500yl-24G limited hardware [41]. Browsing the flow tables of the switch also revealed that even the most simple flow rules were not handled by the forwarding hardware, but by its CPU:

```
HP-3500yl-24G# show openflow instance ovx flows
<output omitted for clarity>
      Flow Location : Software
      Hardware Index: NA
      Reason Code        : 2
      Reason Description : The rule has a match criterion for MAC address
<output omitted for clarity>
```

The "Flow Location" field indicates that the flow rule is handled by software, and the reason for handling in software is "The rule has a match criterion for MAC address".

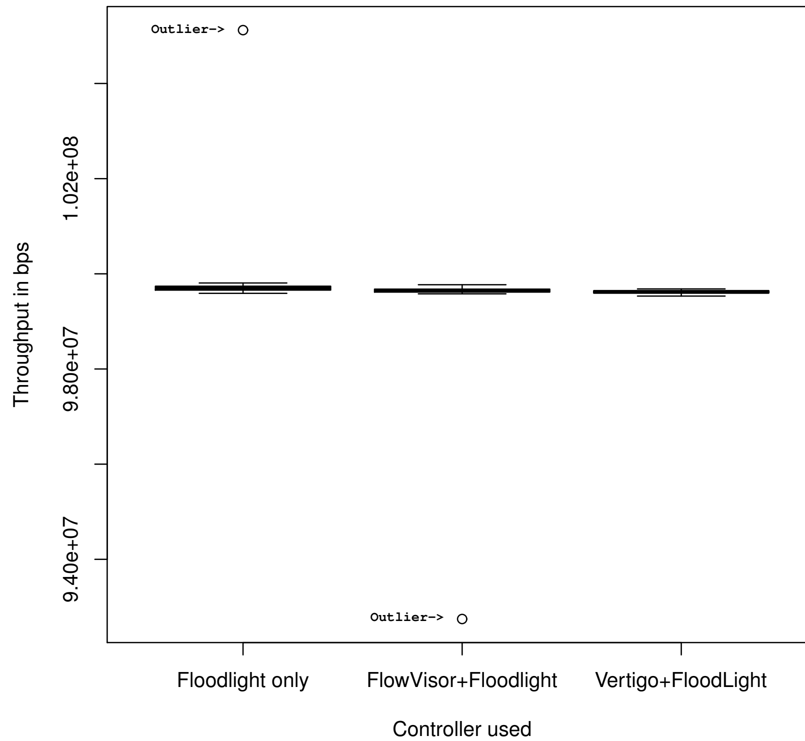

Figure 34. Throughput results in GENI testbed, in bits per second

This means that the HP-3500yl-24G switch cannot even handle the most basic flow rules in hardware.

The throughput tests were repeated using a GENI hardware switch and the results were much more satisfactory. Figures 34 and 35 show the results. While Floodlight, FlowVisor and VeRTIGO managed to install flows that yielded nearly 100Mbps throughput in iperf TCP tests between hosts, OpenVirteX traffic throughput averaged to approximately 339kbps. Again, TCP is a best-effort protocol and tries to use as much bandwidth as it can. Throughput for this experiment should also stay between 90-100% of link capacity (100Mbps).

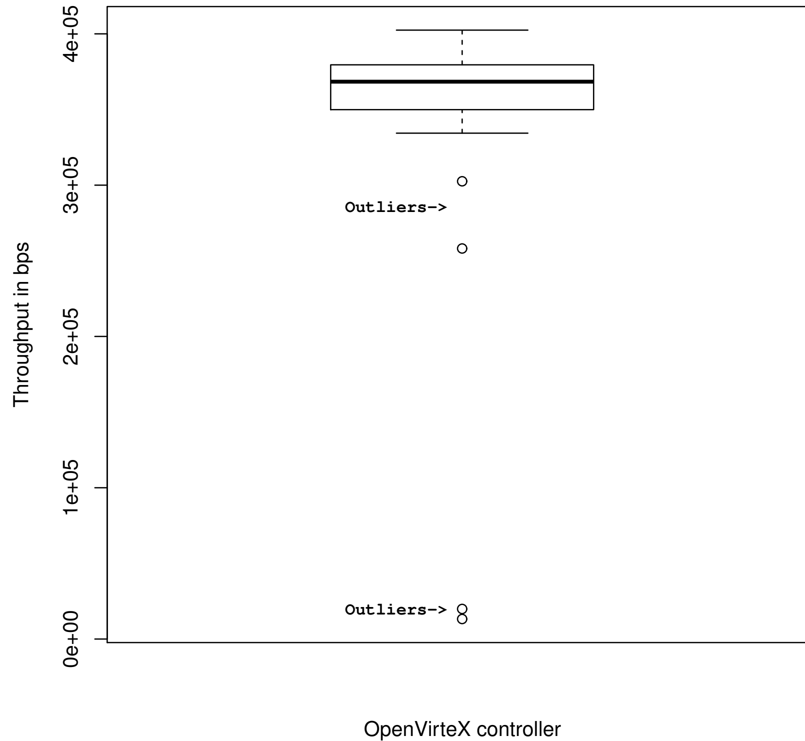

Figure 35. Throughput results in GENI testbed, for OVX, in bits per second

Figure 34 also shows two outlier data points. The first outlier is in the "Floodlight only" column, a data point which exceeds the link capacity of 100Mbps. This value is not realistic, and the reason for the anomaly is unknown. The second outlier occurred with "FlowVisor+Floodlight", and it shows that one measurement was significantly lower than the rest. This could mean several different things, such as temporary CPU overload in the computer running iperf, or temporary packet loss could have triggered TCP's congestion control mechanism. Fortunately, the outliers were few in number, and the results are still considered valid.

GENI switches cannot be logged into to confirm the reason for such a low throughput with OVX, but most likely the relatively complex header rewriting flow rules

required by OVX cannot be handled by this switch's hardware. The outlier data points in Figure 34 also indicate that the switch was overwhelmed by the TCP flow while processing packets with its slow CPU.

4.3 Results Summary and Analysis

Two out of the three network virtualization applications were successfully evaluated by the experiments proposed here. Both FlowVisor and OVX lived up to the expectations and, for most of the experiments, worked well according to their documentation. Most of the experiments with VeRTIGO could not be completed due to limited documentation and interoperability issues with OVS and Floodlight. VeRTIGO presented several limitations and was usable only for very simple tests with the hardware switch. Attempts to use virtual links to allow different virtual topologies caused exceptions in Floodlight when exchanging messages with VeRTIGO.

Regarding network and topology isolation, FlowVisor implements a simpler flavor of network virtualization when compared to OVX. On one hand, FlowVisor allows the slices to be split based on very flexible constraints, such as a whole switch, physical ports, MAC addresses or TCP/UDP port numbers. On the other hand, OVX does not allow any traffic in the network without explicitly "connecting" a host to one of its virtual networks by supplying a MAC address and Virtual Port. This means that the administrator of the network – or the application controlling it – needs to know in advance all of the MAC addresses that are going to be connected to the network.

FlowVisor does not care whether the network traffic uses IP or not, as long as the network controller is able to handle network traffic exposed through FlowVisor's slicing mechanism. OVX, based on its current operation mechanism, assumes that

the networks under control will operate with IPv4. Both network virtualization tools, however, are limited by the OpenFlow capabilities that they support. For instance, currently FlowVisor and OVX do not support IPv6 because they can only handle OpenFlow 1.0, while most of the IPv6 implementations are only offered by versions 1.2 and 1.3 [40]. This is made evident by the experimental results presented in Section 4.1.3.

Both VeRTIGO and OVX claimed to have support for autonomous rerouting in case of link failure. VeRTIGO was not verifiable due to compatibility issues. OVX autonomous rerouting feature worked, but still cannot guarantee autonomous recovery if the client host does not let flow rules expire before trying to send traffic again. It is not possible to measure the network recovery time, because the rerouting feature depends on network usage.

On the subject of addressing standards, FlowVisor's transparent mode of operation makes sure that it is prepared to handle almost any kind of addressing, as long as it is supported by OpenFlow and the controlling application. OVX, however, relies almost exclusively on rewriting of the IPv4 header to provide network isolation features. While it works very well in the networks tested here, this method could be a problem in an IPv6 network, because headers would have to be rewritten from IPv6 to IPv4 and vice versa. This could lead to many compatibility problems. Another negative aspect of assigning arbitrary networks for header rewriting is that the isolated traffic could cause problems if leaked to the Internet. For instance, OVX uses internal networks such as 1.0.0.0/8 and 2.0.0.0/8 for isolation, but these are valid public IP ranges and should never be used in private networks.

Interestingly, the experiment intended to verify compliance to addressing standards was useful to find some serious limitations in the network virtualization layer proposed

by both FlowVisor and OVX. Experiments with ARP and multicast frames revealed that the network hypervisors under test handled these frames by always forwarding them through the control plane, instead of installing flow rules to deal with this kind of traffic. This could become a major bottleneck in multicast intensive networks, as every ARP or IP/MAC multicast frame has to be processed by the controller before being sent out to the network. Additionally, the virtualization layer imposed by the network hypervisors does not forward these frames to tenant controllers. Instead, network hypervisors process these frames locally, leaving the tenant controllers unaware that there ever was a multicast frame travelling across the network. This means that the tenant controller is limited by the virtualization layer, even if it is programmed to support multicast.

Concerning flow setup time, tests show that the introduction of network hypervisors increases this time considerably, as it is expected when adding additional processing stages to setup and install new flows. The incorporation of extra network virtualization controller layers merely highlights an already existing concern of SDN. Flow setup time is already recognized as one of OpenFlow's (and SDN) disadvantages of having a centralized controller taking care of forwarding decisions [62]. Traditional distributed layer 2 networks have a very low flow setup time, frequently a few milliseconds or even less than one millisecond. This is a particularly important aspect for networks that require quick recovery from failure.

Throughput tests revealed that network virtualization proxy controllers may decrease throughput significantly if the installed flows are not supported by the device's hardware. In the experiments carried out in the local physical laboratory, none of the network applications was supported by the OpenFlow hardware, and all controllers performed poorly due to limitations of the OpenFlow switch. However,

results of experiments using a GENI hardware switch show that only OpenVirteX virtualization caused a significant performance impact in throughput. Although the reason for OpenVirteX throughput limitation could not be confirmed, this problem is most likely related to complex header rewriting flows that OpenVirteX requires.

Chapter 5

CONCLUSION

This work has presented a literature review of traditional and SDN based network virtualization technologies. It also presented experiments with some of the latest SDN based options available to the public. Traditional technologies such as those based on the use of VLANs, MPLS and VxLAN have years of development and standardization which translate into relatively stable solutions. They are still useful to solve most of the virtualization challenges required by Internet service providers (ISPs) and data centers. SDN, however, brings more flexibility to the virtualization problem.

FlowVisor allows the network administrator to slice the network using many different matching rules, allowing multiple different applications to share the same network. This kind of flexibility is hard to achieve in traditional networks. For instance, once a host is connected to a port using a VLAN, there are not many ways to give different treatment to all the traffic that is tagged with the same VLAN ID. With FlowVisor, different treatment could be easily given to different TCP or UDP application ports. OVX provides a virtualization service similar to what could be achieved with a MPLS based VPN. All hosts belonging to the same tenant are assigned to a single virtual network, and OVX calculates the best routes within the network to interconnect these hosts, as if the whole network was a big switch. This solution is similar to a VPLS VPN service, offered by MPLS networks. From the tenant's point of view the network appears to be a single big switch. The advantage of OVX is that the configuration and management of the network is completely centralized, whereas in a traditional network each switch would have to be configured separately.

Based on the observations made throughout this work, FlowVisor is best suited to handle different applications within the same network. It can be used as a means of virtualizing a network to allow different customers to share layer 2 forwarding resources, providing isolation between slices. However, its true potential is to provide each slice with a different application. This means that a set of OpenFlow switches could potentially be used to provide a layer 2 based forwarding solution to one customer while providing IP routing to another customer. On the other hand, OVX is mostly focused on network isolation. It is very well suited to be used in data center networks, as its virtualization mechanism relies on the information about port and MAC addresses to determine to which network a host belongs. Although OVX by itself requires manual configuration of all the hosts of the network, there are development efforts which aim to integrate OVX with OpenStack, enabling the provisioning of virtual machines along with the provisioning of virtual networks with OVX through the neutron plug-in. This is essentially the infrastructure as a service (IaaS) concept [44].

Even though the SDN approach is flexible, it still has some disadvantages. As seen in the performance test results, flow setup time is already high in SDN networks and further increased when the network virtualization proxy controller is introduced. The open source solutions available and tested here are in the early stages of development and may suffer from bugs and interoperability issues, as was observed in experiments with VeRTIGO or the OVX autonomous rerouting feature. Another negative aspect of SDN is that the virtualization methods are not standardized at all. For instance, if there is ever a need to interconnect two networks from different providers using different virtualization controllers (e.g. FlowVisor and OVX), special care would have

to be taken when handling traffic at the edges of the network to ensure that networks would remain isolated properly.

When faced with processing and transparent forwarding of special types of traffic, both FlowVisor and OVX presented serious limitations. None of these network hypervisors is able to forward IPv6. Some of the layer 2 control protocols are blocked by the network virtualization layer, and multicast applications have to rely on CPU intensive control plane forwarding. By restricting the types of traffic that tenants can use in the network, the currently available SDN-based virtualization techniques discourage migration of existing applications to the new paradigm of SDN.

The creative approach of OVX to use IP header rewriting to provide network virtualization works surprisingly well in experimental environments. However, the indiscriminate use of arbitrary IP addresses to implement network virtualization is questionable. Rewriting IP headers makes the network troubleshooting process more complex, and the risks involved with the use of valid public IP addresses to implement virtualization may not be worth the cost of potential leaks of sensitive data to the Internet.

5.1 Future Work

5.1.1 Development of Network Hypervisors

The OpenVirteX approach to network virtualization is promising. If OVX could support MPLS through OpenFlow version ≥ 1.2, the software could be adapted to provide different services, such as MPLS based VPNs. This would contribute towards employing SDN in a service provider use case, instead of only using it in data centers.

In addition, this would eliminate all the problems related to using the header rewriting method for virtualization.

Additionally, OVX deployed by itself requires the administrator to program all the hosts (MAC addresses and ports) in order to configure the virtual networks. This could quickly become a cumbersome task even in small networks. According to De Leenheer [13], it should be straightforward to add host discovery capabilities to OVX. However, even if OVX could dynamically learn and add new hosts to the network, how would it decide to which network each host belongs? This is an interesting research area to be further explored.

5.1.2 IPv6 Support

Current SDN related research gives a surprising amount of focus to IPv4 and solving problems that would not exist with IPv6, such as making the full header space available to tenants. With IPv6, addresses would not need to be reused and network header space limits should no longer be a concern. IPv6 is already widely supported and stable in most operating systems. As OpenFlow 1.3 becomes popular and more widely available in network devices, there is a clear opportunity for research on support for IPv6 in network hypervisors.

5.1.3 Improved Experiments

The performance experimental results done in this work were partially inconclusive. It would be interesting to experiment with network virtualization in bigger physical networks with more OpenFlow switches that support flows in hardware. Using the

GENI [21] infrastructure to repeat all of the experiments could be a next step to improve the quality and reproducibility of the experiments presented in this work.

Additionally, more standardized tools could be used to measure and compare the network hypervisor's performance impact. Although, the ping based flow setup time experiment used in this work is useful in representing a "real user" point of view of controller performance, the flow setup time could have been tested with cbench, a tool that is better suited to produce comparable experimental results and benchmarking of OpenFlow controllers [58].

REFERENCES

[1] A. Al-Shabibi et al. (Aug. 2013). Flowvisor @ github.com, [Online]. Available: https://github.com/opennetworkinglab/flowvisor (visited on 11/25/2015).

[2] ——, (May 2014). Openvirtex installation @ ovx.onlab.us, [Online]. Available: http://ovx.onlab.us/getting-started/installation/ (visited on 11/25/2015).

[3] ——, (May 2014). OpenVirteX Repository, [Online]. Available: https://github.com/OPENNETWORKINGLAB/OpenVirteX.git (visited on 03/23/2016).

[4] "LDP specification," Tech. Rep., Oct. 2007. DOI: 10.17487/rfc5036. [Online]. Available: https://tools.ietf.org/html/rfc5036.

[5] "Framework for Layer 2 Virtual Private Networks (L2VPNs)," Tech. Rep., Sep. 2006. DOI: 10.17487/rfc4664. [Online]. Available: https://tools.ietf.org/html/rfc4664.

[6] D. Awduche, L. Berger, D. Gan, T. Li, V. Srinivasan, and G. Swallow, "RSVP-TE: Extensions to RSVP for LSP tunnels," Tech. Rep., Dec. 2001. DOI: 10.17487/rfc3209. [Online]. Available: https://tools.ietf.org/html/rfc3209.

[7] A. N. Bhagat. (n.d.). Understanding PBB, [Online]. Available: https://sites.google.com/site/amitsciscozone/home/pbb/understanding-pbb (visited on 03/11/2016).

[8] D. Bombal. (n.d.). Datapath IDs, [Online]. Available: http://pakiti.com/datapath-ids/ (visited on 04/01/2016).

[9] Z. Bozakov and P. Papadimitriou, "Autoslice: Automated and scalable slicing for software-defined networks," in *Proceedings of the 2012 ACM Conference on CoNEXT Student Workshop*, ser. CoNEXT Student '12, Nice, France: ACM, 2012, pp. 3–4. DOI: 10.1145/2413247.2413251. [Online]. Available: http://doi.acm.org/10.1145/2413247.2413251.

[10] S. Bradner and J. McQuaid, "Benchmarking Methodology for Network Interconnect Devices," Tech. Rep., Mar. 1999. DOI: 10.17487/rfc2544. [Online]. Available: https://www.ietf.org/rfc/rfc2544.txt.

[11] N. M. M. K. Chowdhury and R. Boutaba, "A survey of network virtualization," *Computer Networks*, vol. 54, no. 5, pp. 862–876, 2010. DOI: http://dx.doi.org/10.1016/j.comnet.2009.10.017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1389128609003387.

[12] R. D. Corin and M. Gerola. (Nov. 2013). Vertigo @ github.com, [Online]. Available: https://github.com/fp7-ofelia/VeRTIGO (visited on 11/25/2015).

[13] M. De Leenheer. (Dec. 2015). Openvirtex discussion forum - can ovx discover the hosts information? [Online]. Available: https://goo.gl/hL8yTe (visited on 03/11/2016).

[14] S. Deering and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification," Tech. Rep., Dec. 1998. DOI: 10.17487/rfc2460. [Online]. Available: https://tools.ietf.org/html/rfc2460.

[15] R. Doriguzzi Corin, M. Gerola, R. Riggio, F. De Pellegrini, and E. Salvadori, "Vertigo: Network virtualization and beyond," in *Software Defined Networking (EWSDN), 2012 European Workshop on*, Oct. 2012, pp. 24–29. DOI: 10.1109/EWSDN.2012.19.

[16] D. Drutskoy, E. Keller, and J. Rexford, "Scalable Network Virtualization in Software-Defined Networks," *Internet Computing, IEEE*, vol. 17, no. 2, pp. 20–27, Mar. 2013. DOI: 10.1109/MIC.2012.144.

[17] D. Erickson, "The Beacon OpenFlow Controller," in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '13, Hong Kong, China: ACM, 2013, pp. 13–18. DOI: 10.1145/2491185.2491189. [Online]. Available: http://doi.acm.org/10.1145/2491185.2491189.

[18] D. Farinacci, T. Li, S. Hanks, D. Meyer, and P. Traina, "Generic Routing Encapsulation (GRE)," Tech. Rep., Mar. 2000. DOI: 10.17487/rfc2784. [Online]. Available: https://tools.ietf.org/html/rfc2784.

[19] Floodlight. (n.d.). Floodlight project, [Online]. Available: http://www.projectfloodlight.org/ (visited on 03/23/2016).

[20] ——, (n.d.). Floodlight repository, [Online]. Available: https://github.com/floodlight (visited on 03/23/2016).

[21] GENI. (n.d.). Global Environment for Network Innovations (GENI), [Online]. Available: http://www.geni.net/ (visited on 03/23/2016).

[22] R. R. Hain. (Mar. 2015). Intro to openflow tutorial (hardware switch), [Online]. Available: http://groups.geni.net/geni/wiki/GENIExperimenter/Tutorials/OpenFlowHW/DesignSetup (visited on 03/24/2016).

[23] "IEEE Standard for Ethernet," *IEEE Std 802.3-2012 (Revision to IEEE Std 802.3-2008)*, pp. 1–3747, Dec. 2012. DOI: 10.1109/IEEESTD.2012.6419735.

[24] "IEEE Standard for Local and Metropolitan Area Networks – Bridges and Bridged Networks," *IEEE Std 802.1Q-2014 (Revision of IEEE Std 802.1Q-2011)*, Dec. 2014. DOI: 10.1109/IEEESTD.2014.6991462.

[25] "IEEE Standard for Local and metropolitan area networks: Media Access Control (MAC) Bridges," *IEEE Std 802.1D-2004 (Revision of IEEE Std 802.1D-1998)*, pp. 1–277, Jun. 2004. DOI: 10.1109/IEEESTD.2004.94569.

[26] "IEEE Standard for Local and Metropolitan Area Networks – Virtual Bridged Local Area Networks, Amendment 4: Provider Bridges," *IEEE Std 802.1ad-2005 (Amendment to IEEE Std 8021Q-2005)*, 2006. DOI: 10.1109/IEEESTD.2006.216360.

[27] "IEEE Standard for Local and Metropolitan Area Networks – Virtual Bridged Local Area Networks Amendment 7: Provider Backbone Bridges," *IEEE Std 802.1ah-2008 (Amendment to IEEE Std 802.1Q-2005)*, Aug. 2008. DOI: 10.1109/IEEESTD.2008.4602826.

[28] Internet Assigned Numbers Authority. (Mar. 2015). Ethernet Numbers Assignment, [Online]. Available: http://www.iana.org/assignments/ethernet-numbers/ethernet-numbers.xhtml (visited on 04/02/2016).

[29] ——, (Mar. 2016). IPv4 Multicast Address Space Registry, [Online]. Available: http://www.iana.org/assignments/multicast-addresses/multicast-addresses.xhtml (visited on 04/02/2016).

[30] A. A. Jaha, F. B. Shatwan, and M. Ashibani, "Proper Virtual Private Network (VPN) Solution," in *2008 The Second International Conference on Next Generation Mobile Applications, Services, and Technologies*, Institute of Electrical & Electronics Engineers (IEEE), 2008. DOI: 10.1109/ngmast.2008.18.

[31] Juniper. (May 2014). Understanding Q-in-Q Tunneling and VLAN Translation, [Online]. Available: http://www.juniper.net/documentation/en_US/junos13.2/topics/concept/qinq-tunneling-qfx-series.html (visited on 02/14/2016).

[32] S. Kent and K. Seo, "Security Architecture for the Internet Protocol," Tech. Rep., Dec. 2005. DOI: 10.17487/rfc4301. [Online]. Available: https://tools.ietf.org/html/rfc4301.

[33] D. Kreutz, F. M. V. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: a comprehensive survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, Jan. 2015. DOI: 10.1109/JPROC.2014.2371999. eprint: 1406.0440. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6994333.

[34] M. Lewis, *Comparing, Designing, and Deploying VPNs*. Cisco Press, 2006. [Online]. Available: http://ptgmedia.pearsoncmg.com/images/1587051796/samplechapter/1587051796content.pdf (visited on 03/12/2016).

[35] M. Mahalingam, D. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright, "Virtual eXtensible local area network (VXLAN): A framework for overlaying virtualized layer 2 networks over layer 3 networks," Tech. Rep., Aug. 2014. DOI: 10.17487/rfc7348. [Online]. Available: https://tools.ietf.org/html/rfc7348.

[36] Mininet. (Feb. 2016). Mininet overview, [Online]. Available: http://mininet.org/overview/ (visited on 01/20/2016).

[37] ——, (n.d.). Mininet GitHub, [Online]. Available: git://github.com/mininet/mininet (visited on 01/23/2016).

[38] R. Molenaar. (2014). 802.1Q Tunneling (Q-in-Q) Configuration Example, [Online]. Available: https://networklessons.com/switching/802-1q-tunneling-q-q-configuration-example/ (visited on 02/21/2016).

[39] NOX. (n.d.). The NOX Controller Repository, [Online]. Available: https://github.com/noxrepo/nox (visited on 03/30/2016).

[40] ONF. (Mar. 2015). Openflow switch specification version 1.5.1, [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.1.pdf (visited on 03/21/2016).

[41] OpenDaylight. (Jan. 2015). Odl openflow questions, [Online]. Available: https://lists.opendaylight.org/pipermail/openflowplugin-dev/2015-January/002469.html (visited on 04/04/2016).

[42] OpenVPN. (n.d.). OpenVPN security overview, [Online]. Available: https://openvpn.net/index.php/open-source/documentation/security-overview.html (visited on 03/30/2016).

[43] OVS. (Dec. 2014). Open vSwitch version 2.3.1 release, [Online]. Available: https://github.com/openvswitch/ovs/releases/tag/v2.3.1 (visited on 03/23/2016).

[44] OVX. (n.d.). Openvirtex and openstack, [Online]. Available: http://ovx.onlab.us/openstack/ (visited on 03/22/2016).

[45] ——, (n.d.). Openvirtex architecture, [Online]. Available: http://ovx.onlab.us/documentation/architecture/overview/ (visited on 03/13/2016).

[46] ——, (n.d.). Openvirtex architecture - operation and subsystems, [Online]. Available: http://ovx.onlab.us/documentation/architecture/operation-and-subsystems/ (visited on 03/13/2016).

[47] F. Pakzad, M. Portmann, W. L. Tan, and J. Indulska, "Efficient topology discovery in software defined networks," in *Signal Processing and Communication Systems (ICSPCS), 2014 8th International Conference on*, Dec. 2014, pp. 1–8. DOI: 10.1109/ICSPCS.2014.7021050.

[48] "Fast Reroute Extensions to RSVP-TE for LSP Tunnels," Tech. Rep., May 2005. DOI: 10.17487/rfc4090. [Online]. Available: https://tools.ietf.org/html/rfc4090.

[49] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado, "The Design and Implementation of Open vSwitch," in *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'15, Oakland, CA: USENIX Association, 2015, pp. 117–130. [Online]. Available: http://dl.acm.org/citation.cfm?id=2789770.2789779.

[50] POX. (n.d.). The POX Controller Repository, [Online]. Available: https://github.com/noxrepo/pox (visited on 03/30/2016).

[51] F. S. Rechia. (Mar. 2016). Test packets repository, [Online]. Available: https://bitbucket.org/fsrechia/test_packets/ (visited on 03/14/2016).

[52] ——, (2016). Thesis Repository, [Online]. Available: https://bitbucket.org/fsrechia/thesis/overview (visited on 03/23/2016).

[53] Y. Rekhter, B. Moskowitz, D. Karrenberg, G. J., and E. Lear, "Address Allocation for Private Internets," Tech. Rep., Feb. 1996. DOI: 10.17487/rfc1918. [Online]. Available: http://dx.doi.org/10.17487/RFC1918.

[54] E. Rosen, A. Viswanathan, and R. Callon, "Multiprotocol Label Switching Architecture," Tech. Rep., Jan. 2001. DOI: 10.17487/rfc3031. [Online]. Available: https://tools.ietf.org/html/rfc3031.

[55] Scapy. (Jan. 2016). Scapy GitHub repository, [Online]. Available: https://github.com/secdev/scapy/ (visited on 01/20/2016).

[56] ——, (Jan. 2016). Scapy project, [Online]. Available: http://www.secdev.org/projects/scapy/ (visited on 01/20/2016).

[57] A. Al-Shabibi, M. De Leenheer, M. Gerola, A. Koshibe, G. Parulkar, E. Salvadori, and B. Snow, "Openvirtex: Make your virtual SDNs programmable," in *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '14, Chicago, Illinois, USA: ACM, 2014, pp. 25–30. DOI: 10.1145/2620728.2620741. [Online]. Available: http://doi.acm.org/10.1145/2620728.2620741.

[58] R. Sherwood. (2014). Cbench: A Benchmarking Tool for Controllers, [Online]. Available: https://github.com/andi-bigswitch/oflops/tree/master/cbench (visited on 03/23/2016).

[59] 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, Jan. 2010. DOI: 10.1145/1672308.1672333. [Online]. Available: http://doi.acm.org/10.1145/1672308.1672333.

[60] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, "Flowvisor: a network virtualization layer," Deutsche Telekom Inc. R&D Lab, Stanford, Nicira Networks, Tech. Rep., 2009.

[61] Tcpdump. (n.d.). Tcpdump Command-line Packet Analyzer, [Online]. Available: http://www.tcpdump.org/ (visited on 04/01/2016).

[62] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood, "On Controller Performance in Software-Defined Networks," in *Presented as part of the 2nd USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*, San Jose, CA: USENIX, 2012. [Online]. Available: https://www.usenix.org/conference/hot-ice12-0/controller-performance-software-defined-networks.

[63] URIH. (n.d.). Smart whois lookup, [Online]. Available: http://whois.urih.com/record/1.0.0.1/ (visited on 04/02/2016).

[64] Wireshark. (n.d.). Ethernet vendor codes, and well-known MAC addresses, [Online]. Available: https://code.wireshark.org/review/gitweb?p=wireshark.git;a=blob_plain;f=manuf (visited on 04/02/2016).

[65] ——, (n.d.). Wireshark Network Protocol Analyzer, [Online]. Available: https://www.wireshark.org/ (visited on 04/01/2016).

[66] H. Yamanaka, E. Kawai, S. Ishii, and S. Shimojo, "Autovflow: Autonomous virtualization for wide-area openflow networks," in *Software Defined Networks (EWSDN), 2014 Third European Workshop on*, Sep. 2014, pp. 67–72. DOI: 10.1109/EWSDN.2014.28.

APPENDIX A

EXPERIMENTAL SETUP DETAILS

A.1 Physical Laboratory

A.1.1 Hosts

The lab test hosts where running Linux with the following hardware and operating system specifications:

- General Hardware: Dell OptiPlex 755
- CPU: Intel(R) Core(TM) 2 Duo E8400
- Linux Distro: Fedora release 20
- Linux Kernel: 3.19.5-100.fc20.x86_64
- Network interfaces: 2x Realtek RTL-8100/8101L/8139 PCI Fast Ethernet Adapter

By default, the Linux distro used in the lab has iptables configured with security rules that prevent iperf from running properly. The configuration below was required in all of the hosts to remove firewall rules and allow use of iperf:

```
# iptables -F
```

In order to reduce the variance of ping tests, ARP entries were manually added to the ARP tables of the computers under test:

```
# arp -s <IP address> <MAC address>
```

The iperf and ping versions used were:

```
[root@hostA ~]# iperf -v
iperf version 2.0.5 (08 Jul 2010) pthreads
[root@hostA ~]# ping -V
ping utility, iputils-s20140519
```

The test scripts for flow setup time and throughput tests are available in [52]:

- 04_common_tools/ping_test.sh
- 04_common_tools/iperf_test.sh

A.1.2 OpenFlow Hardware Switch

Tests run in the lab used the switch Hewlett Packard ProCurve as specified below:

- J8692A HP 3500-24G-PoE yl Switch
- Software: K.16.01.0004

Switch configuration can be found in the file 04_common_tools/openflow_config-hp.txt in [52].

A.2 Virtual Laboratory

A.2.1 Host Machine

- CPU: Intel(R) Core(TM) i7-6700HQ CPU @ 2.6GHz, 16GB RAM (4 cores, 8 threads)
- OS: Windows 10
- Network interfaces: Realtek USB GbE Family Controller
- Software: running VirtualBox Version 5.0.16 r105871

A.2.2 Virtual Machine

- Linux distro: Ubuntu 14.04.3 LTS
- Linux kernel: 3.13.0-76-generic #120-Ubuntu SMP Mon Jan 18 15:59:10 UTC 2016 x86_64 x86_64 x86_64 GNU/Linux
- Virtual machine reserved resources:
  - Memory: 4096MB
  - Processors: 4

A.3 Floodlight

Floodlight version commit 63849d2 Jan 11, 2016 is used. It was downloaded from [20] and compiled locally. The following configuration files, which can be found in [52], were used to instantiate Floodlight:

- Network A: 03_OpenVirtex_resources/floodlightdefault.properties10000
- Network B: 03_OpenVirtex_resources/floodlightdefault.properties20000

- Network C: 03_OpenVirtex_resources/floodlightdefault.properties30000

A.4 OVX

OpenVirteX version 0.0-MAINT was used, latest commit 25f38b7 on May 16, 2014, available in their repository at [3]. Several JSON configuration files were used to configure OVX for the experiments conducted in this work (stored at [52]):

- The virtual Mininet laboratory used three configuration files which correspond to figure 14:
  - JSON config file for network A 03_OpenVirtex_resources/test03a.json;
  - JSON config file for network B 03_OpenVirtex_resources/test03b.json;
  - JSON config file for network C 03_OpenVirtex_resources/test03c.json;
- The physical testbed configuration 03_OpenVirtex_resources/test_lab2.json, connecting computers C (physical port 13) and D (physical port 15) to the same virtual network managed by OVX. This configuration file correspond to figure 15;

Furthermore, start and stop scripts were created to initiate OpenVirteX, Floodlight and the Mininet network all at once, to make experiments easier (stored at [52]):

- 03_OpenVirtex_resources/startup_test_scenario.sh starts up everything needed to test OVX with the Mininet testbed;
- 03_OpenVirtex_resources/lab_test.sh starts up everything needed to test OVX with the physical switch in the lab;
- 03_OpenVirtex_resources/stop_test_scenario.sh kills all the processes related to OVX, Floodlight and Mininet to stop experiments.

A.5 FlowVisor

FlowVisor version used for the experiments was flowvisor-1.4.0, latest commit b45b58f on Aug 30, 2013, available at [1]. All of the configuration files and scripts mentioned in this section are stored at [52].

Starting with an empty database, the script 01_flowvisor_resources/configure_slices.sh configures the topology slicing for the three networks in the Mininet scenario shown in figure 14. Then the scripts 01_flowvisor_resources/start_fv_testbed_with_mininet.sh and 01_flowvisor_resources/stop_fv_testbed_with_mininet.sh start and stop, respectively, the Mininet based scenario depicted by figure 14.

The script 01_flowvisor_resources/start_fv_testbed.sh starts with an empty database, configures and starts the FlowVisor and Floodlight controllers to run tests with the hardware switch at the lab, which is depicted by figure 15.

A.6 VeRTIGO

VeRTIGO version vertigo-0.3.8 (based on flowvisor-0.8.1) was used for the experimental attempts conducted in this work and can be found at their repository in [12]. All of the configuration files and scripts mentioned in this section are stored at [52]. VeRTIGO can be started using XML files to configure the slicing of the topology under test. The configuration files for the physical and Mininet testbeds are respectively:

- 02_VeRTIGO_resources/config_phylab.xml;
- 02_VeRTIGO_resources/reduced_config_mininet.xml.

The topology start and stop scripts are:

- 02_VeRTIGO_resources/start_vertigo_testbed_with_mininet.sh starts up everything needed to test VeRTIGO with the Mininet testbed;
- 02_VeRTIGO_resources/start_vertigo_testbed.sh starts up everything needed to test VeRTIGO with the physical switch in the lab;
- 02_VeRTIGO_resources/stop_vertigo_testbed.sh kills all the processes related to VeRTIGO, Floodlight and Mininet to stop experiments.

A.7 Mininet

Mininet version 2.2.1d1 [37] was used with a local patch to make OVSBridge work as a traditional layer 2 bridge. The patch is very simple and just corrects a simple bug that does not exist in the HEAD version of Mininet:

```
diff --git a/mininet/node.py b/mininet/node.py
index 8e91c6e..f48e9c7 100644
--- a/mininet/node.py
+++ b/mininet/node.py
@@ -1183,7 +1183,7 @@ def bridgeOpts( self ):
         if self.protocols and not self.isOldOVS():
             opts += ' protocols=%s' % self.protocols
         if self.stp and self.failMode == 'standalone':
-            opts += ' stp_enable=true' % self
+            opts += ' stp_enable=true' #% self
```

```
        return opts

    def start( self, controllers ):
```

The topology configuration files used for the experiments are in [52]:

- 03_OpenVirtex_resources/virtual_mininet_testbed_simple.py, corresponding to figure 14;
- 01_flowvisor_resources/virtual_mininet_testbed_simple_no_repeated_addresses.py, which just changes network B's range from 10.0.0.0/24 to 11.0.0.0/24.

A.8 Open vSwitches

The virtual switches used for the experiments were Open vSwitches compiled from the source code version 2.3.1, available in [43].

A.9 GENI Testbed

The GENI testbed was reserved using the GENI portal graphical interface and the RSpec 04_common_tools/ig-clemson.rspec.xml stored at [52].

A.10 Test Packets Tool

The test packets tool has a predefined set of packet samples which are used for testing. It sends the following protocols (or packet types) out on a specified network interface card:

- IPv4 unicast:
  - ping: ICMP Echo request over IPv4,
  - udp: UDP over IPv4,
  - tcp: TCP over IPv4,
  - ipfix: Internet Protocol Flow Information Export (IPFIX),
  - ssh: Secure Shell (SSH),
  - telnet: Telnet,
  - http: Hypertext Transfer Protocol (HTTP) ,
  - https: HTTP Secure (HTTPS),
  - snmpv2c: Simple Network Management Protocol (SNMP) version 2c,
  - bgp: Border Gateway Protocol (BGP),

- dns: Domain Name System (DNS),
- mdns: multicast DNS,
- tacplus: Terminal Access Controller Access-Control System Plus (TACACS+) protocol,
- radius: Remote Authentication Dial-In User Service (RADIUS) protocol,
- ntp: Network Time Protocol (NTP).

- IPv6 unicast:
  - ping6: ICMPv6 Echo request,
  - udp6: UDP over IPv6,
  - tcp6: TCP over IPv6.
- IPv4 multicast:
  - igmpv2: Internet Group Management Protocol version 2 (IGMPv2),
  - ldp: Label Distribution Protocol (LDP),
  - ospf: Open Shortest Path First (OSPF) protocol hello message,
  - ripv2: Routing Information Protocol version 2 (RIPv2),
  - pim: Protocol Independent Multicast version 2 (PIMv2),
  - vrrp: Virtual Router Redundancy Protocol (VRRP) ,
  - igmpv3: Internet Group Management Protocol version 3 (IGMP).
- L2 unicast:
  - unicastarp: Address Resolution Protocol (ARP) Reply (unicast),
  - eaps: Ethernet Automatic Protection Switching (EAPS) protocol.
- L2 multicast/broadcast:
  - arp: Address Resolution Protocol (ARP) Request (broadcast),
  - dhcp: Dynamic Host Configuration Protocol (DHCP) (broadcast),
  - glbp: Gateway Load Balancing Protocol (GLBP),
  - stp: Spanning-tree Protocol (STP),
  - lldp: Link Layer Discovery Protocol (LLDP),
  - cfm: Configuration Fault Management (CFM),
  - oam: operations, administration and maintenance (OAM)
  - lacp: Link Aggregation Control Protocol (LACP),
  - cdp: Cisco Discovery Protocol (CDP),
  - pagp: Port Aggregation Protocol (PAGP) ,
  - udld: Unidirectional Link Detection (UDLD),
  - vtp: VLAN Trunking Protocol (VTP),
  - pvst: per-VLAN Spanning-tree (PVST) protocol,
  - marker: Marker Protocol,
  - gvrp: GARP VLAN Registration Protocol (GVRP),
  - dot1x: Extensible Authentication Protocol (EAP),
  - loopback: Loopback-detection protocol (Ethertype 0x8809).