SD-MCAN: A SOFTWARE-DEFINED SOLUTION FOR IP MOBILITY IN CAMPUS AREA NETWORKS

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Adam Calabrigo

December 2017

COMMITTEE MEMBERSHIP

TITLE:	SD-MCAN: A Software-Defined Solution for IP Mobility in Campus Area Networks
AUTHOR:	Adam Calabrigo
DATE SUBMITTED:	December 2017
COMMITTEE CHAIR:	John Bellardo, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	Hugh Smith, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	Phillip Nico, Ph.D. Professor of Computer Science

ABSTRACT

SD-MCAN: A Software-Defined Solution for IP Mobility in Campus Area Networks

Adam Calabrigo

Campus Area Networks (CANs) are a subset of enterprise networks, comprised of a network core connecting multiple Local Area Networks (LANs) across a college campus. Traditionally, hosts connect to the CAN via a single point of attachment; however, the past decade has seen the employment of mobile computing rise dramatically. Mobile devices must obtain new Internet Protocol (IP) addresses at each LAN as they migrate, wasting address space and disrupting host services. To prevent these issues, modern CANs should support IP mobility: allowing devices to keep a single IP address as they migrate between LANs with low-latency handoffs. Traditional approaches to mobility may be difficult to deploy and often lead to inefficient routing, but Software-Defined Networking (SDN) provides an intriguing alternative. This thesis identifies necessary requirements for a software-defined IP mobility system and then proposes one such system, the Software-Defined Mobile Campus Area Network (SD-MCAN) architecture. SD-MCAN employs an OpenFlow-based hybrid, label-switched routing scheme to efficiently route traffic flows between mobile hosts on the CAN. The proposed architecture is then implemented as an application on the existing POX controller and evaluated on virtual and hardware testbeds. Experimental results show that SD-MCAN can process handoffs with less than 90 ms latency, suggesting that the system can support data-intensive services on mobile host devices. Finally, the POX prototype is open-sourced to aid in future research.

ACKNOWLEDGMENTS

I am incredibly indebted to many people for their help in this undertaking. I will now try to thank as many as I can.

This thesis would not be possible without the guidance of my advisor Dr. Bellardo. I am grateful for all of his guidance in the development of this work; his immense knowledge of networking was an invaluable resource.

I would like to thank Keith Glover for giving me my first professional experience in network programming. Of course, I must also thank Raj Srinivasan and all of the engineers at Bivio Networks for their guidance and mentor-ship during my time there: it was there that my interest in networks was born.

I must also thank Dr. Smith, whose introductory networks class I took, then TAed for five quarters, during my time at Cal Poly. His teaching formed the building blocks of my networking knowledge, and the TAing helped me develop my knowledge in a practical setting. It was also just a lot of fun.

Thank you to the Cisco engineers who put up with all of my questions and provided valuable answers.

Five years of engineering really takes its toll on the psyche, so I have to acknowledge my 411 Henderson roommates for keeping me sane throughout this process and making my last three years at Cal Poly delightful.

Finally, I would not have made it this far without the support of my family; thank you for always encouraging me to pursue my passions. And to my girlfriend, Katie, thank you for listening to my complaints for an entire year. I love you all.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

Enterprise networks must support Internet traffic from thousands of hosts. Traditionally, a host remained stationary, connected to the network at a single point of attachment; however, the past decade has seen the employment of mobile computing rise dramatically. As such, modern networks must support mobile devices: devices whose points of attachment change as users physically move around the network.

1.1 Campus Area Networks and Mobility

Campus Area Networks (CANs) are a specific subset of enterprise networks, typically comprised of a network core connecting multiple Local Area Networks (LANs) across the physical area of a college campus. CANs provide connectivity to thousands of students and faculty simultaneously. To this end, CANs are designed with performance and reliability in mind.

However, CANs demonstrate several noticeable shortcomings. Often built from a wide variety of hardware (switches, routers, middleboxes, etc.) from different vendors, CANs can be difficult for network operators to manage. With devices running different proprietary OSes and protocols, network updates must be tailored to each unique device, making network policy changes time-consuming to deploy. Additionally, the behavior of hosts on CANs indicates a clear need for Internet Protocol (IP) mobility which often goes unsatisfied.

CANs exemplify the need for mobility in enterprise networks, as the majority of students and faculty carry one to many mobile devices on their person as they navigate the campus. Whether mobile phones, laptop computers, etc. these devices connect

to many unique access points across multiple LANs as a student walks from one end of campus to another. In their 2005 study of user traffic patterns on the Dartmouth campus, Kotz and Essien found that 40% of mobile sessions involved roaming to different subnets [30]. Upon each connection, a host obtains a local IP address using Dynamic Host Configuration Protocol (DHCP). The DHCP server contains a pool of valid IP addresses and leases these addresses to hosts for a given duration. Before the lease expires, it must be renewed; however, when the device leaves the network, the host's assigned address remains unavailable for the duration of the lease regardless of whether or not the address remains in use. Thus, as students move around campus, addresses are leased for longer than they are used, resulting in wasted address space.

As hosts have become increasingly mobile on CANs in the last decade, their data needs have also evolved. The modern campus-goer streams music and video while navigating campus, in addition to making phone and video calls over IP. Such data-intensive services require a reliable connection to the network. When a person moves to a new LAN and their device leases a new IP address, these services are disrupted. In addition to wasted address space, the disruption of host services reveals a need for mobility on CANs.

The demand for mobility in CANs creates several new challenges in regards to routing and connectivity in the network. Many approaches to mobility on campus networks have emerged. Some approaches, based on the original Internet Engineering Task Force (IETF) standard Mobile IP (MIP), use additional IP addresses to allow mobile devices to remain identifiable by their original IP addresses as they move across LANs. These solutions may be difficult to deploy and often lead to inefficient routing of packets throughout the CAN. Mobility on the CAN has also been achieved using Virtual LANs (VLANs) without the need for additional addresses [57]. Commercial solutions to mobility exist, including offerings from Cisco Systems and Aruba Networks. Cisco provides mobile IP configuration via the iOS on its routers [1], while

Aruba offers a line of mobility controllers (hardware boxes) to provide mobility to enterprise networks [2]. While these solutions function well, this thesis focuses on applying a new technology, software-defined networking (SDN), to the problem of mobility on campus area networks.

1.2 Software-Defined Networking as a Solution

Software-Defined Networking (SDN) provides an intriguing alternative to traditional mobility approaches. A new networking paradigm, SDN decouples the control and data planes of networking devices. Traditionally, the control plane of a switch or router makes forwarding decisions while the data plane forwards the packets. SDN removes the control plane from network devices, choosing instead to centralize control of the network to a separate server, called a *controller*. Centralizing control provides a complete view of the network which the controller can leverage to make routing decisions and forward those decisions to the network devices, which now act as dumb devices. SDN's centralized network view makes it an ideal tool for mobility solutions.

This thesis identifies four design criteria for any successful CAN mobility solution: compatibility with the existing network, routing efficiency, handoff latency, and scalability. It then proposes a mobility solution, the Software-Defined Mobile Campus Area Network (SD-MCAN) architecture, designed to satisfy these criteria. SD-MCAN is built on OpenFlow, a popular and widely supported implementation of the SDN paradigm. In OpenFlow, network devices route packet flows based on flow tables, and a centralized controller manages all traffic flows on the network through the installation of flow table entries. Using OpenFlow, SD-MCAN's operation is three-fold. First, it acts as a mobility-enabled DHCP server to all LANs on the CAN. This allows SD-MCAN to lease a single IP address to all hosts on the network, even mobile hosts. Additionally, SD-MCAN maintains a complete view of the core network

topology and tracks the location of all hosts on the network. Finally, SD-MCAN utilizes this complete network view to efficiently route traffic flows using a hybrid, label-switched routing scheme. The scheme is hybrid as SD-MCAN installs flow table entries proactively in the network core and reactively on the network edges.

To evaluate the proposed system, a prototype is built as an application on top of POX, a modular, Python-based SDN controller. The prototype is tested on virtual and hardware networks to analyze handoff performance and scalability. Experimental results show that the SD-MCAN prototype suffers only transient performance degradation (<400 ms performance degradation) during host handoffs. The prototype handles host migration with minimal performance impact, packet loss <2% and throughput degradation <15%, when hosts move constantly at intervals greater than two seconds. The prototype also handles constant movement at shorter intervals, albeit with more significant performance impact (3.5% packet loss and 21% throughput degradation). While the SD-MCAN prototype is limited due to its Python implementation, analysis shows that a production quality SD-MCAN deployment is capable of handling host handoffs with <90 ms of negative performance impact. This suggests that SD-MCAN could support data-intensive services on mobile host devices.

Large scale experimentation shows that SD-MCAN's label-switched routing scheme keeps core flow tables small (average 11 entries for 1024 mobile hosts); however, edge flow tables do not scale as well (average 362 entries for 1024 mobile hosts), limiting SD-MCAN's deployability on network devices with restricted flow table sizes. While edge flow tables can grow large, experimental results show that the load on the controller remains reasonable (<21 packets per second with a host moving every 10 ms); thus, the presented SD-MCAN design is feasible on networks with devices containing adequately sized flow tables given the number of hosts on the network.

1.3 Contributions

This thesis focuses on applying modern SDN technologies to current problems in CANs, namely the need for scalable mobility with fast handoffs between LANs. The contribution of this thesis is as follows.

1. Identifies urgent design considerations for applying SDN to mobility problems on CANs,
2. Presents and specifies a new OpenFlow-based CAN architecture called SD-MCAN,
3. Introduces a novel hybrid, label-switched mobile routing scheme for use in SD-MCAN,
4. Details three new Python-based applications for extending the POX controller to implement an SD-MCAN prototype, and
5. Provides a comprehensive evaluation of the system's performance.

The remainder of this thesis documents the research and design decisions behind SD-MCAN. Chapters 2 and 3 discusses background research and related work, respectively. Chapter 4 presents design considerations and the SD-MCAN architecture, and Chapter 5 details the implementation of the SD-MCAN prototype on the POX controller. Chapter 6 features the experimentation and validation of the prototype on a virtual test bed of Open vSwitches using the Mininet tool. Finally, Chapter 6 suggests areas of future work, and Chapter 6 draws conclusions from this work.

Chapter 2

BACKGROUND

Since the introduction of Mobile IP (MIP) by the IETF in 1996, understanding of and approaches to mobility in IP networks have significantly evolved, as have networks themselves. This chapter begins with a short introduction to two networking concepts which inform SD-MCAN, the Dynamic Host Configuration Protocol (DHCP) and the Multi-Protocol Label Switching (MPLS) system, followed by an examination of mobility through MIP and the improvements that have been made since its inception. The chapter then surveys the evolution of alternative networking paradigms, leading to the emergence of SDN. Finally, this chapter introduces OpenFlow, the SDN implementation upon which this work builds.

2.1 Dynamic Host Configuration Protocol

To begin this section, it must be noted that this section presupposes a minimal knowledge of TCP/IP in the reader. A more in-depth discussion of these concepts is outside the scope of this work, but can be found in [51]. Instead, this section provides a brief introduction to the history and the operation of the Dynamic Host Configuration Protocol (DHCP).

2.1.1 History

In the early days of the Internet, hosts connected to the Internet needed to know and statically configure their IP addresses in order to use the Internet Protocol. This demands some knowledge of the network configuration; therefore, a host without this information would not be able to send datagrams over a network using IP. Emerging

in 1984, the now-obsolete Reverse Address Resolution Protocol (RARP) offered a solution to this problem by allowing a host to request its IPv4 address from a server [21].

But RARP fell short in its reliance on the data link layer and its demand for many servers across the network. The following year, RFC 951 introduced the Bootstrap Protocol (BOOTP). Unlike RARP, BOOTP operates on the network layer and utilizes relay agents, removing RARP's need for a server on every IP subnet. BOOTP allocates IP addresses out of an address pool [16].

First introduced in [19], DHCP superceded BOOTP by introducing the concept of address leases. Based on BOOTP, DHCP dynamically allocates IP addresses from an address pool. The key innovation of DHCP is its ability to reclaim allocated addresses back into its address pool after expiration. DHCP has been updated numerous times since.

2.1.2 Operation

Like BOOTP, DHCP utilizes the User Datagram Protocol (UDP) for connectionless communication with hosts on a network. DHCP's operation has four phases when initiating a lease. To start, the client broadcasts a DHCP *Discovery* on its network. The DHCP server receives this broadcast, reserves an IP address from its address pool, and unicasts a DHCP *Offer* message to the client. Upon receiving the offer, the client sends a DHCP Request packet, verifying that it wants the offered IP address. Finally, the server sends a DHCP *Acknowledge* packet back to the client, acknowledging that the IP address has been leased. This packet contains information about the duration of the lease. Figure 2.1 below shows this process.

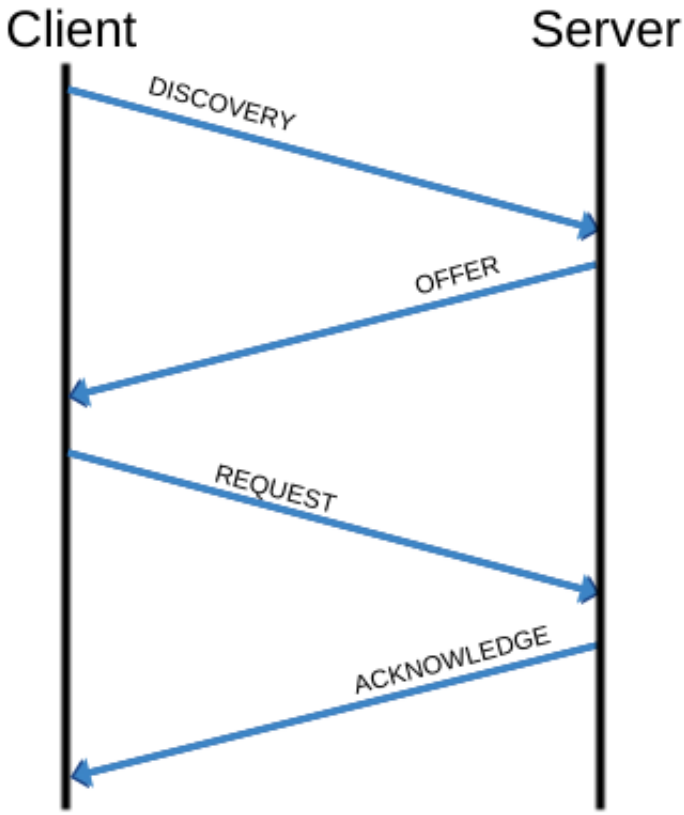

Figure 2.1: DHCP Session

The client must periodically renew its lease with the server to keep its lease. If the client knows that it is leaving the network, it may send a DHCP *Release* packet to the server, indicating that it is no longer using the lease. In response, the server will add the leased IP address back to its address pool. However, clients often do not know when they will be disconnected from the network (*i.e.* when a computer is manually unplugged from the network). In this case, the leased IP address remains unused for the remaining duration of its lease before being added back into the address pool.

2.2 Multi-Protocol Label Switching

Traditional IP routers route datagrams via a series of lookups in a routing table. The router receives a datagram, determines the datagram's next-hop, and forwards the datagram out of the interface connected to that next hop. This process occurs at each router in the datagram's path across the network. In large networks, routing tables grow large and route look-ups become costly.

This motivated the creation of Multi-Protocol Label Switching (MPLS), a method of protocol-independent data transfer using *label switching* to reduce the number of look-ups in the datagram's path [47]. MPLS is often considered a "layer 2.5" protocol as it sits between the link layer (layer 2) and the network layer (layer 3) in the traditional OSI model. It combines the functionality of layer 3 routing with the simplicity of layer 2 switching.

Rather than route like traditional routers, MPLS utilizes label switching. When a packet enters the network, the first router, called a *Label Edge Router* (LER), performs a traditional routing look-up. However, instead of finding the next hop, the LER locates the packet's destination router on another edge of the network and finds a precalculated path between the routers, called a *Label Switched Path* (LSP). The LER pushes a label onto the ingress packet, and routers along the LSP, called *Label Switching Routers* (LSRs), route the packet using label switching only. A final LER pops the label off before the packet leaves the network. Figure 2.2 below shows an example MPLS network.

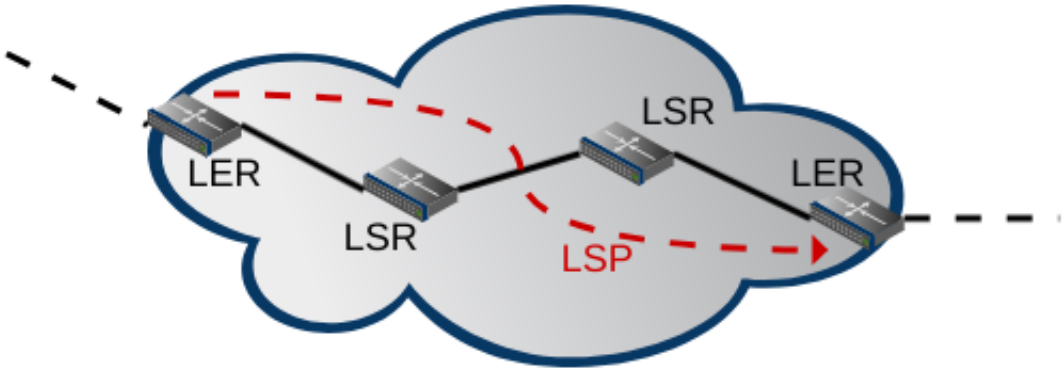

Figure 2.2: MPLS Network

In order for an LSP to be used, it must be shared across the routers; this is because, although the LSP is network-wide, the labels are only link-local. Two MPLS routing protocols are commonly employed to share labels: *Label Distribution Protocol* (LDP) and *Resource Reservation Protocol with Traffic Engineering* (RSVP-TE). RSVP-TE is the more complex protocol, but some complex MPLS networks use both protocols.

2.3 Mobility

Campus Area Networks (CANs) consist of multiple Local Area Networks (LANs) interconnected via layer 3 routing devices. Classically, college campuses housed computers in fixed locations, *i.e.* a node subscribed to a single LAN via a fixed point of attachment. However, the proliferation of mobile devices (laptops, cellphones) on campuses and the change in student needs have created new demands.

In modern CANs, devices move around campus, disconnecting from one LAN and connecting to another as students and faculty navigate the physical area. As they move, students stream music and video, as well as making calls over IP. Typically, CANs employ DHCP to allocate IP addresses to nodes; thus, a node occupies multiple

addresses as it moves between LANs. These inter-LAN handoffs disrupt services like video streaming and voice-over-IP (VoIP). Given the needs of students, it becomes evident that modern CANs must support lossless handoffs between LANs over IP.

The original design of the Internet did not account for this need, which becomes evident when looking at the role of IP addresses. At a high level, IP addresses serve two main functions in computer networks: 1) they allow end nodes, which this work will also refer to as hosts, to be uniquely identified by other hosts, and 2) they provide layer 3 devices a means of routing Internet datagrams between hosts [51]. Looking at these functions in terms of mobility, the physical movement of a mobile node between points of attachment in a network without connectivity loss, reveals an inherent conflict.

A host needs a stable IP address in order to be identified by other hosts, yet datagrams will always be routed to a stable IP address along the same path (disregarding network congestion impacts on routing). This inhibits mobility, as datagrams bound for a mobile host will always be routed to the same location regardless of the host's physical location.

2.3.1 Mobile IP

Mobile IP, introduced in RFC 2002, addresses this issue by distributing the functionality of a host's IP address across two addresses, a *home address* for identification and a *care-of address* (CoA) for routing [40]. Mobile IP has since become known as *Mobile IPv4* (MIPv4), and its specification has been updated in RFC 5944 [41]. MIPv4 labels hosts as *Mobile Nodes* (MNs) and extends IP with three main functions to provide mobility to MNs: 1) agent discovery, 2) node registration, and 3) tunneling [43].

MIPv4 achieves mobility through the use of *Mobility Agents* (MAs), which take

the form of *Home Agents* (HAs) and *Foreign Agents* (FAs). An HA is a router on the MN's home network, the network having a network prefix matching that of the MN's home address, while an FA is a router on any other network. In order for these MAs to provide utility, MNs must be made aware of their presence.

The *agent discovery* mechanism of MIPv4 allows MAs to make themselves known to MNs through advertisements. MAs periodically broadcast advertisements to attached subnets. Upon receiving an advertisement, a MN checks whether the advertising agent is on its home network. If the MA is on the MN's home network, then the MN operates as it would with unextended IP; however, if the MA is on a foreign network, then the MN must obtain a CoA from the FA and register its CoA with its HA. Alternately, an impatient MN may send its own advertisement to elicit a MA advertisement.

After obtaining a CoA from an FA, the MN registers its CoA with its HA through *registration*. The HA creates a *binding*, linking the CoA and the home address. This process is important, as the HA must know the location of the MN in order to forward traffic to the MN. As long as the MN stays on a foreign network, it will continually register new CoAs with its HA.

Other nodes still identify the MN by its home address, thus all traffic bound for the MN will be addressed to its home address. Fortunately, the HA knows the MNs current location after successful registration and can forward MN-bound traffic accordingly. This process is known as *tunneling*.

In tunneling, the home agent intercepts datagrams headed for the MN's home address and reroutes them to the MN's CoA. MIPv4 accomplishes this by encapsulating the IP datagram in a new IP header (a tunnel header) containing the MN's CoA as destination IP address. The tunnel header uses a special value in the IP protocol field to divulge the presence of the encapsulated datagram inside.

When another node, a *Correspondent Node* (CN), sends a datagram to the MN, the HA will tunnel the datagram to the MN via the FA. Going the other direction, the MN's response does not have to be tunneled and can be handled using normal IP routing (assuming the CN is not itself an MN). This creates a situation known as *triangle routing*, shown in Figure 2.3.

Triangle routing is often inefficient [15]. Consider, for example, a case where the MN's current foreign network is the CN's network and the MN's home network is across the campus in another building. If the CN sends datagrams to the MN, that traffic will be routed across the campus, only to be tunneled back to the subnet from whence it originated. This problem is often the focus of routing optimization in newer mobility protocols, seen in later sections.

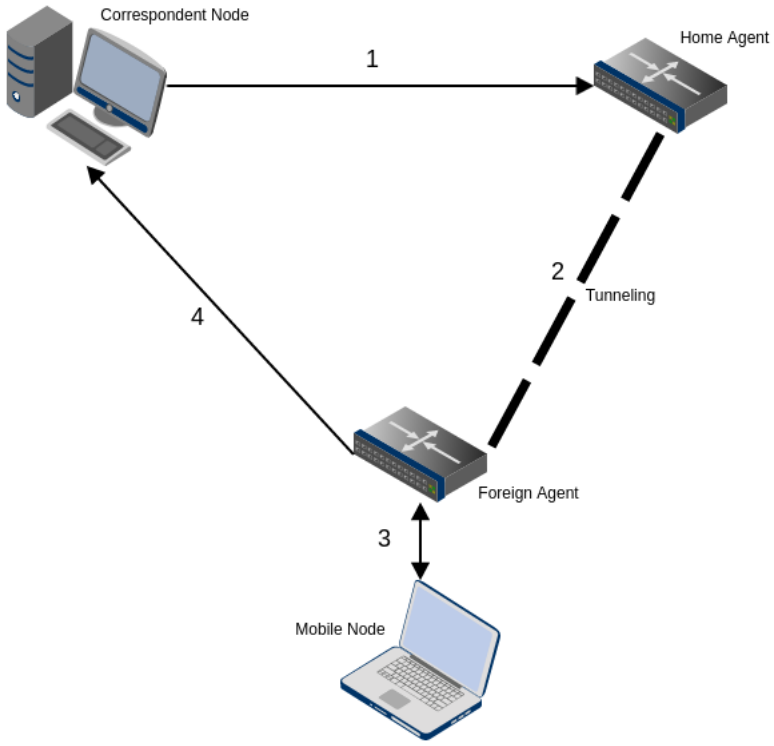

Figure 2.3: Triangle Routing

2.3.2 Mobile IPv6

Due to the exhaustion of IPv4 address space, the IETF introduced IPv6 in RFC 2460 [18]. IPv6 features 128-bit addresses versus IPv4's 32-bit addresses, allowing for practically limitless address space. This protocol necessitated its own mobility solution, known as Mobile IPv6 (MIPv6) [42]. MIPv6 operates similarly to MIPv4, so this section will focus on the differences between the protocols and the various extensions which researchers have developed to improve MIPv6.

The idea behind MIPv6 remains the same as that of MIPv4: HAs forward datagrams to a MN's CoA to enable mobility. However, MIPv6 makes use of IPv6's address configuration protocols, *Neighbor Discovery* and *Address Auto-configuration*, to remove the need for FAs. Rather, MIPv6 uses *Access Routers* (ARs) to track MNs. Additionally, MIPv6 features built-in route optimization. MIPv6 also has its own drawbacks. MNs must send *Binding Updates* (BUs) every time they move between ARs, which leads to a significant amount of signaling between devices and limits the scalability of the protocol.

2.3.2.1 Extensions

Many extensions have been introduced to further improve MIPv6. Hierarchical MIPv6 (HMIPv6) aims to reduce the amount of signaling between MNs, CNs, and HAs [50]. It accomplishes this through the use of *Mobility Anchor Points* (MAPs). A MAP acts as a local HA, located using a *Regional CoA* (RCoA). The MAP then maps the MN's home address to an *On-Link CoA* (LCoA), hidden from the rest of the network. Thus, the MN's HA is only aware of the MAP, which may have multiple ARs connected to it. As the MN moves between ARs in a single MAP, its HA does not need to be notified. This way, HMIPv6 localizes signaling from the rest of the network. Proxy MIPv6 (PMIPv6) offers a similar solution but utilizes *Mobile Access*

Gateways (MAGs) for mobility management rather than MAPs [8].

2.4 Software-Defined Networking

Computer networking presents many challenges on both a local and global scale. Within enterprises, network administrators must translate high-level policies into low-level commands, a significant task given that the typical network is comprised of a vast array of devices, each utilizing many different protocols. Globally, *internet ossification*, the idea that Internet protocols are so widely used that they can no longer be realistically changed, presents a monumental challenge to network evolution. This ossification can be seen in the extremely slow adoption of IPv6. As of 2014, only 12% of allocated prefixes are IPv6 [17]. Software-Defined Networking (SDN) arose in response to this ossification.

2.4.1 Overview

Traditional packet-switched networks rely on distributed protocols to forward packets; that is, each switch has a control plane for determining the forwarding of traffic and a data plane for forwarding traffic out of interfaces. The control plane features data tables for storing and looking up path information, and the data plane consists of the underlying hardware interfaces on the switch [20]. Figure 2.4 depicts a traditional packet-switching network.

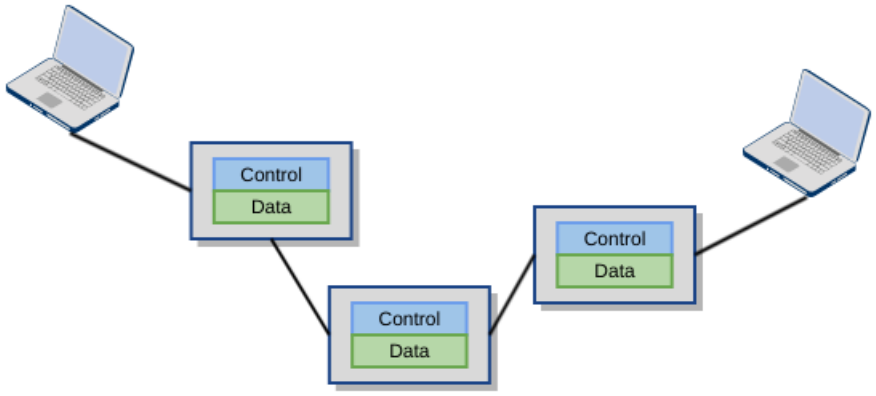

Figure 2.4: Packet-Switching Network

SDN simplifies network management by decoupling forwarding hardware from control decisions [39]. A new paradigm for programmable networks, it centralizes control decisions into software-based controllers, leaving network devices to function as basic packet forwarding hardware. Figure 2.5 shows the same packet-switching network, now deployed using the SDN architecture.

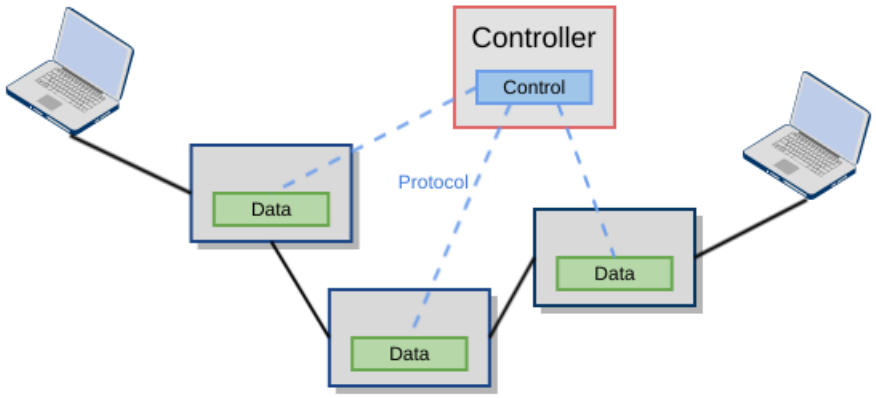

Figure 2.5: Packet-Switching Network with SDN

This decoupling means that the distribution model of the control plane no longer needs to match the distribution model of the data plane. This results in greater control over packet switching throughout the network. By centralizing the control plane to a

remote controller, SDNs provide the network operator flexibility to experiment with new protocols and alter network policy. See the Related Work chapter for modern examples of SDN deployments.

2.4.2 History

SDN has emerged as a popular buzzword for programmable networks in recent years. However, when considering the history of programmable networks, it becomes clear that SDN is the result of the natural evolution of programmable networks over the last twenty years. This section aims to provide background on the development of programmable networks leading to the rise of SDN. In doing so, it details the ideas at the core of the SDN paradigm and the design decisions that molded its evolution.

Before the advent of programmable networks, networks were proprietary. A network was composed of switches, routers, and other middleboxes (such as load balancers and firewalls), each making control decisions with proprietary software and forwarding packets out of proprietary interfaces. In such a closed network, control is distributed across devices. These devices operate on the protocol level and need to be configured via configuration interfaces by a network administrator. Thus, these networks were complex and resistant to innovation [52].

2.4.2.1 Clean Slate Approach to Networking

The explosion of the Internet in the 1990s led to the rise of more complicated network applications and stoked researchers' interest in network development. However, innovation at the protocol level involved the slow process of testing and standardization through the Internet Engineering Task Force (IETF). Frustration with this process, combined with reducing costs of computing, gave rise to research into *clean slate* approaches to networking, where the focus was rethinking network architecture rather

than improving the existing Internet [52].

One of these approaches was *active networking* (AN), which proposed viewing network nodes much the same way as programmable computers, where an *application programming interface* (API) provides access to underlying resources like CPUs and queues. Thus executable programs could be deployed dynamically into network nodes. In the mid-1990s, funding from the U.S. Defense Advanced Research Projects Agency (DARPA) created the Architectural Framework for Active Networks, which defined a unified vocabulary and framework for active networking research collaboration [13].

This minimal framework was intentionally general in order to allow researchers to work independently and experiment with new paradigms. The framework accommodated two predominant active network programming models at the time: the *capsule* model and the *programmable router* model. In the capsule model, programs to be executed by network nodes are carried in-band inside data packets. Alternately, in the programmable router model, programs are defined by an external mechanism and carried out-of-band [22].

Researchers realized several potential benefits to these approaches. With programmable nodes, service providers could potentially alter network behavior and deploy new services much faster than in traditional networks. Similarly, researchers could experiment with new features without disrupting normal network service. AN also had the potential to allow network administrators to create and deploy services in reaction to current network conditions [9].

While AN offered a new paradigm to accelerate network innovation, AN solutions never saw large-scale deployment due to their inability to provide exciting solutions to urgent problems [12]. However, AN research did generate several enduring ideas which informed future work, including SDN. AN advocated the idea of programmable networks as a solution to stagnate network innovation. While AN focused mostly on

data-plane programmability, SDN focuses on control-plane programmability. Regardless, AN first recognized the need for separating normal and experimental traffic on production networks, a principal which pervades modern SDN literature.

2.4.2.2 ForCES and RCP

As the Internet developed in the 1990s, link speeds increased to accommodate larger volumes of traffic, particularly in network cores. To meet this need for speed, vendors began implementing the data plane of network devices entirely in hardware while the control plane remained software. As this traffic increase continued into the 2000s, demand for network performance, reliability, and traffic engineering (TE) grew as well. The difficulty in ensuring these qualities using conventional routing protocols led to innovation.

One of these innovations came in the form of *Forwarding and Control Element Separation* (ForCES). Introduced in RFC 3746, ForCES defines a framework and protocol for standardizing information exchange between the control and data planes of networking devices [56]. Standardizing this communication allows the control and data planes to be physically separate in a network. ForCES intended to allow vendors to develop data and control plane devices independently, speeding up innovation [55].

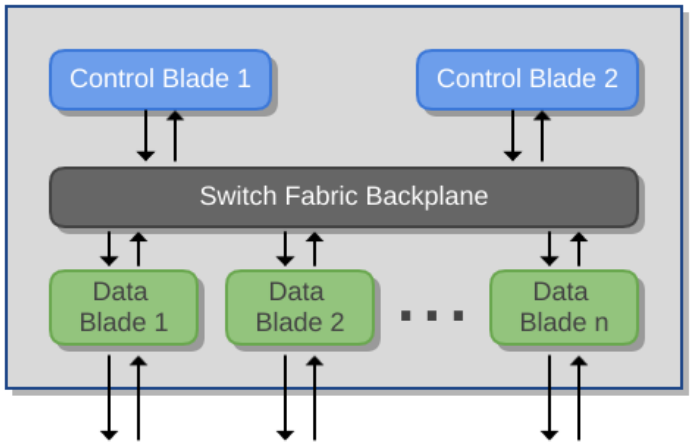

Figure 2.6: ForCES Router with Separate Blades

To this end, ForCES supports two forms of separation: *blade level* and *box level*. At blade level separation, ForCES replaces communication between proprietary interfaces of the control and data planes in a single box. Figure 2.6 above illustrates one such configuration in a router with two control blades and multiple forwarding planes. At box level, control and data planes can exist in physically separate boxes but form a single networking element. Figure 2.7 shows a single ForCES router composed of four separate boxes.

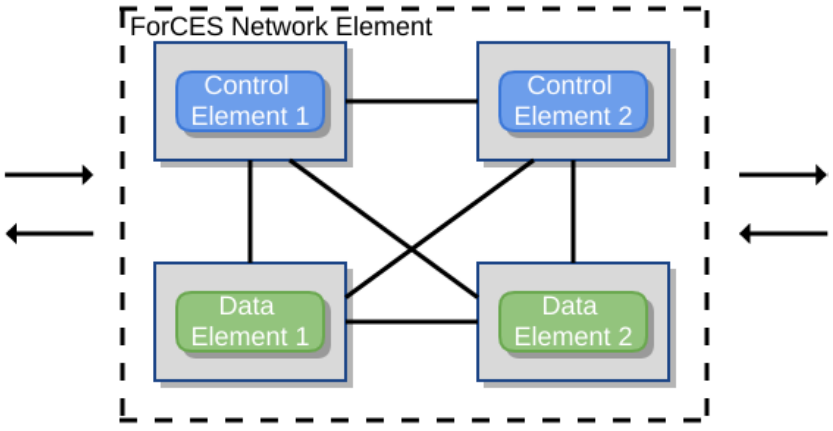

Figure 2.7: ForCES Router with Separate Boxes

Another innovation came via the *Routing Control Platform* (RCP) [11], which collects information about external network destinations and internal network topology and chooses Border Gateway Protocol (BGP) routes for the routers in an Autonomous System (AS). The significant innovation here is the introduction of a logically-centered control plane. This control plane utilizes a complete view of the AS topology to install BGP routes [20]. Figure 2.8 below shows the simple setup of an RCP AS.

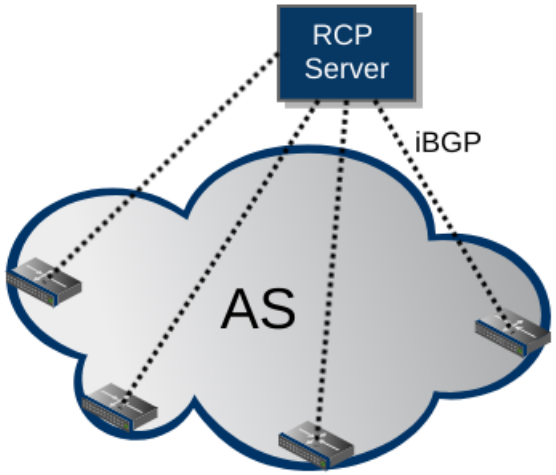

Figure 2.8: AS using RCP

2.4.2.3 Ethane

In the late 2000s, the Ethane project at Stanford University coupled centralized control with simple, flow-based Ethernet switches, laying the groundwork for modern SDN [14]. Ethane presents a management architecture for enterprise networks, featuring two main components: the *controller* and the *switch*. The controller contains the policy for routing of all packets throughout the network; Ethane allows no communication between hosts without explicit permission. The controller knows the network topology and allows packets to navigate the network by installing flows in the switches.

The Ethane switch is a dumb device, featuring a flow table and a secure channel to the controller. When a packet enters the switch, one of two things occurs. If the packet is in the flow table, the switch forwards the packet according to the controller's instruction. If the packet is not in the flow table, then the switch forwards the packet to the controller along with the port on which the packet arrived. The Ethane project is the base on which OpenFlow, the SDN API used in this thesis, is built.

2.5 OpenFlow

The term SDN emerged in reference to Stanford's OpenFlow project and is often used synonymously with OpenFlow to this day; however, the two are not the same. SDN is a paradigm, and OpenFlow is a widely-used API implementation of that paradigm. Other implementations exist, notably FlowVisor, POF, and Cisco OpFlex. Researchers developed OpenFlow as a way to implement experimental protocols on in-use networks [38].

Since the introduction of OpenFlow 1.0 in 2009, the protocol has seen numerous updates. While the current version of OpenFlow is 1.5, this work focuses on version 1.3, as newer versions do not yet have widespread support by hardware manufacturers. This section aims to introduce the overall architecture of OpenFlow and provide and overview to its operation as it pertains to this work.

2.5.1 OpenFlow Architecture

OpenFlow follows the SDN paradigm: network devices become OpenFlow Switches (OFSs), controlled by a separate SDN controller. In this way, OpenFlow allows the controller(s) to dictate the paths of packets across the network. Because all control plane decisions are made in the controller, OpenFlow provides far more flexibility with regard to traffic engineering and quality-of-service than routing protocols or access

control lists [10]. Additionally, OpenFlow frees the network operator from having to manage proprietary software on vendor switches differently: the controller can uniformly manage switches from different vendors, assuming they support OpenFlow.

Each OFS contains one or more *flow tables*, which associate an action with a flow entry. In this context, a *flow* is a stream of traffic consisting of many packets. Flow tables allow the controller to control all flows in the network. Each OFS also has a secure *OpenFlow channel*, connecting it to a remote SDN controller. Figure 2.9 above shows the basic OFS architecture.

The secure channel between the controller and an OFS takes the form of a network connection between an interface on the controller and an interface exported from the OFS [45]. The channel is typically encrypted through TLS, although it may not be. The controller utilizes the *OpenFlow Protocol* (OFP) to communicate with switches in the network. An OFS acts as a dumb device, simply forwarding flows according to its flow tables.

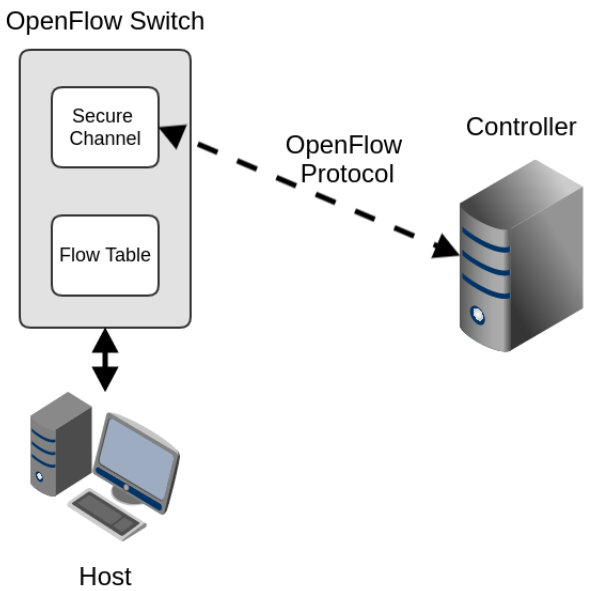

Figure 2.9: OpenFlow Architecture

2.5.2 OpenFlow Channel

The OpenFlow channel is the interface through which the controller connects to the switch [45]. In most cases, the controller maintains multiple OpenFlow channels, one for each switch with which the controller is connected. The channel may be in-band or out-of-band.

The channel connection setup resembles that of a standard TLS or TCP connection. By default, the OpenFlow controller is located on TCP port 6633 or 6653, depending on the OpenFlow version. If the devices use TLS, then the connection will be encrypted. Both devices authenticate by exchanging certificates signed by a site-specific private key. To enable this, the switch must be configured with certificates to authenticate traffic to and from the controller.

Figure 2.10 below shows the initial connection handshake between the switch and the controller. The switch knows the IP address of the controller and initiates the connection. Upon connecting, both the controller and the switch send an OFPT_HELLO message. The message contains a header with a version field; this field is set to the highest OFP versions supported by both devices. The devices negotiate what version of OFP to use. If this version is supported by both parties, then the connection goes ahead; otherwise, the connection is terminated.

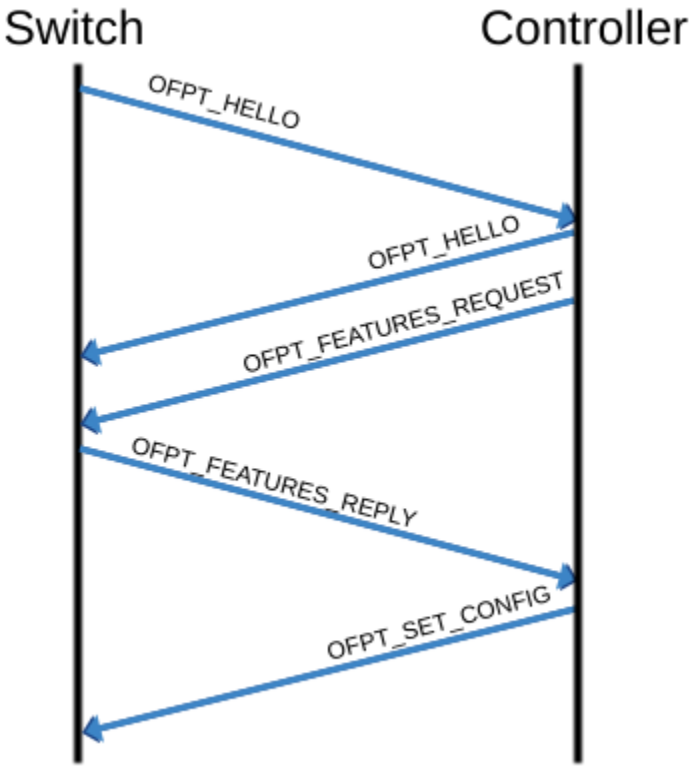

Figure 2.10: OpenFlow Connection Setup

After establishing the connection, the controller sends an OFPT_FEATURES_REQUEST to the switch, containing only a header. The controller collects the switch's information through via the switch's OFPT_FEATURES_REPLY response. This packet tells the controller the switch's datapath ID, the number of packets that the switch can buffer, the number of flow tables supported by the switch, and the capabilities of the switch.

If at any point the OpenFlow channel connection is lost, then the switch fails into one of two modes: *fail secure mode* or *fail standalone mode*. In fail secure mode, the switch resumes its operation except that it no longer forwards packets and messages to the controller. Rather, these packets are dropped instead. In fail standalone mode, the switch reverts to behaving like a legacy switch and stops using OpenFlow. The

switch's configuration determines which fail mode it enters.

2.5.3 OpenFlow Protocol

All switch management occurs through this channel via one of three message types: *controller-to-switch*, *asynchronous*, and *symmetric*. This section provides an incomplete survey of the distinctions and uses of these three message types and the scenarios in which they are employed. For a complete treatment, see the official OpenFlow specification [44].

2.5.3.1 Controller-to-Switch Messages

Controller-to-switch messages provide the controller a means of configuring and updating switches, and they do not always require a response. First, the controller utilizes these messages to learn about each switch during the initial handshake, seen in the previous section. After setup, the controller can access and modify the switch configuration at any time. Notably, this configuration includes how many bytes of each packet are sent to the controller from the switch as well as how many flow tables are in the switch.

The controller also uses controller-to-switch messages to modify the state of the switch. The OFP_FLOW_MOD message allows the controller to add, remove, and modify flow entries from a switch's flow tables. See the next section for more information on the capabilities of these messages.

The final controller-to-switch message type that this section will detail is the *packet-out* message. The controller uses packet-out messages for two purposes: 1) to create and send new packets out of a given switch port and 2) to forward packet-in packets out of a specified switch port. The packet-out must contain an action set telling the switch where to send the packet.

2.5.3.2 Asynchronous Messages

Unlike controller-to-switch messages, asynchronous message are sent exclusively from the switch to the controller without any solicitation from the controller. Switches use these messages to alert the controller of some action, whether that be incoming packets, switch changes, or errors. Asynchronous messages come in four main varieties: *packet-in*, *flow-removed*, *port-status*, and *error*.

The packet-in message allows a switch to yield control of a packet's routing to the controller. In most cases, this occurs when a packet fails to match any of the flow entries in the switch's flow tables. The controller uses the packet-in to calculate where the packet should be sent in the network and sends packet-out messages to reflect these new paths in the switch's flow tables.

Flow-removed messages tell the controller when a flow has been removed from a flow table in a switch. These messages are only generated for flows which have a flag set explicitly indicating this behavior. Port-status messages simply inform the controller of a change on one of the switch's ports. This includes ports being brought up or down, whether directly by a user or due to failure. Finally, the switch can notify the controller of any problems via error messages.

2.5.3.3 Symmetric Messages

Both switches and controllers send symmetric messages, hence their name. Like asynchronous messages, symmetric messages are sent without solicitation. These messages come in three flavors, two of which are relevant to this work. The switch and the controller exchange *hello* messages during the initial connection setup, shown back in Figure 2.10. Additionally, *echo* messages are sent between the two devices to verify the liveliness of the connection.

2.5.4 OpenFlow Switches

Vendor switches that support OpenFlow come in two varieties. The first type, OpenFlow-only switches, handle packets only with the OpenFlow pipeline of flow tables. The second type are OpenFlow-hybrid switches. These switches process packets with two possible pipelines: the OpenFlow pipeline and the normal pipeline. The normal pipeline refers to the normal operation of the vendor switch, whether that be layer 2 switching or layer 3 routing, etc. This section details the operation of OFSs and the OpenFlow pipeline.

2.5.4.1 Flow Tables

Every OFS contains at least one flow table containing flow entries. OFP provides the controller three options for managing these entries in its connected switches, called OFP Flow Commands (OFPFC). These commands include adding, removing, and updating flowtable entries; their functions are shown in Table 2.1 below. The controller can push changes both proactively (before receiving packets) or reactively (in response to incoming packets).

Table 2.1: OFP Flow Mod Commands

Command	Function
ADD	Installs a new flow.
MODIFY	Modifies all matching flows.
MODIFY STRICT	Modifies flows strictly matching wildcards and priority.
DELETE	Deletes all matching flows.
DELETE STRICT	Deletes flows matching wildcards and priority.

Each flow entry contains match fields, priority, counters, actions to perform on packets matching the match fields, timeouts, and a cookie. Match fields consist of an

ingress port, packet header fields, and other data which the flow table uses to match against packet contents.

The priority field dictates the matching precedence of the flow entry: entries with higher priorities are matched before those of lesser priorities. In the case of equal priority entries, the entries are matched in the (potentially arbitrary) order in which they appear in the table. Counters are updated when packets match flow entries, recording statistical data on the entries. Each entry has two timeouts, an idle timeout and a hard timeout, which determine when entries expire. The idle timeout tells the switch to remove the entry if it does not match a packet for a given duration, and the hard timeout tells the switch to remove the entry after a given duration regardless of match activity. Lastly, the cookie is a value chosen and used by the controller to identify specific flows; this value is not used in processing packets.

2.5.4.2 Matching

As mentioned in the previous section, each flow entry must have a priority assigned to it. This field is critical, as the table matches entries against the packet based on priority. A packet may match multiple entries in the table, but only the highest priority match will be selected. If a packet matches multiple entries of the same priority, then whichever entry is matched first will be selected. Table 2.2 shows a list of OpenFlow match fields relevant to this work. For a complete list of match fields, refer to the OpenFlow specification.

Table 2.2: OFP Match Fields

Field	Bits	Description
IN PORT	32	Ingress port; either physical of logical.
ETH DST	48	Ethernet destination MAC.
ETH SRC	48	Ethernet source MAC.
ETH TYPE	16	Ethernet type.
IP PROTO	8	IP protocol number.
IPV4 SRC	32	IPv4 source address.
IPV4 DST	32	IPv4 destination address.
MPLS LABEL	20	Label in the first MPLS header.

Flow entries can also have fully and partially wildcarded fields, meaning that the entry can match a packet on a near match. Essentially, this tells the flow entry that certain information in the packet is not necessary to the routing of the packet. This has multiple benefits. Entire fields in the entry can be ignored, *i.e.* an entry can match a packet by IP destination address but not IP source address. On the other hand, wildcarding allows field matches to be less specific; therefore, a single entry can match more flows and the size of the flow table is reduced. For example, the IP source field can be set to a subnet address *192.168.1.0/24* to match all traffic coming from a specific subnet rather than needing 254 flow entries for every host in the network.

2.5.4.3 Actions

When a packet matches a flow entry, the actions in that entry's action set are applied to the packet. Actions come in many distinct types, and not all actions must be supported by all vendor switches. Table 2.3 below shows actions relevant to this work. Again, the complete list of actions can be found in the OpenFlow specification.

Table 2.3: OFP Actions

Action	Required	Description
Drop	Yes	Not an explicit action, but rather the default when the action set is empty.
Output	Yes	Forwards a packet to the specified port.
Set-Field	No	Modify a field in the packet.
Push-Tag/Pop-Tag	No	Add, modify, or remove a VLAN or MPLS header.
Change-TTL	No	Modify IP or MPLS time-to-live (TTL) field.

A flow entry applies actions to the matched packet in the order in which they appear in the action list. Each action is immediately applied as it is reached in the list; that is, only the first action in the list is applied to the original packet and the results of the actions are cumulative. If the action list contains an output action, then that action should be the last action in the list. Any actions after the output action will be ignored.

2.5.4.4 Pipeline

Every OFS processes packets in a pipeline featuring sequentially-numbered flow tables, each containing multiple flow entries. It is worth noting that an OFS does not need multiple flow tables: a switch may only have a single flow table. In fact, the pipeline for single-table switches is greatly simplified. Figure 2.11 below outlines the pipeline of an OFS with multiple flow tables.

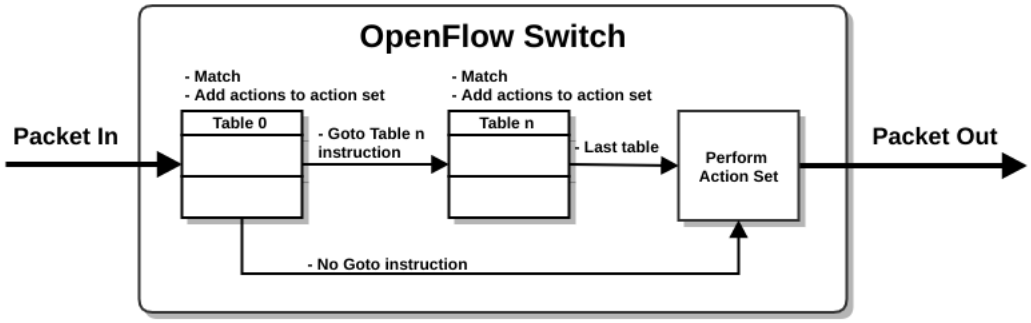

Figure 2.11: OFS Pipeline

When a packet arrives at the switch, it is directed to the first flow table. The flow table matches the packet against its flow entries. If a match is found, the flow entries actions are added to the action set. The entry set is empty when the packet first enters the switch. Each flow entry also has a set of instructions to perform when a packet matches. If the instruction set includes a Goto instruction, then the packet is sent to the flow table specified by the instruction. This continues until either the packet reaches the switch's last flow table or no more Goto actions occur. At this point, the action set is executed on the packet. The action set can only contain one action of a given type.

2.5.4.5 Table-Misses

When a packet does not match any flow entries in the flow table, a table-miss occurs. Each flow table must support a special flow entry to account for this situation. This table-miss flow entry has all fields wildcarded and low priority; thus, any other entry must be matched first. However, this entry behaves as a normal entry in that it may be added or removed by the controller and it can be set to expire.

When a packet matches only the table-miss entry, one of three actions typically occurs: the switch sends the packet (or part of the packet) to the controller, the

switch directs the packet to another flow table, or the switch drops the packet. If the switch does not have a table-miss flow entry, unmatched packets will be dropped by default.

Chapter 3

RELATED WORK

Since the introduction of OpenFlow, numerous tools and controllers have emerged to facilitate the design of software-driven networks. This chapter first introduces available SDN controllers and network virtualization tools. It then discusses how these tools have been employed in several SDN use cases.

3.1 SDN Controllers

The SDN paradigm depends on two components: a software controller and switches. Many vendors manufacture SDN-enabled switches; however, a suitable controller must be selected to realize an SDN architecture. In recent years, researchers have developed several SDN controllers; each has its own defining characteristics [28].

This section provides a brief overview of the architecture and OpenFlow support of four widely-used controllers: *POX*, *Ryu*, *Floodlight*, and *OpenDaylight*. All are modular and support OpenFlow; however, they are written in different languages and support different versions of OpenFlow [49]. This section is not intended to provide a thorough technical specification of these controllers. For that, please refer to each controller's specification documents.

3.1.1 POX

POX [37] is an open source, pure Python SDN controller which evolved out of the *NOX* controller (written in C++) [3]. Officially, POX supports only OpenFlow 1.0, limiting its utility. While POX's Python implementation makes it slower than other controllers, it also provides the benefits of rapid design and deployment. For this

reason, POX is often used in research networks but rarely used in production networks [26].

POX can be deployed on Windows, Linux, and Mac OSes. Immediately after installation, POX can be run as a Python script with no modification; however, running POX alone has little functionality. POX provides functionality by running *components*. POX components are event-driven applications defining network behavior, written in Python using the POX API. The POX API is built around a *core* object. When invoked, a component registers with the core, enabling components to communicate with each other and share resources. These components can parse and forward incoming packets, create and send OpenFlow messages, and listen to asynchronous events. POX is a competent OpenFlow controller, but its lack of documentation gives it a steep learning curve.

3.1.2 Ryu

Like POX, Ryu is component-based and written purely in Python, allowing for agile development [6]. Ryu supports OpenFlow versions 1.0 through 1.5, and is more actively updated than POX. It features an expansive collection of libraries and supports far more protocols than just OpenFlow.

The Ryu architecture is built around the *Ryu manager*, the main executable at runtime. Ryu applications are run on top of this manager, which runs a core process component that handles memory and event management. Ryu provides even more flexibility than POX by allowing components to be written in other languages. Overall, Ryu is feature-rich and more recently updated than POX.

3.1.3 Floodlight

In contrast to POX and Ryu, Floodlight is an Apache-licensed, Java-based controller for OpenFlow, making it better suited to production networks [4]. Like the controllers mentioned thus far, Floodlight is both a controller and a collection of applications built on the controller. The controller contains the logic to interact with and control an OpenFlow network. The applications, built as Java modules with Floodlight's REST API, provide user-defined functionality to the network. When running, the Floodlight controller's applications expose the REST API, allowing any other REST applications to invoke services. Floodlight is well documented and supports OpenFlow version 1.0 through 1.4

3.1.4 OpenDaylight

OpenDaylight [5] is the result of a community-led and industry-supported collaboration from the Linux Foundation, started in 2013 to encourage innovation in SDN. Since its inception, OpenDaylight has seen six releases. The most recent, *Carbon*, was released in June 2017. OpenDaylight is widely used in commercial networks [27], featuring in over 50 vendor services. It is available only on Linux and supports OpenFlow versions 1.0 through 1.3.

A *Model-Driven Service Abstraction Layer* (MS-SAL) resides at the core of OpenDaylight. The SAL processes interactions between network devices and applications, represented as objects. The SAL also provides a means of data exchange between the networking devices, represented using the YANG modeling language, a language which models configuration and state data manipulated by the Network Configuration Protocol (NETCONF). OpenDaylight is modular, providing network flexibility. It is also multi-protocol, supporting OpenFlow, BFP, OVSDB, and more.

3.2 Virtualization Tools

Network virtualization provides an abstraction of a network, decoupled from the underlying network hardware. Virtualization does not require SDN, and SDN does not require virtualization. Rather, SDN enables network virtualization, and virtualization facilitates the evaluation of SDNs. This section details two virtualization tools that are valuable to SDN research: *Open vSwitch* and *Mininet*.

3.2.1 Open vSwitch

Prior to switch virtualization, servers required physical connection to hardware-based switches. The creation of virtual switches removed this requirement, allowing servers to connect to a software layer rather than a physical switch. This section discusses Open vSwitch (OVS), a widely used virtual switch [45].

OVS is an open-source, multi-layer software switch that allows the forwarding functions of a switch to be extended via OpenFlow. While open to programmatic extension, OVS supports many standard protocols and management interfaces. Primarily written in C, OVS is portable and can be used both as a software switch on a *virtual machine* (VM) or as a control stack for switching hardware. These characteristics have led to widespread support for OVS; Table 3.1 below depicts notable platforms that support OVS.

Table 3.1: OVS Support

Platform	Supported
Operating Systems	Linux, Windows, FreeBSD, NetBSD, ESX
Virtual Machines	KVM, VirtualBox, Docker, Xen
Cloud Management	OpenStack, openQRM, OpenNebula, oVirt, CloudStack

The OVS architecture has three main components: *ovsdb-server*, *ovs-vswitchd*, and a *kernel module*. A database server, the ovsdb-server component monitors and updates the configuration of the switch. It communicates with a remote controller (often an OpenFlow controller) via the OBSDB protocol. The ovs-vswitchd component is a daemon that serves as the data plane of the OVS. This component handles the core networking functionality of the OVS. Like ovsdb-server, ovs-vswitchd connects to the remote controller; however, it uses OpenFlow protocol rather than OVSDB. This channel allows the controller to dictate packet forwarding via the installation of flow rules in the OVS. It also communicates with the kernel module using netlink.

The kernel module is responsible for the OVS's packet switching. It contains a cache to speed up the matching of packets to flow entries. When a packet enters an interface, the kernel module attempts to match it to a cached flow entry. If the kernel module cannot find a match, the packet is sent to user space. Figure 3.1 below depicts the three components in the OVS architecture.

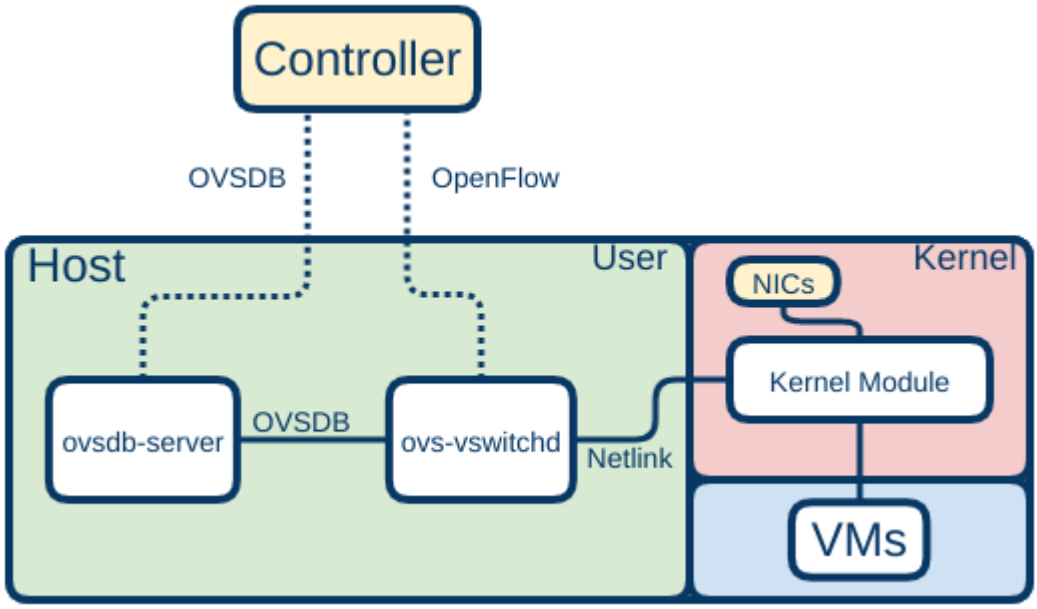

Figure 3.1: OVS Architecture

3.2.2 Mininet

Described as "a network in a laptop", *Mininet* provides researchers a means of rapidly prototyping large and varied network topologies in a single machine [34]. Mininet derives it scalability from its lightweight, OS-level virtualization, making use of processes and network namespaces (containers for network state). This removes the significant memory overhead of running each network device as a VM [33]. To illustrate how Mininet virtualizes a network topology, this section shows both a physical topology and how that topology is represented on a single machine.

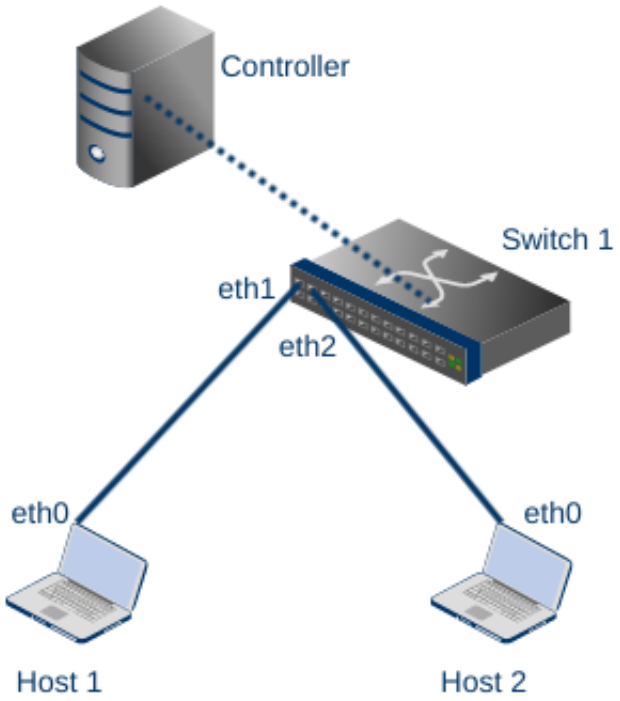

Figure 3.2: Physical Topology

Figure 3.2 illustrates a network topology featuring two hosts and a switch, managed by an OpenFlow controller; Figure 3.3 shows how Mininet creates that topology. Each host is a shell process in its own network namespace. Each namespace provides its shell process exclusive ownership to virtual Ethernet interfaces and routing tables. A pipe from the core Mininet process *mn* to each shell process allows Mininet to communicate with and monitor each host. Mininet utilizes software switches for packet switching. This topology, for example, employs an OVS to simulate an L2 switch. Mininet represents each link in the topology as a virtual Ethernet (veth) pair, which acts like a wire between two fully functional interfaces. Lastly, an OpenFlow controller (either in the machine or remote) updates the flow entries in the OVS to manage packet switching on the network.

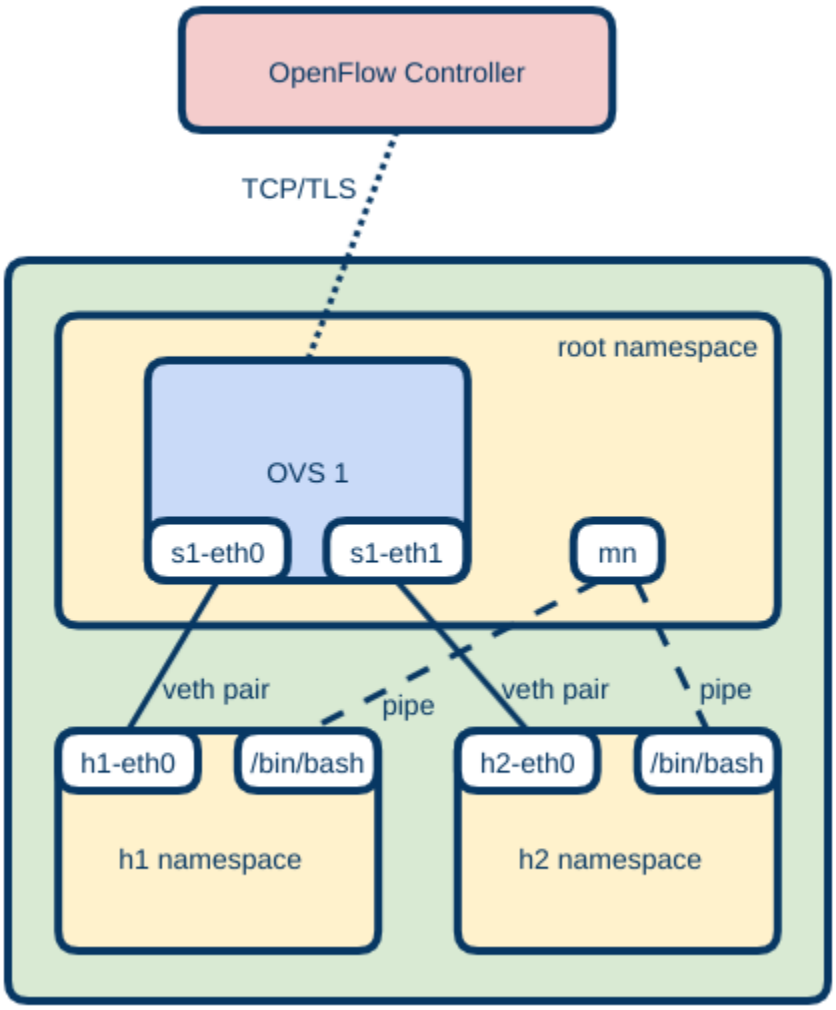

Figure 3.3: Mininet Implementation of Physical Topology

After running a topology, Mininet provides the user a command line interface (CLI). The CLI enables users to run commands on hosts, adjust the topology and run diagnostics. Additionally, Mininet features a programmable Python API, allowing topologies and tests to be created as Python scripts and run directly from the command line. Overall, Mininet is scalable and portable; its ability to be loaded onto a VM and run on any Linux machine makes it a valuable tool for SDN researchers.

3.3 SDN Applications

Given its ability to eliminate the need for middleboxes and provide rapid deployment of new services, researchers have applied SDN to several networking environments. This section surveys SDN solutions.

Wide Area Networks are networks across large distances, typically carrying large amounts of traffic within locations of an organization. Thus, WANs must demonstrate both high performance and high reliability, avoiding congestion and service outages. Traditionally, network designers met these requirements using high-end routers provisioned to low average link utilization (30-40%), leading to both wasted resources and high costs [25]. Software-defined WANs (SD-WANs) address both of these issues by providing unified network management to simpler hardware.

Google's *B4* SD-WAN exemplifies the benefits of SDN in WANs [25]. B4 is a private WAN that connects all of Google's data centers around the globe. As one might assume, this network has significant bandwidth demands. Using OpenFlow and simple routers built from merchant silicon, B4 allows Google's WAN to operate at near 100% utilization via traffic engineering. Importantly, B4 employs a hybrid approach, supporting both existing protocols and new traffic engineering services. The research shows that B4 succeeds in maximizing link utilization; however, it also shows that bottlenecks in control-plane to data-plane communication pose a challenge to SDN deployments.

In recent years, researchers have applied the SDN paradigm to the problems of IP mobility [35] [29] [54] [36] [46] [53]. This research has taken the form of both SDN implementations of existing protocols and entirely new protocols built using SDN.

In [35], Lee et al. surmise that existing approaches to IP mobility are not easily applied to real network topologies due to the difficulty in updating numerous dif-

ferent network devices. They proposed a PMIPv6 implementation over SDN using OpenFlow, and concluded that the proposed approach circumvents the limitations of non-SDN approaches. Similarly, Kim et al. implemented an OpenFlow-based PMIPv6 architecture and found that the SDN approach provided a more flexible PMIPv6 solution [29].

More recently, researchers have been creating their own SDN-based mobility protocols. In [54], Wang and Bi propose a new mobility protocol employing OpenFlow switches with the POX controller. They provide both simulated and practical validation, suggesting that their implementation has advantages over existing work in route efficiency and handoff time.

Chapter 4

DESIGN

College campus networks are a special use case of enterprise networks, sometimes called *campus area networks* (CANs). Multiple interconnected local area networks (LANs), limited to a specific geographical location (a campus), comprise a CAN. This area includes all of the campus's buildings, whether administrative, academic, athletic, etc. The CAN links LANs throughout the buildings of a campus. This work classifies Cal Poly's campus network as a CAN and treats the Cal Poly CAN as representative of a typical CAN.

Unlike many WANs or datacenter networks, CANs feature the unique challenge of accounting for many different hosts moving throughout the campus, roaming from one LAN to another as they go. This mobility is due to the widespread use of mobile computers and cellphones by faculty and students. Both groups have similar data needs: support for video and music streaming, phone calls over IP, and messaging services, etc. While moving between IP subnets on campus, these services will be disrupted if hosts need to obtain new IP addresses. Thus, the need for IP mobility on the CAN is clear.

Cal Poly's CAN is built from networking devices, such as legacy switches and routers, middleboxes, etc. that are not configured for mobility. The standard option would be to employ mobile IP (MIP) on the CAN, but this creates several problems. Triangle routing poses the problem of inefficient routing and wasted bandwidth, and slow handoffs may result from MIPs two-stage handoff process. MIP extensions have been proposed to solve these problems, but they require specialized modifications to the protocol stacks of network devices [15]. This requirement is not ideal in the CAN due to the variety of vendor devices on the network.

Due to its improved control over network resources, SDN can be applied to solve the mobility problems in CANs. This work proposes an OpenFlow-based Software-Defined Mobile CAN (SD-MCAN) architecture with inherent mobility support. By enforcing mobility policy from a centralized controller, SD-MCAN eliminates routing inefficiencies and minimizes handoff latency. Additionally, the utilization of SDN removes the administrative hassle of managing proprietary software across different vendor devices. This section begins with a discussion of the motivation and design considerations informing the development of SD-MCAN. It then details the SD-MCAN architecture and routing scheme.

4.1 Design Considerations

Four design considerations shape the design of the proposed SD-MCAN architecture: 1) *compatibility*, 2) *route efficiency*, 3) *handoff latency*, and 4) *scalability*. This section introduces these considerations and the problems that motivate them, providing rational for the design decisions detailed in the next section.

4.1.1 Compatibility

As institutions of learning, colleges are impacted by budgetary restrictions; their networks are as well. Thus, any proposed networking solution must be cost-effective. Like all CANs, Cal Poly's network is composed of many different LANs connected through a central mesh of routers. This physical topology is built from switches and routers spread throughout the buildings on campus and interconnected by both fiber-optic and copper links. These devices are incorporated into the campus infrastructure; thus, any SDN system requiring modification to the pre-existing physical topology would be both expensive to implement and disruptive to the network's normal operation.

Another consideration is the number of different protocols currently in use on the CAN. The proposed solution enables mobility between the many LANs on campus; however, it must also be compatible with any layer 2, layer 3, and layer 4 protocols currently in use. The system's operation must be fully compatible with existing networking protocols so as to not disrupt operation within the connected LANs. In this way, the mobility solution is transparent.

In summary, for any SDN-based routing solution to be deployable on CANs, it must:

1. be cost-effective
2. be readily deployable over the existing topology
3. support existing protocols

4.1.2 Route Efficiency

The efficiency of assigned routes must also be considered in the design of any mobility system. If the system does not optimize the routes given to mobile host flows, then packets could end up routed across far more physical distance than necessary. This inefficiency results in increased latency for mobile hosts on the network and increased network congestion.

MIP's triangle routing problem, described in Chapter 2, provides a perfect example of how inefficient routing could exist on Cal Poly's CAN. Suppose a student first connects to the campus network at one end of campus, making that first hop router her home agent. The student then walks to her next class in a building on the other side of campus, where she tries to send a file to her friend using File Transfer Protocol (FTP). Supposing her friend's home network is in that same building, the file must travel all the way across campus to the first student's home agent and be

tunneled back to the classroom in order to send a file between students that may well be sitting a few feet away from each other.

4.1.3 Handoff Latency

CANs host many different services over IP traffic, including video streaming, music streaming, and VoIP. While low-bandwidth applications are resilient to short disruptions in IP connectivity, the aforementioned high-bandwidth services require more reliable connections, otherwise connection interruptions become apparent to the end user. The aim of any mobility system should be to provide low-latency handoffs so that an end user does not notice an interruption in service even if they are moving between LANs. Ensuring fast handoffs requires both efficient routing and avoiding congestion at the SDN controller. Congestion at controller interfaces results in increased handoff latency as it increases the time required to install new flows for a mobile host; thus, the presented design intends to achieve fast handoffs via minimizing controller congestion.

4.1.4 Scalability

As previous sections have detailed, an OpenFlow-based SDN architecture has two components: a controller and switches. The controller manages routing by installing flow entries into the flow tables of the connected switches. This leads to several problems when the network scales. In regard to CANs, the limiting factor is the number of hosts in the network. The campus topology is stable: it will not change save for a new switch installed every once in a while. Thus, topology changes will never overload the controller. However, as the number of connected (and mobile) hosts increases, so does the number of required flows in each switch and the demand on the controller. If the controller becomes overloaded with packet-ins from the switches,

then performance will suffer significantly. Colleges often have thousands students on campus; a mobility system must scale with host demand.

Therefore, to scale to the required number of hosts, a mobility system must:

1. minimize congestion at controller interfaces
2. quickly install new flow rules
3. limit the number of flow entries in switches

4.2 Architecture

With the design requirements specified, this section details how the design of SD-MCAN satisfies these requirements. This section begins with an overview of the proposed SD-MCAN architecture. It then details the function of the architecture's components and how they communicate with each other. Finally, it discusses the policies dictating how traffic is routed throughout the network.

4.2.1 Assumptions

In pursuit of simplicity, several assumptions are made in the examples presented in this section. The examples assume that the CAN only supports IPv4 traffic. Additionally, the hosts used in the examples represent a single interface on a host; this representation is useful for depicting host movement. These assumptions apply only to the presented examples: they are not enforced in the system design. The system can easily support both IPv4 and IPv6, as well as multiple interfaces per host.

4.2.2 Overview

SD-MCAN is an SDN-based system, built on OpenFlow, that replaces the core of a campus network. The SD-MCAN topology maps directly to the existing topology of the underlying CAN, and the system requires little modification to existing networking devices. A server is added to the core of the network; this server runs the OpenFlow controller that manages the network. The network devices at and near the core of the CAN become OpenFlow switches via software updates to vendor hardware. Outside of the core, nothing changes for the LANs on the existing network. From the view of the LANs, and all the hosts on them, SD-MCAN behaves exactly like the current legacy routers; the key difference is that SD-MCAN supports IP mobility.

Devices at the core of the network are assigned either the role of *Edge Router* (ER) or *Core Router* (CR). CRs form a complete mesh at the center of the network; they do not connect directly to the LANs. Rather, their purpose is to route IP traffic between the ERs based on MPLS *labels*, essentially forming an MPLS tunnel across the network core. ERs connect the core mesh to the LANs. The ERs serve two purposes: pushing MPLS labels onto ingress traffic and popping them off of outgress traffic.

When a packet enters the network, the LAN of the destination host is determined and a label is added to the packet in the form of an MPLS header. The controller labels traffic according to the destination host's *current* location in the network. If a host is found to be mobile, the controller updates the host's current LAN: the routing process does not differentiate between static and mobile hosts.

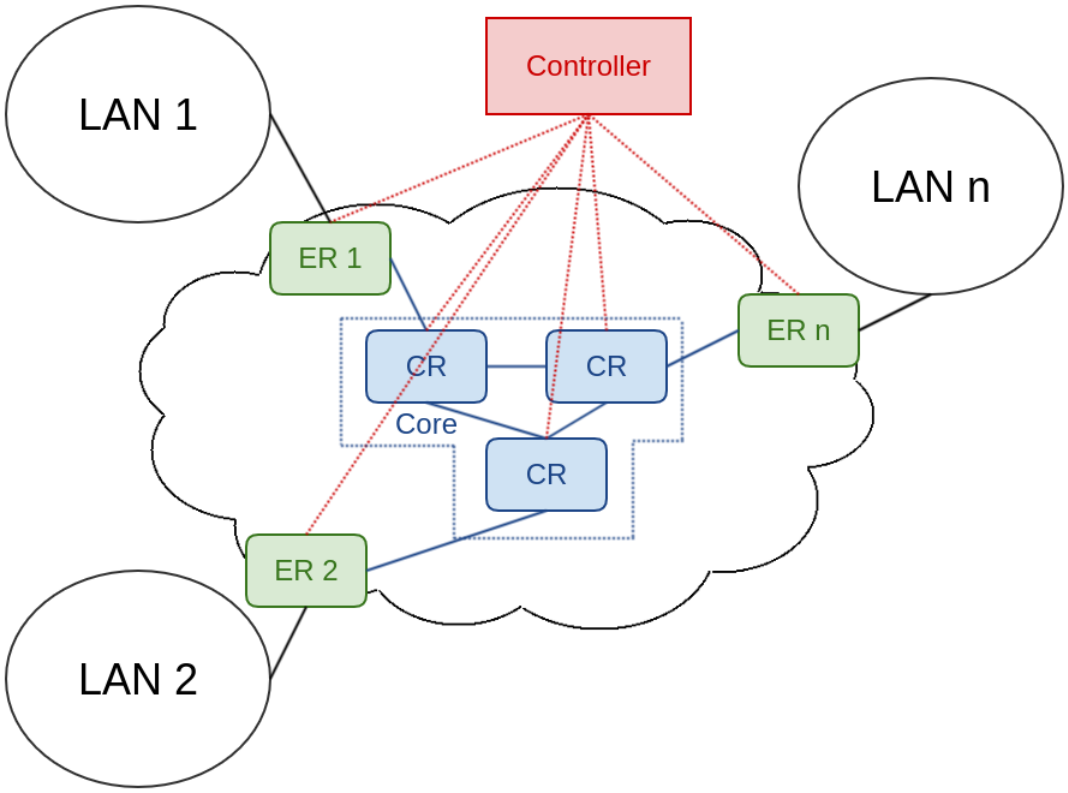

Figure 4.1: Network Architecture

Figure 4.1 illustrates the overall architecture of SD-MCAN. It must be noted that ERs and CRs do not need a one-to-one ratio, nor does each ER need to connect only to a single LAN. The core can have an arbitrary number of switches, and multiple LANs can connect to a single ER. In this sense, both ERs and CRs can be linked like legacy routers.

4.2.3 Components

The SD-MCAN OpenFlow controller manages the flow of traffic in the network using complementary components. The controller contains three modules, each providing its own functionality: a *Topology Tracker* (TT), a *DHCP Server* (DS), and a *Route Manager* (RM). The TT interacts with OpenFlow to deliver a complete view of the

status of the OpenFlow switches and the location of hosts in the network. The DS behaves like a standard DHCP server, allocating IP addresses to hosts across all of the network's LANs; however, it features the added functionality of managing mobile hosts on the network. Finally, the RM leverages information provided by the other modules to install flow rules to the switches on the network. Figure 4.2 below shows the components of the OpenFLow controller at the heart of SD-MCAN. This section covers each component in detail.

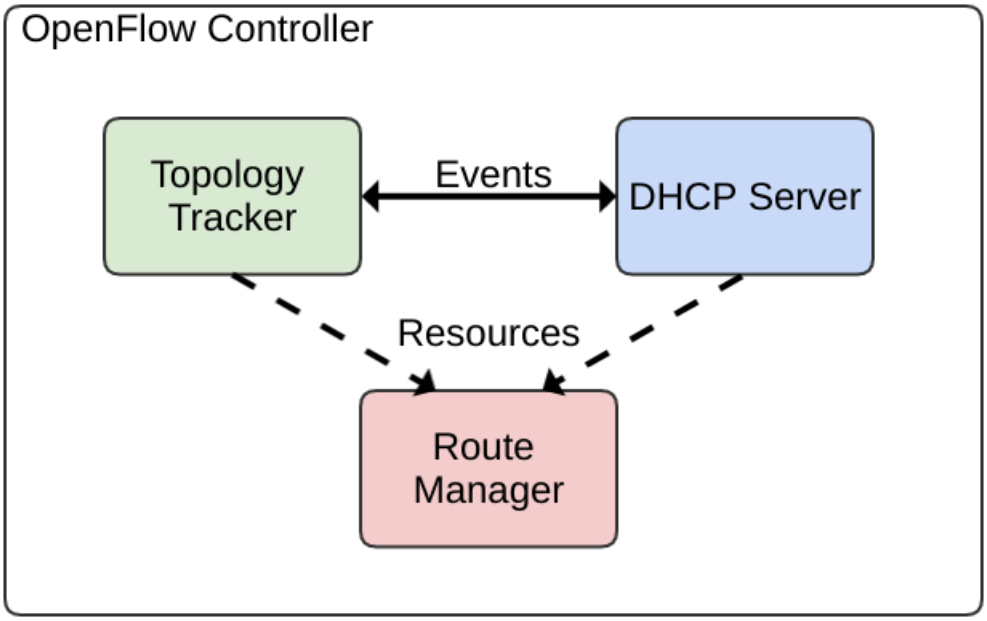

Figure 4.2: Controller Architecture

4.2.3.1 Topology Tracker

TT is responsible for storing and updating the controller's knowledge of the network topology. This component uses information provided by OpenFlow to create a graphical representation of the switch topology, where switches are nodes and links are edges. These edges also contain information about which switch ports are used in the link.

OpenFlow makes tracking switch locations and links straightforward by providing the controller topology updates via three *events*: *Connection Up* (CU) events, *Connection Down* (CD) events, and *Port Status* (PS) events. Figure 4.3 shows how network information reaches TT via events. When an OpenFlow switch completes its initial connection to the controller, the controller raises a CU event. TT hears the event and obtains the switch's datapath identifier (DPID) and connection interface.

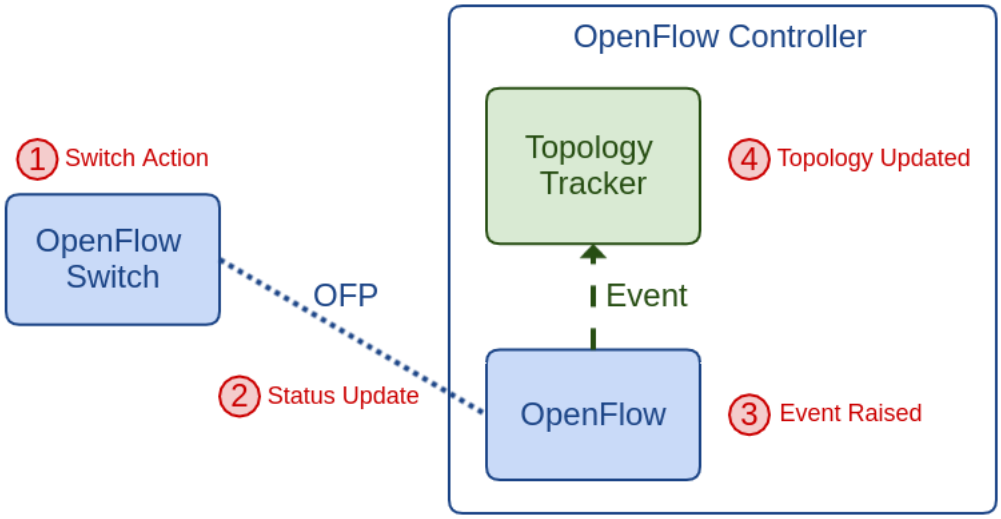

Figure 4.3: TT Topology Update Process

Each switch has a DPID to uniquely identify it to the controller and a connection interface out of which the controller can communicate with the switch. TT takes this information and stores it as a node in the network graph. After registering the switch, TT immediately installs two flow entries into the switch's flow table. A table-miss entry specifies that the switch should send unmatched packets to the controller, and a special ARP-based entry allows TT to check the liveliness of hosts (more on this below). Inversely, when a switch disconnects from the controller, TT uses the resulting CD event to remove the switch's node from the network graph. When a switch-to-switch (SS) link is added to the network, OpenFlow registers a PS up event

containing the link switch and port information. Using this information, TT adds a new edge to the graph. Similarly, when a link goes down, OpenFlow registers a PS down event, and TT removes the edge from its graph.

OpenFlow provides detailed information about the status of switch-to-switch links and ports; however, it provides no default support for locating hosts. TT leverages OpenFlow *Packet-In* (PI) events to track the location of hosts around the network. When a packet fails to match any flows entries in a switch, the switch forwards that packet to the controller as a packet-in. OpenFlow then raises a PI event; by listening for PI events with a high priority, TT receives the packets first as they enter the controller.

Upon receiving the packet-in, TT first parses the packet's Ethernet header. TT looks at the packet's source MAC address to determine the source host. If TT does not have an entry for this host in its topology graph, it creates an entry for the new host. This entry contains the host's MAC address and the DPID of the ER to which the host is connected. If TT already has an entry for the host, TT checks whether the host has moved by looking at the ER at which the PI arrived. If the new ER does not match the ER in the host's entry, then TT updates the host's entry with the host's new location.

After discovering or updating the host, TT parses the packet's next header. If this next header is an IP header and contains a DHCP packet, then TT forwards the packet to the DS component. Otherwise, TT forwards the packet to the RM. Figure 4.4 below shows the flow of this process.

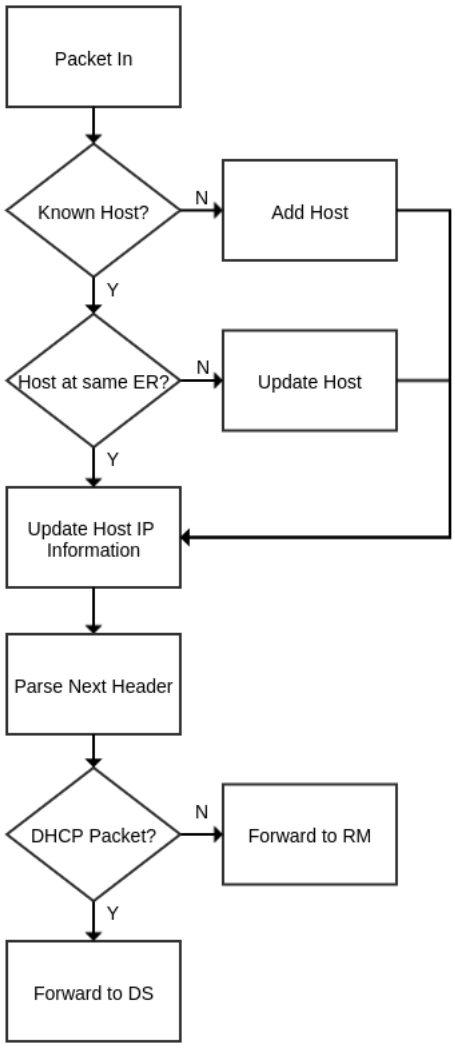

Figure 4.4: Topology Tracker Packet-In Processing

While switches automatically send liveliness updates to the controller over OFP, the controller must manually check host liveliness. Monitoring host liveliness prevents the system from unnecessary re-learning of host information while also removing unneeded host entries in memory. This results in a more stable network graph at the cost of a slight overhead. TT check host liveliness by periodically sending ARP requests to known hosts via packet-out messages from their connected ERs; these ARP requests contain a special address as the MAC source in their ARP header.

When a host responds with an ARP reply with the special address as the destination MAC, the ER sends the response directly to the controller, matching a special flow entry installed when the switch first connects to the controller. Figure 4.5 shows the match and action fields of this flow entry. At this point, TT updates the host's liveliness information. If a host fails to respond for a given *host liveliness interval*, then TT marks the host as expired and removes the host from the network graph.

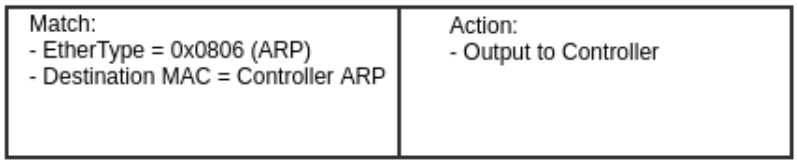

Figure 4.5: Topology Tracker ARP Flow Entry

TT provides one final service to SD-MCAN: it monitors the stability of the network. The network is considered *stable* if no new switches have joined the network and no new SS links have been added to the network in a given period of time, called the *stability interval*. When the controller starts up, it handles all of the switch CU events and PS events for the currently connected devices. Once these switches have stopped joining the network, TT will alert the DHCP server that the topology is stable. The addition of host-switch links does not impact the stability of the network.

4.2.3.2 DHCP Server

When TT knows the network topology to be stable, the DS component initializes. The DS component largely behaves like a typical DHCP server, allocating IP addresses to hosts on the network from an address pool; however, DS also discovers mobile hosts. Like TT, DS installs a flow entry into ER flow tables as soon as they connect to the controller, seen in Figure 4.6. This entry ensures that all DHCP traffic goes to the

controller. In this way, ERs function as DHCP relays in SD-MCAN, allowing DS to serve IP addresses to all LANs on the network.

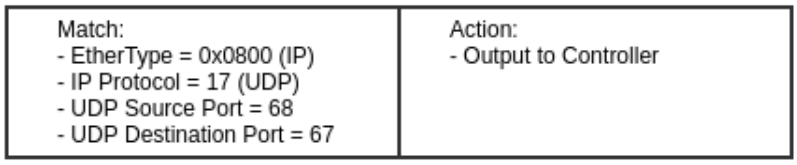

Figure 4.6: DHCP Flow Entry

A host cannot send packets over IP without a valid IP address. Therefore, each host must broadcast DHCP discovery packets to obtain a valid IP address upon connecting to an SD-MCAN ER. As mentioned in the previous section, TT receives these packets and, upon identifying them as DHCP packets, forwards them to DS. DS allocates an IP address from the IP address pool and creates DHCP packets, which OpenFlow then forwards to the ER as a packet-out. The ER then communicates with the host and completes the DHCP lease process. Figure 4.7 depicts this initial lease process in SD-MCAN.

Additionally, DS keeps a mapping from each host interface to its IP address. When a host migrates to a new LAN, it sends a new DHCP discovery. When DS receives this packet, it recognizes that the host currently has another IP address leased to it from another LAN's address pool. To prevent allocating the host interface multiple addresses, DS marks the interface as mobile and extends its current lease. This provides SD-MCAN a key benefit over existing mobility approaches: SD-MCAN does not require multiple addresses per host interface.

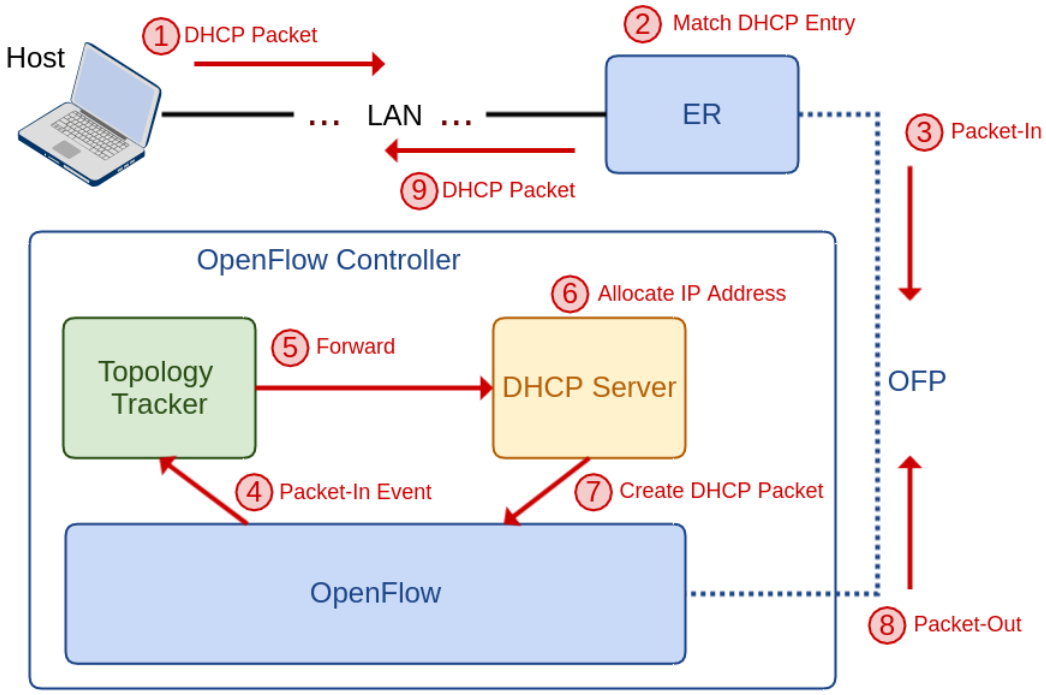

Figure 4.7: DHCP Server Packet Flow

Figure 4.8 illustrates how DS processes incoming DHCP packets. Through this behavior, DS enables SD-MCAN to make better use of its address space by preventing interfaces from holding multiple addresses across LANs. Anytime DS issues a DHCP lease, it notifies TT of the interface's IP address change.

Like a typical DHCP server, DS periodically checks for lease expiration. DS stores information about each IP lease in a lease table, including the duration of the lease and when it will expire. If DS finds an expired entry, it adds the expired IP address back into the address pool. If the IP address belonged to a mobile host interface, then DS unmarks the interface as mobile.

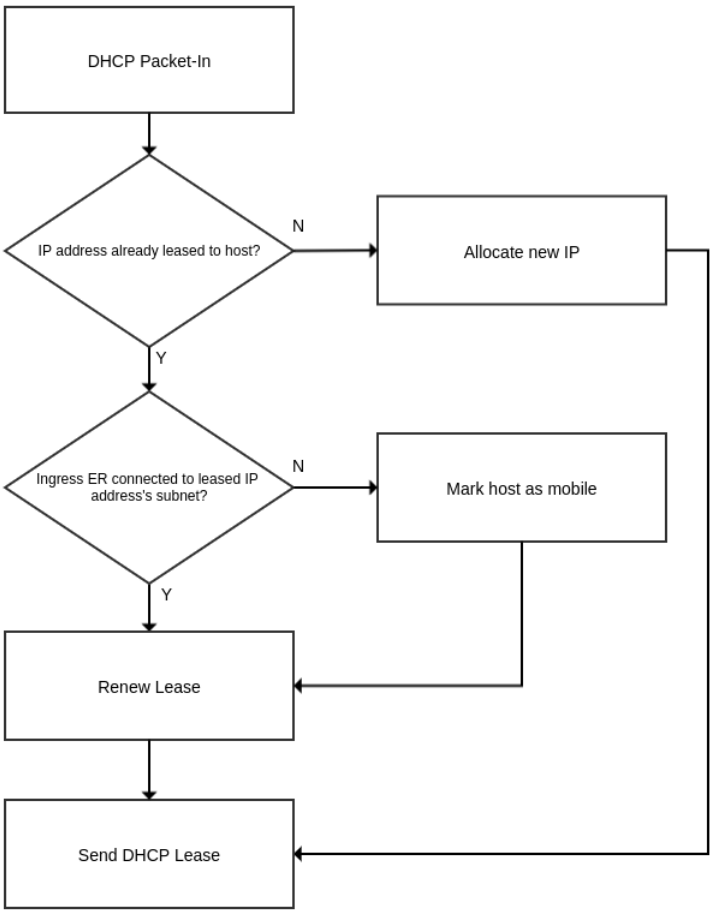

Figure 4.8: DHCP Server Packet-In Proccessing

When a host attempts to send IP traffic to a destination outside its LAN, it will send the traffic to its default gateway. Typically, a host's default gateway is the router interface to which the LAN is directly connected. In SD-MCAN, the OpenFlow switches responsible for routing packets through the core network do not have interface-specific IP addresses. Rather, DS assigns each LAN-facing ER interface a *fakeway* IP address. When a host leases an IP address from DS, the resulting DHCP lease packet contains information about the host's default gateway IP address: DS inserts the fakeway address here. When a host ARPs for its default gateway, an ER receives the ARP packets and respond with a designated dummy MAC address. In this way, SD-MCAN simulates normal router operation.

4.2.3.3 Route Manager

The RM utilizes a complete view of the network, provided by TT and DS, to manage all packet routing on SD-MCAN. RM implements hybrid, label-switched routing, similar to MPLS. In fact, RM uses MPLS tagging to enable label-switching on the network; however, whereas MPLS routers exchange label information using a distributed routing protocol (LDP or RSVP-TE), RM centrally manages the labeling of all routes on the SD-MCAN network.

As mentioned earlier, the OpenFlow switches on the network function as either CRs or ERs. RM utilizes a hybrid routing scheme: the controller updates CR flow entries proactively and ER flow entries reactively. ERs provide two functions: they push labels onto ingress traffic and they pop labels off of outgress traffic. Like a normal MPLS network, the hosts on SD-MCAN never encounter labeled traffic; labels are only used within the core network. Once TT determines that the network is in a stable state, RM analyzes the core mesh of CRs and assigns a link-local label to each core link. Each label represents part of a unidirectional path to a specific destination LAN.

When a packet from a new flow enters an ER, it will match only the default flow entry, so the ER sends a packet-in message to the SD-MCAN controller via OpenFlow. TT receives the packet and performs any host identification it needs before passing the packet to RM. RM first verifies that the packet's destination MAC address is either the designated default gateway dummy MAC address or a valid host before ensuring that the packet is an IP packet. After verifying the packet, RM queries TT and DS to obtain the current location of the destination IP host on the network. If RM finds the host on the network, RM looks up the correct label for the packet and the MAC address of the destination host (if the ingress packet was addresses to the gateway dummy MAC). RM then creates and installs a flow entry for the packet flow

into the ERs flow table.

ERs contain two types of flow entries: *push* entries and *pop* entries. Push entries match on three fields. They match the destination MAC address, EtherType, and destination IP address fields. The destination MAC address should be either a host MAC (in the case that the destination host's home LAN is the same as the source host) or the dummy default address MAC. The EtherType will be IP, and the destination IP address will be the IP address of the destination host. All other fields are wildcarded. Figure 4.9 below shows the match fields and actions of a push flow entry.

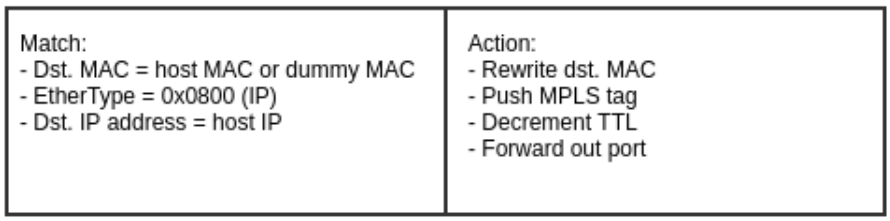

Figure 4.9: Push Flow Entry

A push flow entry also contains four actions in its action list. In the case where the incoming packet's destination MAC is the dummy default gateway MAC, the entry rewrites the destination MAC address to the actual MAC address of the destination host. In this way, SD-MCAN simulates legacy router behavior. Next, the flow entry pushes the appropriate label onto the packet as an MPLS tag. RM determines this label by looking up the source host's next hop ER interface and the current LAN of the destination host. Then, the IP TTL is decremented. Finally, the flow entry forwards the modified packet out the appropriate core-facing interface. Push flow entries are not permanent; they expire if no packet matches the entry after a specified idle timeout. This way, unused flows are quickly removed from flow tables, decreasing the size of the flow tables.

ERs also contain pop flow entries. Pop entries are simpler than push entries, and the ER contains less of them. Pop entries match two fields: EtherType and MPLS label. EtherType is once again IP, and the label corresponds to one of the ERs connected LANs; all other fields are wildcarded. The label lets the ER know out of which interface to forward the packet. The pop flow entry holds four actions as well. It pops the MPLS label off of the packet before decrementing the IP TTL, rewriting the Ethernet checksum, and forwarding the packet out the interface facing the destination LAN, signified by the label. Figure 4.10 shows a pop flow entry. Like push entries, pop entries expire if not used.

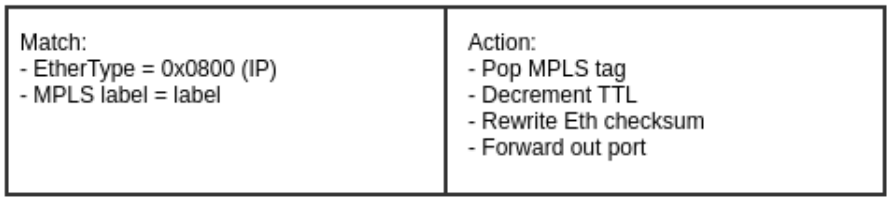

Figure 4.10: Pop Flow Entry

CRs also contain flow entries; however, CR flow tables are not updated nearly as often as ER flow tables. While an ER updates its flow table with every new flow it sees, CRs only receive flow table updates when a new LAN joins the network. Importantly, this limits the load on the SD-MCAN controller because the controller rarely sends or receives packets with the CRs. When TT finds the topology to be stable, RM creates a unique label for each CR-to-CR link for each destination LAN. RM then installs all of the required flow table entries at once into the CRs. CRs route packets entirely based on the labels pushed by ERs. Figure 4.11 below shows the format of a CR flow entry. These flow entries do not have idle timeouts like ER flow entries.

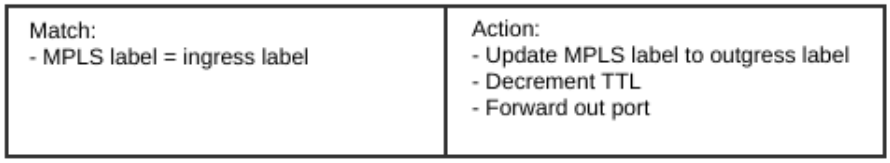

Figure 4.11: CR Flow Entry

Figure 4.12 below outlines an algorithm by which RM could install CR flow entries. The purpose of CR flow entries is to route packets between all LANs on the network using only label-switching. To this end, this algorithm computes the shortest path between every pair of LANs using Dijkstra's algorithm. The routing metric used is implementation-specific, allowing for potential traffic engineering. For each pair of switches along the path toward a LAN, a unidirectional label is allocated. If a label already exists for that link-LAN pair, then the algorithm simply retrieves that label. Each flow entry has two defining fields: an *ingress* label to match against packets and an *outgress* label to add to the packet before forwarding it along the path. The ingress label is simply the label assigned to the previous link in the path, and the outgress label is the label to be applied to a packet before sending it to the next link along the path. The algorithm does not install flow entries into the first or last switch along the path: this is because these switches are ERs.

```
Algorithm CRFlows( G )
// Allocates labels and installs flow entries for all CRs
on the network.
// Input: G the topology graph
// Output: None

for each pair of LANs L1, L2 where L1 != L2:
  p = ShortestPath( L1, L2 )
  inlabel = None
  for i in ( 0, len( p ) – 1 ):
    link = ( p[ i ], p[ i + 1 ] )
    outlabel = GetLabel( link, L2 )
    if i > 0:
      Add inlabel -> outlabel flow entry in L1 flow table
    inlabel = outlabel
```

Figure 4.12: CR Flow Entry Algorithm

When a CR receives a packet flow, the CR checks only the label on the packet. Due to the pre-installed flows, the CR automatically knows what label to place on the packet and out of which port to forward the packet in order to route it to the proper LAN. This approach to flow table management offers multiple benefits. First, this approach prevents CRs from ever sending packet-in messages to the controller, lessening demand on the controller and alleviating congestion. This also makes packet routing in the core much faster because flow misses will never occur (save for link malfunctions) and the flows will always take the efficient paths to their destination LANs.

Additionally, label-switching in the core allows the core to scale well. OpenFlow-enabled vendor switches have limits to how large of flow tables they can support. Typically, a switch can support more L2 flows than L3 flows, as L3 flows require more memory. By employing label-switching at the core, SD-CAN allows CR flow tables to occupy less memory in the underlying switch hardware, providing space for

more flows. Finally, using L2 flow entries allows the core to route packets faster, as switches can match packets against L2 flows faster than they can L3 flows. Figure 4.13 shows an example of a flow routed across the SD-MCAN core, while Table 4.1 shows the contents of packet header fields at each hop.

Table 4.1: Flow Packet Fields per Hop

Hop	DST MAC	MPLS Tag
1	f0:00:00:00:56:b1	None
2	9e:3b:d5:f1:d8:3d	16
3	9e:3b:d5:f1:d8:3d	17
4	9e:3b:d5:f1:d8:3d	18
5	9e:3b:d5:f1:d8:3d	None

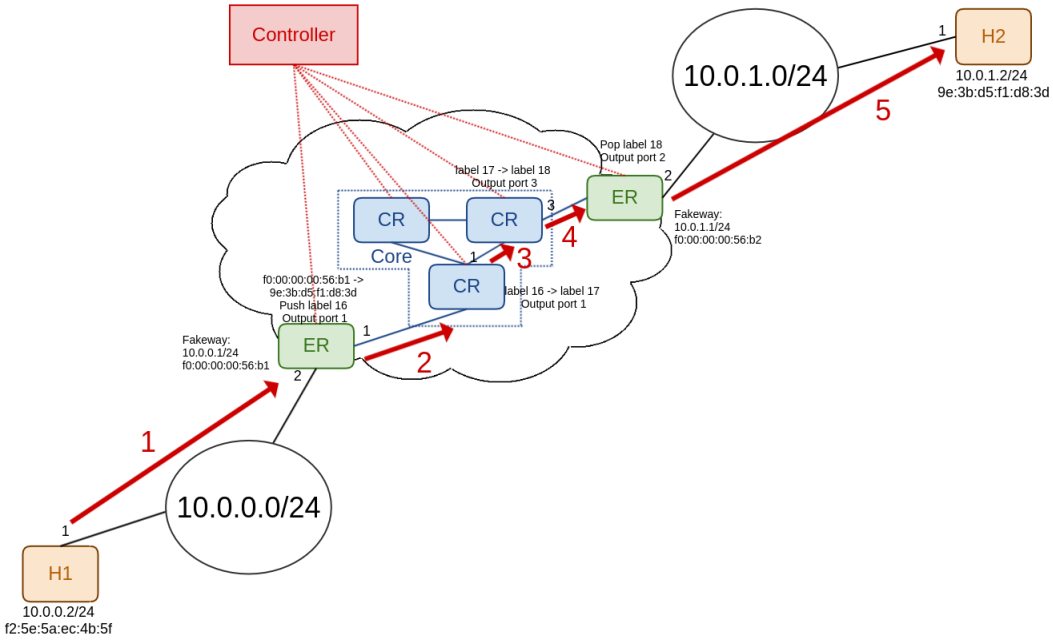

Figure 4.13: Label-switched Flow

4.2.4 Routing

With the core SD-MCAN components established, this section illustrates how SD-MCAN routes traffic flows across the network. The section presents several routing examples, on simple topologies, featuring both static and mobile hosts. The examples detail how SD-MCAN handles communication setup between hosts and the migration of mobile hosts around the network.

4.2.4.1 Connection Establishment

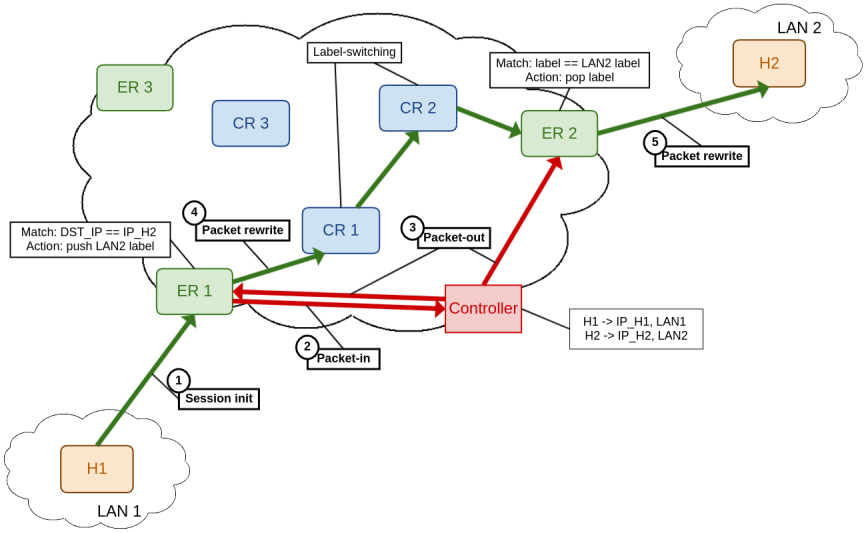

Figure 4.14: Communication Establishment between H1 and H2

Figure 4.14 depicts the establishment of a TCP session between two hosts, H1 and H2, on LAN1 and LAN2 respectively. The figure presumes that both hosts already leased an IP address from the controller, and therefore the controller knows the location of both hosts. H1 starts the session by sending the first TCP SYN packet to its first-hop switch ER1. The packet does not match any flow entries in ER1, so ER1 forwards

the packet to the controller as a packet-in. The controller sees that the packet is destined for H2, which is on LAN2. Thus, the controller updates ER1's flow table with an entry that matches packets addressed to H2 and pushes an MPLS label onto those packets signifying that the packet should be routed to LAN2. The controller also updates ER2's flow table with an entry that matches the LAN2 MPLS label. When a packet matches that label, ER2 pops the label and forwards the packet out the LAN2-facing interface.

After adding the two flow entries to ER1 and ER2, ER1 pushes an MPLS label onto the packets and forwards the TCP stream toward the network core. The CRs route the flow to ER2 using pre-installed flow entries, and ER2 pops the MPLS label off the stream before forwarding the stream to LAN2.

4.2.4.2 Host Migration

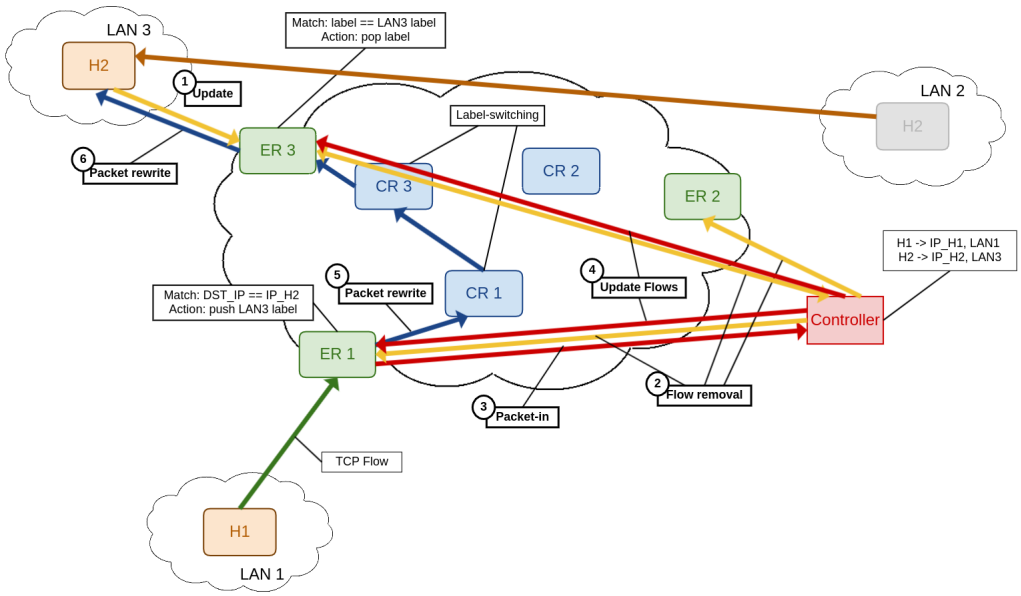

Figure 4.15: H2 LAN Migration

Figure 4.15 illustrates SD-MCAN behavior when H2 moves from LAN2 to LAN3 during the same TCP session. When H2 connects to LAN3, it sends out a new DHCP discovery packet to retrieve an IP address on the new network. ER3, now the closest switch to H2, receives the packet and forwards it to the controller. The controller recognizes H2 and leases H2 the same IP address it had on LAN2 (Note: had the lease on H2's LAN2 IP address expired, H2 would be leased a new LAN3 address). The controller then updates H2's entry to reflect its new location on LAN3 before sending flow removal packet-out messages to all ERs, removing any push flow entries for H2.

Meanwhile, H1 continues to send TCP packets to H2 during H2's migration to LAN3. The controller removes the push flow entry in ER1, so ER1 sends the next ingress flow packet to the controller, which repeats the connection establishment process with the knowledge that H2 now resides on LAN3. The connection suffers minimal interruption (on the order of milliseconds), requiring minor TCP retransmission as the controller adds flow entries in ER1 and ER3.

These examples illustrate how SD-MCAN routes IP flows and supports mobility in the network by tracking the location of hosts. The presented routing approach offers several benefits. By pre-installing CR label-switched flow entries, the controller ensures that every route takes the shortest path between LANs without having to compute shortest paths during handoffs, reducing handoff latency. The utilization of label-switched routing results in smaller core flow tables, faster flow-table lookups, and lower memory requirements on switch hardware. Additionally, aggressive flow removal during mobility handoffs prevents ER flow tables from being temporarily bloated during mobility handoffs.

Chapter 5

IMPLEMENTATION

Several SDN technologies exist with which SD-MCAN may be implemented. The selection of an appropriate implementation depends on multiple factors, including compatibility with available networking hardware and ease of prototyping. This chapter discusses the rationale behind the currently deployed SD-MCAN prototype. The selected implementation allows for rapid prototyping via compatible virtualization tools while also being well-suited to the networking hardware available at Cal Poly.

5.1 Controller

SD-MCAN is an OpenFlow-based SDN solution; thus, an OpenFlow controller must be used in its implementation. Chapter 3 introduced four established controllers, all of which were considered for use in this implementation. However, this implementation is built on the POX controller for several reasons. While POX's documentation is minimal, the fact that it's written purely in Python and provides an accessible API makes it incredibly well-suited for rapid prototyping. POX's modular design also makes it well-suited to SD-MCAN's component-based architecture.

One goal of this thesis is to deploy an SD-MCAN prototype on a physical testbed on Cal Poly's campus; thus, the selected OpenFlow controller must be supported by the available switches. As the only networking lab on campus, Cal Poly's Cisco Networking Lab provides the physical testbed on which this work evaluates SD-MCAN. The lab is fitted with Cisco Catalyst 3850 series switches; these switches support OpenFlow versions 1.0 and 1.3. The documentation explicitly lists POX as a supported controller; thus this prototype is developed using POX.

POX does have its limitations. The latest official release of POX supports only OpenFlow 1.0; this limits its utility as several newer features of OpenFlow are not supported. Notably for SD-MCAN, OpenFlow 1.0 does not support MPLS tagging. This prototype implements label-switching using VLAN tagging instead. While this disables VLAN support in the presented test networks, it also provides an appropriate estimation of the performance of MPLS tagging in a production SD-MCAN network. As both MPLS and VLAN sit near layer 2, OpenFlow switches process their flow rules at analogous speeds. Additionally, the presented prototype does not feature other newer OpenFLow features like the TTL decrement and Ethernet checksum rewrite actions.

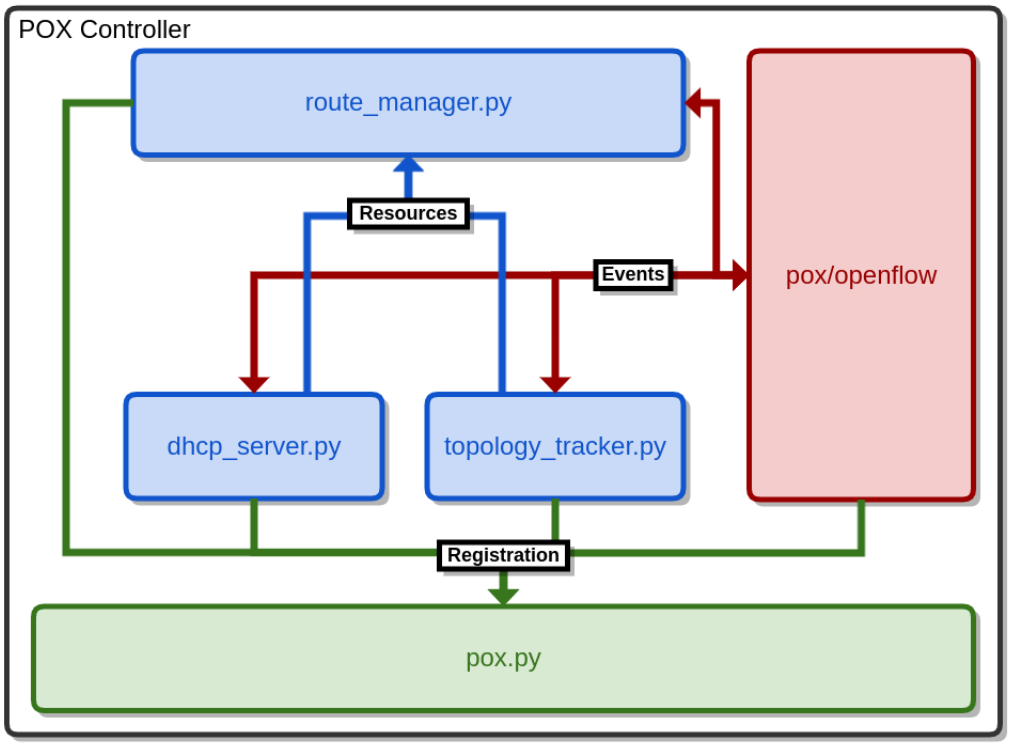

Figure 5.1: POX Controller with SD-MCAN

With this drawback, it may appear as though the POX controller is not well-suited to an SD-MCAN prototype; however, the OpenFlow restrictions on the available

networking hardware prevent a fully-functioning SD-MCAN from being implemented on hardware regardless of controller selection. While the Catalyst 3850 switches support OpenFlow 1.3, they only support a subset of OpenFlow 1.3 functionality; this subset excludes MPLS tagging. Therefore, selecting an OpenFlow 1.3-compatible controller, such as OpenDaylight, would necessitate the development of a second controller prototype for hardware testing. With two different prototypes, comparing simulated and practical performance would be difficult. Therefore, the presented prototype uses POX as it allows for comparable testing on simulated and hardware testbeds.

Figure 5.1 above depicts the POX implementation of the prototype SD-MCAN controller. The main POX executable, *pox.py* resides at the heart of the controller and contains the POX core object. POX's default OpenFlow library enables OFP on the controller. The SD-MCAN prototype utilizes the *Discovery* module of the POX OpenFlow library to track links on the network; this module uses specially-crafted Link Layer Discovery Protocol (LLDP) packets [32] to discover the topology.

The three components of SD-MCAN are implemented as Python applications, written using the POX API, which run on top of the POX core object. All three components register with the core object, as does the OpenFlow library; registering allows the components to communicate with each other. The SD-MCAN components communicate with the OpenFlow library to send and receive OpenFlow messages, enabling communication with OpenFlow switches as well as the retrieval of topology information via OFP. The *dhcp server* and *topology tracker* components exchange information via POX events, and the *route manager* component accesses resources from the other SD-MCAN components through the POX core. For this implementation, the route manager installs the CR flow entries using only hop-count as a routing metric for the shortest path algorithm. This is simple and should be improved in production-ready implementations.

The code for this SD-MCAN prototype is available on GitHub. The current version supports only IPv4 networking and a single interface per host, but the controller could easily be extended to support IPv6 and multiple interfaces. Thus, the prototype is only evaluated on IPv4 traffic in this work. ER flow entry idle timeouts are set at 15 seconds; this value is chosen because it allows the controller to limit flow table sizes. Additionally, host idle timeout is set at 30 seconds. These values are low to allow aggressive flow and host table management. The values could be increased to decrease the amount of controller messages needed at the cost of accuracy.

5.2 Simulation

The SD-MCAN prototype is first evaluated on a virtual network. The network is run as a Mininet instance on a single laptop, the specifications of which can be seen in Table 5.1 below. Mininet utilizes Open vSwitches as switches on the network, while processes with unique networking namespaces simulate hosts on the network. Table 5.2 shows which versions of these tools are used.

Table 5.1: Laptop System Specification

Operating System	CPU	Clock	Memory
Ubuntu 16.04 LTS	Intel Core i7-3520M	2.90 GHz	12 GB DDR3 RAM

Table 5.2: Network Virtualization Software

Network	Switch	Controller
Mininet 2.2.1	Open vSwitch 2.5.2	POX 0.5.0

5.3 Hardware Testbed

SD-MCAN's architecture enables rapid deployment on vendor switches with need for only minimum switch firmware updates. To evaluate the SD-MCAN prototype on vendor switches, a testbed of Cisco Catalyst 3850 series switches, each running iOS 3.07, is set up in Cal Poly's networks lab. 1 Gbps copper Ethernet links connect the switches. To enable OpenFlow support, the Cisco Plug-in for OpenFlow version 2.0.7 is installed on the switches via an OVA package. This plug-in runs in a virtual service container on the switch OS.

After installing the package, the virtual service is installed and enabled. OpenFlow is configured on each switch to use OpenFlow Protocol version 1.0 and connect to a remote controller using IPv4; each switch connects to the controller out-of-band via a dedicated interface. Additionally, specific ports on each switch are designated OpenFlow ports, allowing OpenFlow traffic to be isolated. On edge switches, the default flow table miss behavior is set to forward packets to the controller, while core switches are configured to drop unmatched packets. This prevents unnecessary packet-in messages to the controller during handoffs.

With the plug-in enabled and configured, the switches support all OpenFlow 1.0 functionality; however, the switches have several notable restrictions. The Catalyst 3850 supports only 1000 L2 flows and 500 L3 flows. Additionally, the maximum sustained rate of flow programming can not exceed 40 flows per second and the rate of packet-in messages sent to the controller cannot exceed 300 packets per second.

Unfortunately, through testing, the plug-in is found not to support certain action combinations that the SD-MCAN prototype requires. VLAN tag rewriting is found to not work at all. As a work around, the IP Type-of-Service (ToS) byte of the IP header is used to store the desired label. For the IPv4 datagrams the prototype will be

routing, the IP ToS value can be assumed to be zero when entering the network. Thus, this value is used as the "no label" value. The available ToS values are extremely limited due to the format of the ToS bits. The first six bits of ToS define the DiffServ Code Points (DSCP), while the least significant two bits are reserved for Explicit Congestion Notification (ECN). OpenFlow only uses the DSCP bits of ToS, so the only valid non-zero values are shown in Table 5.3 below. This label pool is extremely limited compared to the 4 byte MPLS label field in the MPLS header.

Table 5.3: Valid Non-Zero ToS Values

32	40	48	56	64	72	80
88	96	104	112	120	128	136
144	152	160	184	192	224	

However, the switches are then found to be unable to process the rewriting of both Ethernet addresses and IP ToS bits. Cisco was contacted and confirmed the limitation and that the plugin is no longer in development. Cisco will replace the plug-in with a new package to be released in the future. This discovery proves disappointing as it leaves SD-MCAN unable to be evaluated in full on the testbed; however, a work-around is found to allow basic testing on the testbed. As a hack, Ethernet destination address rewriting is removed from flow rules. Thus, limited CR and ER flow entries are employed on the Cisco switches, seen in Figures 5.2, 5.3, and 5.4 below.

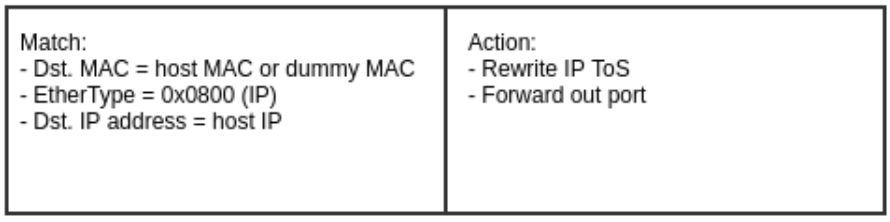

Figure 5.2: Limited ER Push Flow Entry

Match:	Action:
- EtherType = 0x0800 (IP) - IP ToS = label	- IP ToS = 0x0 - Forward out port

Figure 5.3: Limited ER Pop Flow Entry

Match:	Action:
- EtherType = 0x0800 (IP) - IP ToS = ingress label	- Update IP ToS to outgress label - Forward out port

Figure 5.4: Limited CR Flow Entry

This hack allows only testing of UDP traffic on the physical testbed, whereas the simulated testbed allows for both TCP and UDP. This is because the lack of Ethernet destination MAC change causes the receiving host to drop the packets. While the physical testbed is limited it still allows observations to be made about SD-MCAN's potential on vendor hardware.

Chapter 6

VALIDATION

With the theoretical benefits of SD-MCAN established, this chapter examines the practical performance of the implemented SD-MCAN controller prototype. Performance is evaluated through four sets of experiments; the first three experiments utilize a Mininet virtual network with Open vSwitches, and the final experiment is conducted on the Cisco hardware testbed. Experiment set A confirms the completeness of the proposed prototype. Experiment set B focuses on evaluating the handoff performance of the controller, and experiment set C analyzes switch and controller load at scale. Finally, experiment set D shows how the system performs on vendor hardware.

6.1 Experiment Set A - Completeness

Before testing the performance of the SD-MCAN prototype, its operation must be verified. This experiment set contains two experiments to ensure that the SD-MCAN prototype behaves as expected in regard to DHCP functionality and flow table management. For this experiment set, a simple three-LAN topology is constructed on Mininet, shown in Figure 6.1. The topology features 6 switches, 3 core and 3 edge, with all switches connected by 1 Gbps links with 1 ms delay. LAN1 hosts subnet 192.168.0.0/24 and attaches to the network via S4. LAN2 and LAN3 host subnets 192.168.1.0/24 and 192.168.2.0/24 at S5 and S6, respectively. The POX controller, running the SD-MCAN application, resides on the loopback interface on port 6633.

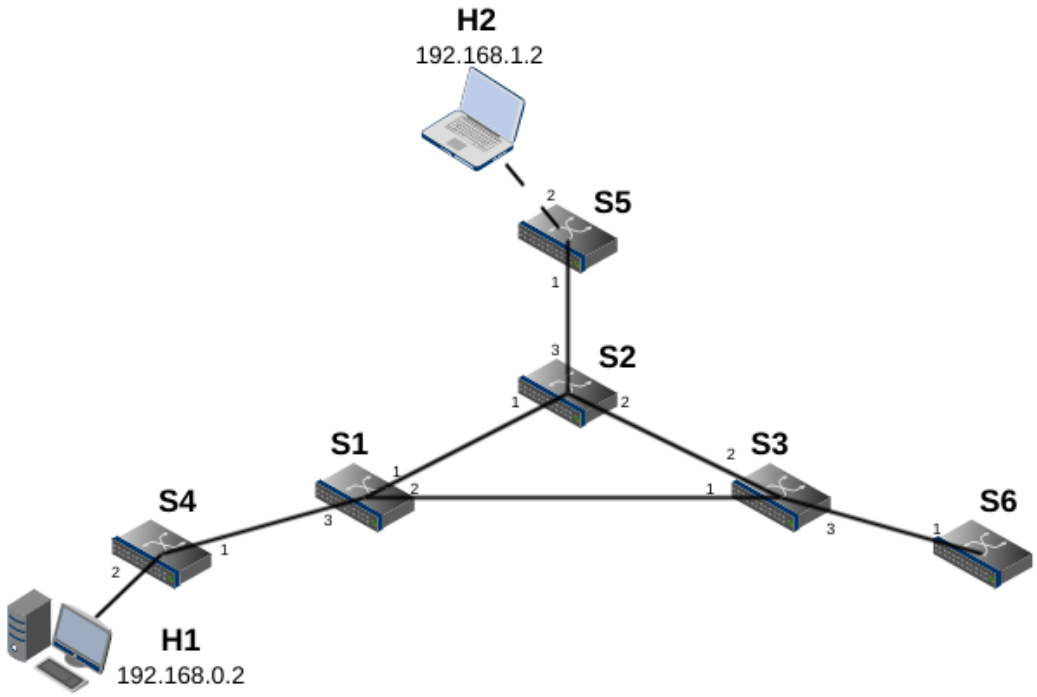

Figure 6.1: Experiment Set A Mininet Topology

For the first experiment, 253 hosts are connected on each LAN; this is the maximum number of hosts supported on each subnet due to the network address, broadcast address, and fakeway address. IPv6 and multicast are disabled on the hosts. After connecting, each host runs *dhclient* to lease an IPv4 address from the controller, then the *ifconfig* command is used to verify the proper IP configuration. After connecting all hosts, a 254th host attempts to connect to each LAN, and the controller confirms that the leases are declined as no address space remains. Finally, all hosts are disconnected from the network, and the controller confirms that the expired leases are added back in to the controller's address pool.

The next experiment features 2 hosts, H1 and H2. H1 connects to S4, and H2 connects to H2 to S5; both connect via a 100 Mbps link with 1 ms delay. After connecting the hosts, dhclient and ifconfig are run to obtain and verify an IP address

on each host. Next, H1 sends 2 ping packets to H2, verifying the connection between the two. The connectivity indicates that the controller properly installs all needed flow entries. To verify this, the *ovs-ofctl dump-flows* command is run for each switch in the network. *ofctl* is a command line tool for monitoring and administering OpenFlow switches; the *dump-flows* command prints out the contents of a given switch's flow table. Figure 6.2 shows the printouts for the edge switches S4, S5, and S6; the complete flow table printouts from this experiment can be found in Appendix A.

```
S4 NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=17.352s, table=0, n_packets=3, n_bytes=123, idle_age=4, priority=65000,dl_dst=01:23:20:00:00:01,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x0, duration=17.312s, table=0, n_packets=0, n_bytes=0, idle_age=17, priority=32769,arp,dl_dst=02:00:00:00:be:ef actions=CONTROLLER:65535
 cookie=0x0, duration=17.312s, table=0, n_packets=2, n_bytes=684, idle_age=2, udp,tp_src=68,tp_dst=67 actions=CONTROLLER:65535
 cookie=0x0, duration=2.042s, table=0, n_packets=2, n_bytes=196, idle_timeout=10, idle_age=1, ip,dl_dst=03:00:00:00:be:ef,nw_dst=192.168.1.2
actions=mod_dl_dst:4e:c0:c1:7f:07:34,mod_vlan_vid:16,output:1
 cookie=0x0, duration=1.972s, table=0, n_packets=3, n_bytes=306, idle_timeout=10, idle_age=1, ip,dl_vlan=24 actions=strip_vlan,output:2

S5 NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=17.358s, table=0, n_packets=3, n_bytes=123, idle_age=3, priority=65000,dl_dst=01:23:20:00:00:01,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x0, duration=17.328s, table=0, n_packets=0, n_bytes=0, idle_age=17, priority=32769,arp,dl_dst=02:00:00:00:be:ef actions=CONTROLLER:65535
 cookie=0x0, duration=17.328s, table=0, n_packets=2, n_bytes=684, idle_age=2, udp,tp_src=68,tp_dst=67 actions=CONTROLLER:65535
 cookie=0x0, duration=2.046s, table=0, n_packets=3, n_bytes=306, idle_timeout=10, idle_age=1, ip,dl_vlan=18 actions=strip_vlan,output:2
 cookie=0x0, duration=1.977s, table=0, n_packets=2, n_bytes=196, idle_timeout=10, idle_age=1, ip,dl_dst=03:00:00:00:be:ef,nw_dst=192.168.0.2
actions=mod_dl_dst:9e:e8:52:3f:4e:5e,mod_vlan_vid:22,output:1

S6 NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=17.373s, table=0, n_packets=3, n_bytes=123, idle_age=5, priority=65000,dl_dst=01:23:20:00:00:01,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x0, duration=17.334s, table=0, n_packets=0, n_bytes=0, idle_age=17, priority=32769,arp,dl_dst=02:00:00:00:be:ef actions=CONTROLLER:65535
 cookie=0x0, duration=17.334s, table=0, n_packets=0, n_bytes=0, idle_age=17, udp,tp_src=68,tp_dst=67 actions=CONTROLLER:65535
```

Figure 6.2: Edge Switch Flow Tables Static

The printouts verify that each switch has special flow entries for handling ARP, DHCP, and LLDP packets (used by POX's OpenFlow component). Additionally, S4 has a push rule for H1 and a pop rule for H2, while S5 has a push rule for H2 and a pop rule for H1. Since S6 sees no traffic, its flow table contains only the three special entries. This matches the expected controller behavior.

After verifying the flows, H2 is moved from LAN2 on S4 to LAN3 S5, and the process is repeated a second time. Once again, *dhclient* and *ifconfig* are run on H2 to verify that the controller leased H2 its LAN2 IP address. H1 is made to ping H2, and the flow tables are dumped again. Figure 6.3 shows the edge switch flow tables after H2 goes mobile. The flow tables reflect H2's new location, as S6 now contains a push entry for H1 and a pop entry for H2 while S4's push entry for H2 has been updated.

This scenario demonstrates that the SD-MCAN prototype controller handles mobile hosts as defined in the design.

```
S4 NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=19.557s, table=0, n_packets=4, n_bytes=164, idle_age=1, priority=65000,dl_dst=01:23:20:00:00:01,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x0, duration=19.517s, table=0, n_packets=0, n_bytes=0, idle_age=19, priority=32769,arp,dl_dst=02:00:00:00:be:ef actions=CONTROLLER:65535
 cookie=0x0, duration=19.517s, table=0, n_packets=2, n_bytes=684, idle_age=4, udp,tp_src=68,tp_dst=67 actions=CONTROLLER:65535
 cookie=0x0, duration=2.032s, table=0, n_packets=4, n_bytes=408, idle_timeout=10, idle_age=0, ip,dl_vlan=24 actions=strip_vlan,output:2
 cookie=0x0, duration=2.040s, table=0, n_packets=4, n_bytes=392, idle_timeout=10, idle_age=0, ip,dl_dst=03:00:00:00:be:ef,nw_dst=192.168.1.2
actions=mod_dl_dst:4e:c0:c1:7f:07:34,mod_vlan_vid:19,output:1

S5 NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=19.563s, table=0, n_packets=3, n_bytes=123, idle_age=5, priority=65000,dl_dst=01:23:20:00:00:01,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x0, duration=19.533s, table=0, n_packets=0, n_bytes=0, idle_age=19, priority=32769,arp,dl_dst=02:00:00:00:be:ef actions=CONTROLLER:65535
 cookie=0x0, duration=19.533s, table=0, n_packets=2, n_bytes=684, idle_age=4, udp,tp_src=68,tp_dst=67 actions=CONTROLLER:65535
 cookie=0x0, duration=4.251s, table=0, n_packets=4, n_bytes=408, idle_timeout=10, idle_age=2, ip,dl_vlan=18 actions=strip_vlan,output:2
 cookie=0x0, duration=4.182s, table=0, n_packets=2, n_bytes=196, idle_timeout=10, idle_age=3, ip,dl_dst=03:00:00:00:be:ef,nw_dst=192.168.0.2
actions=mod_dl_dst:9e:e8:52:3f:4e:5e,mod_vlan_vid:22,output:1

S6 NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=19.579s, table=0, n_packets=4, n_bytes=164, idle_age=2, priority=65000,dl_dst=01:23:20:00:00:01,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x0, duration=19.540s, table=0, n_packets=0, n_bytes=0, idle_age=19, priority=32769,arp,dl_dst=02:00:00:00:be:ef actions=CONTROLLER:65535
 cookie=0x0, duration=19.540s, table=0, n_packets=1, n_bytes=342, idle_age=2, udp,tp_src=68,tp_dst=67 actions=CONTROLLER:65535
 cookie=0x0, duration=2.051s, table=0, n_packets=3, n_bytes=306, idle_timeout=10, idle_age=1, ip,dl_vlan=21 actions=strip_vlan,output:12
 cookie=0x0, duration=2.043s, table=0, n_packets=3, n_bytes=294, idle_timeout=10, idle_age=0, ip,dl_dst=03:00:00:00:be:ef,nw_dst=192.168.0.2
actions=mod_dl_dst:9e:e8:52:3f:4e:5e,mod_vlan_vid:27,output:1
```

Figure 6.3: Edge Switch Flow Tables Mobile

6.2 Experiment Set B - Handoff Performance

This experiment set evaluates the handoff performance of the SD-MCAN controller prototype on a virtual network through two different experiments. The methodology is significantly inspired by research conducted by Wang and Bi [54]. To begin, a multi-LAN topology is constructed in Mininet based on the real CAN at Cal Poly. The network consists of a core mesh of 5 switches. Each core switch connects to a single edge switch, and each edge switch is connected to a single LAN. No virtual switches comprise each LAN; instead, hosts are connected directly to the edge switches to simulate a LAN connection. In total, the topology features 10 switches and 14 links; each link is assigned 1 Gbps bandwidth and 1 ms delay to simulate real network conditions. Figure 6.4 below shows the topology. As Mininet does not support in-band messaging between controller and switch, all communication between the two occurs out-of-band through the loopback interface on port 6633.

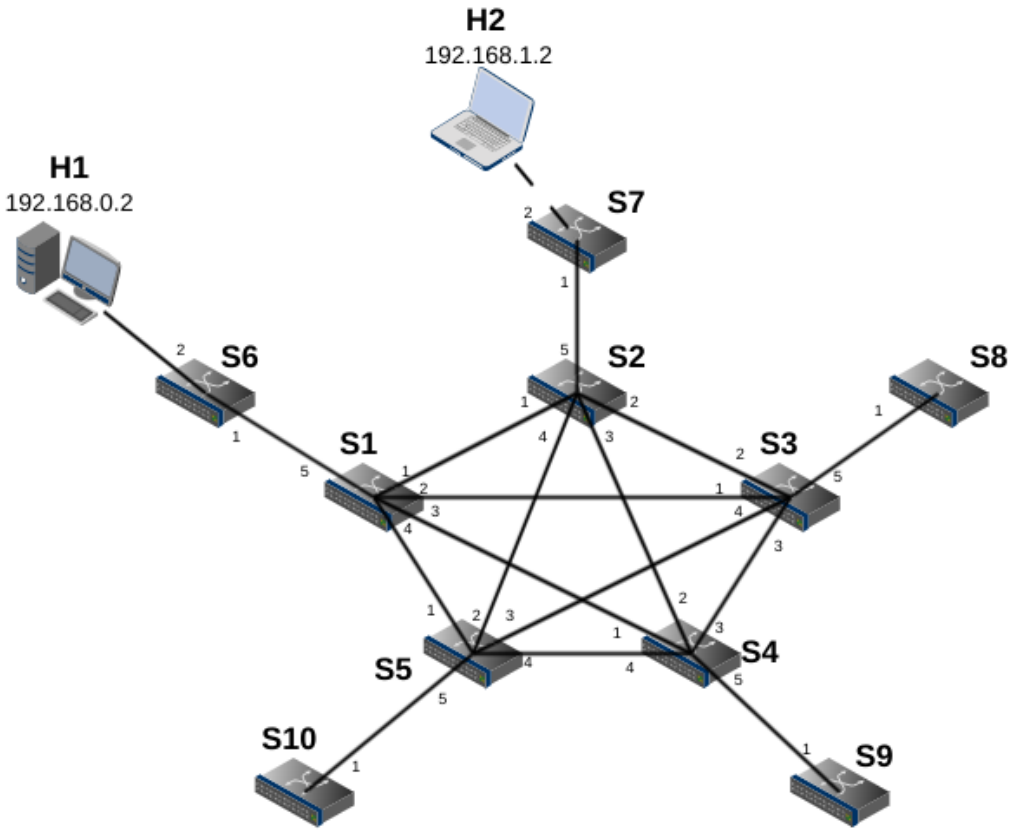

Figure 6.4: Experiment Set B Mininet Topology

In these experiments, 2 hosts are connected to the network. Once again, IPv6 and multicast are disabled on the hosts. H1 remains static, always attached to LAN1 via S6, while H2 moves around the topology acting as a mobile host. Each host is attached to an edge switch with a 100 Mbps bandwidth, 1 ms delay link. This limits each host's bandwidth to prevent the network from exceeding the total available bandwidth on the underlying laptop hardware.

The experiments employ *Iperf*, a widely-used network testing tool, to generate both TCP and UDP traffic between the hosts. The stationary host H1 acts as the Iperf server, while the mobile host H2 acts as the client. During each experiment, Iperf monitors and records TCP and UDP metrics, revealing how H2's movement impacts

the network's performance. Additionally, ubiquitous packet-sniffing tool *Wireshark* is used to analyze metrics that Iperf does not detail.

In the first experiment, Iperf is run with both TCP and UDP between H1 and H2 for 60 seconds. During this time, H1 remains stationary at S6. Meanwhile, H2 walks clockwise around the LANs on the path [S7, S8, S9, S10, S6, S7, S8, S9, S10, S6, S7, S8], making one handoff every five seconds. At each new point of connection, *dhclient* is run on H2 to force it to ask for a new DHCP lease. Figures 6.5, 6.6, and 6.7 show plots of the TCP throughput, UDP packet loss and jitter of the 11 handoffs, collected by Iperf during each 1 second interval.

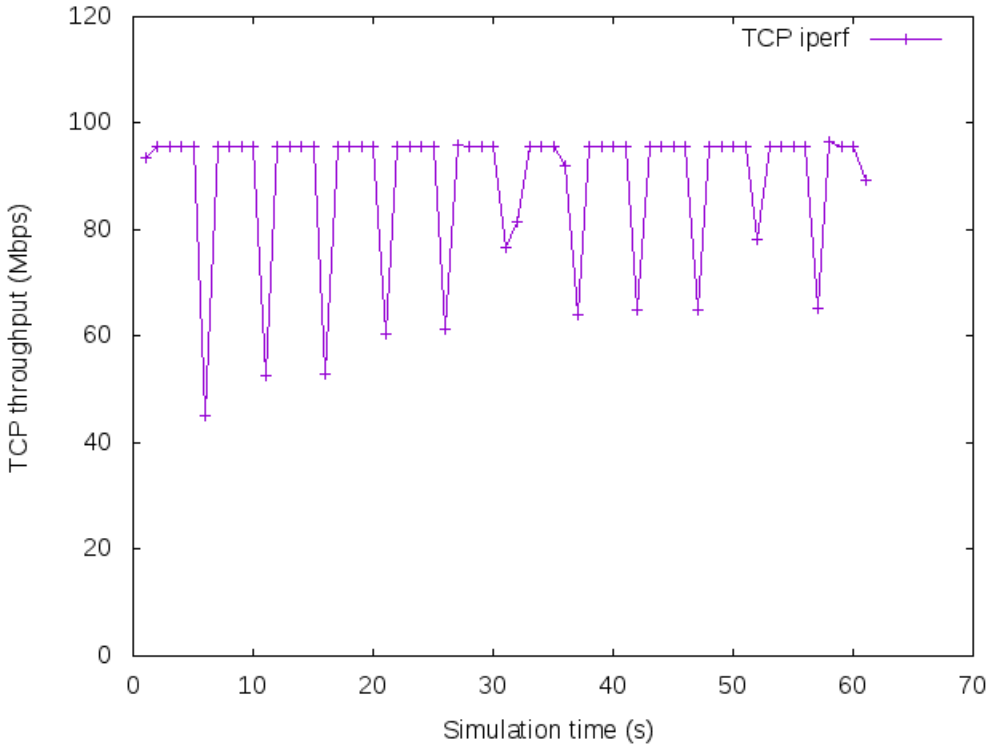

Figure 6.5: TCP Throughput

Throughput is the amount of data transmitted over a give period of time. The throughput graph shows that each handoff degrades end-to-end performance and that the degradation varies in severity from between 20% and 50%. While the degradation

is significant, it is also transient; the TCP transmission recovers within one second.

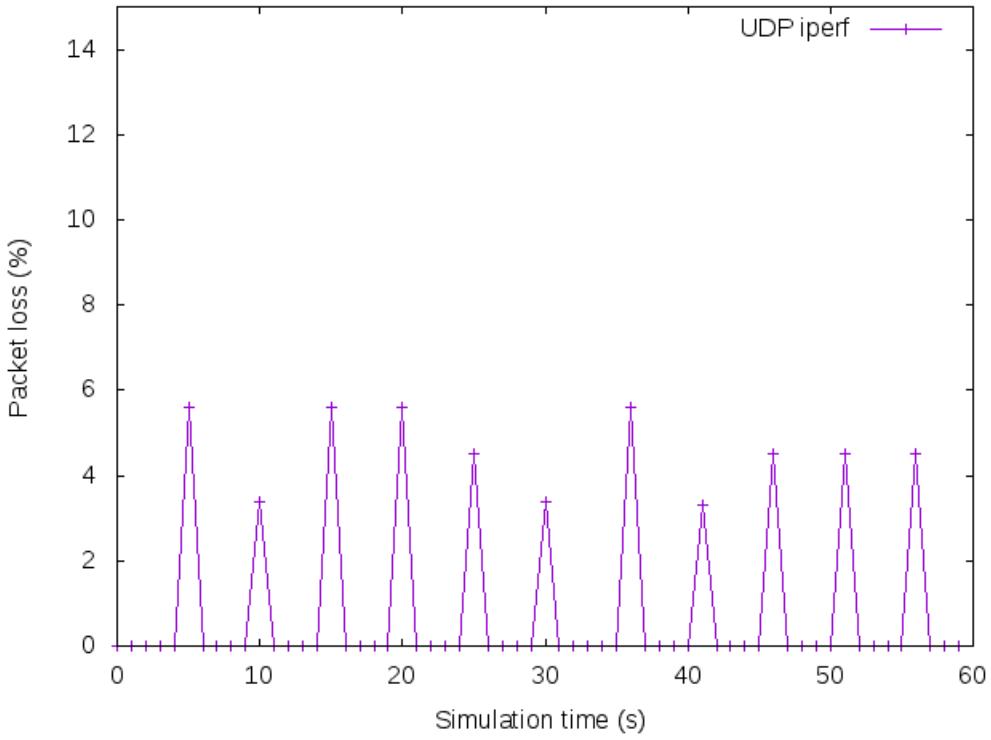

Figure 6.6: UDP Packet Loss

Looking at UDP, a connectionless transmission protocol, allows each handoff's impact on packet loss to be seen. The plot indicates that each handoff results in a 3-6% packet loss. As with the TCP throughput drop, this packet loss is transient and recovers quickly. The packet loss is due to the brief interval in which the edge switch flow tables have not been updated. Iperf also reveals the jitter, the variation in delay of received packets, between the hosts. As this metric only considers received packets, the impact of each handoff on the UDP jitter is slight (always under 1 ms).

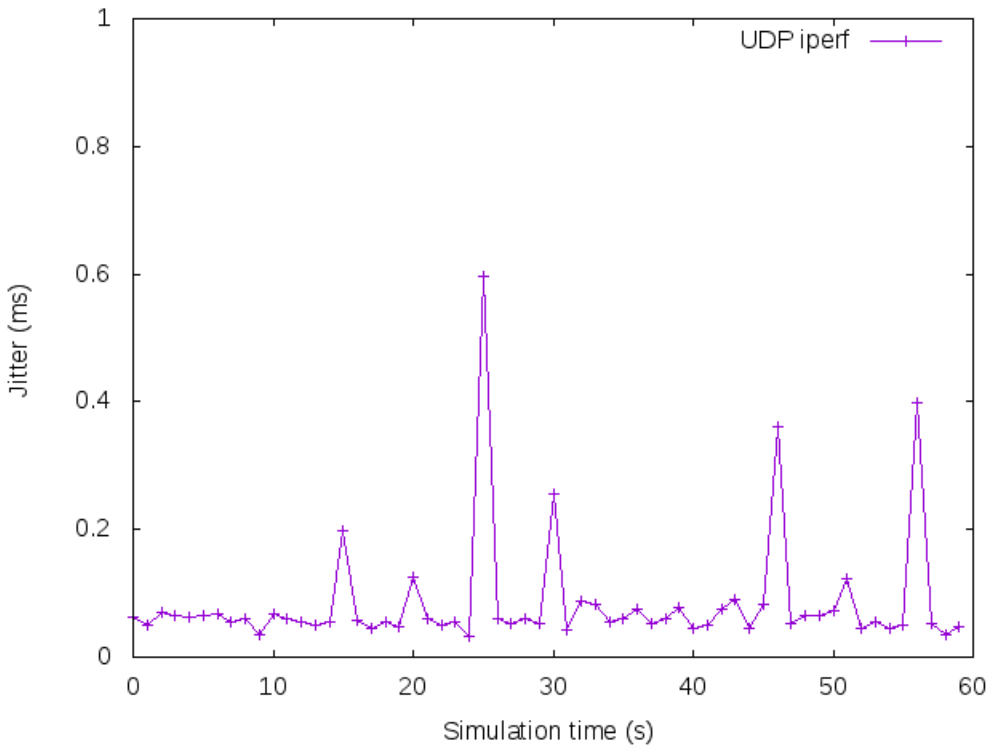

Figure 6.7: UDP Jitter

During the Iperf walk experiment, Wireshark sniffs packets on port 1 of S6. Analyzing the resulting packet trace allows for a closer look at TCP behavior during the 11 handoffs. Figure 6.8 shows the round-trip-times (RTTs) for the TCP stream, while Figure 6.9 shows the progression of sequence numbers. The RTT graph reveals a significant spike in RTT during each handoff, although the impact varies between handoffs from just over 45 ms to almost 270 ms. The sequence plot shows that the TCP stream remains relatively smooth despite the handoffs.

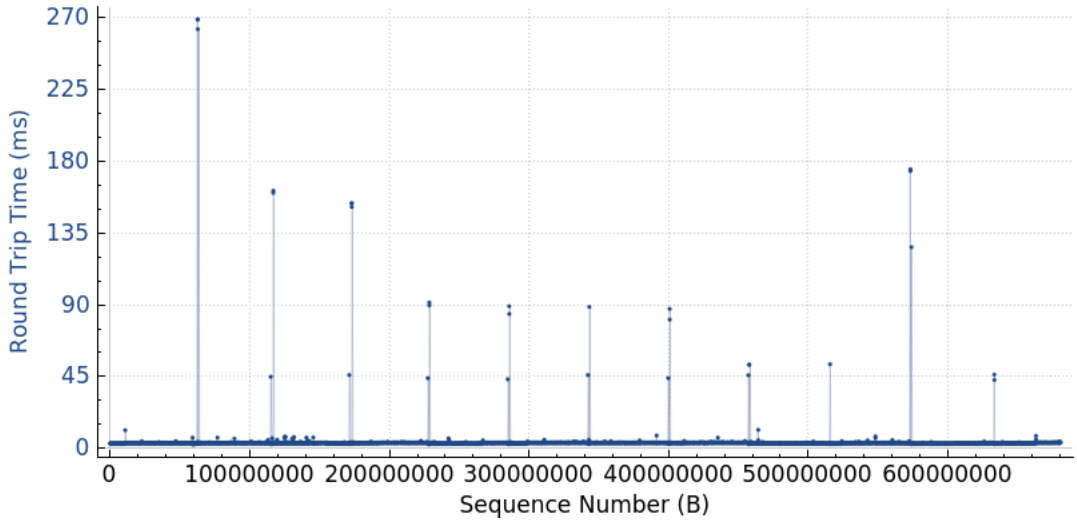

Figure 6.8: Walk TCP RTT

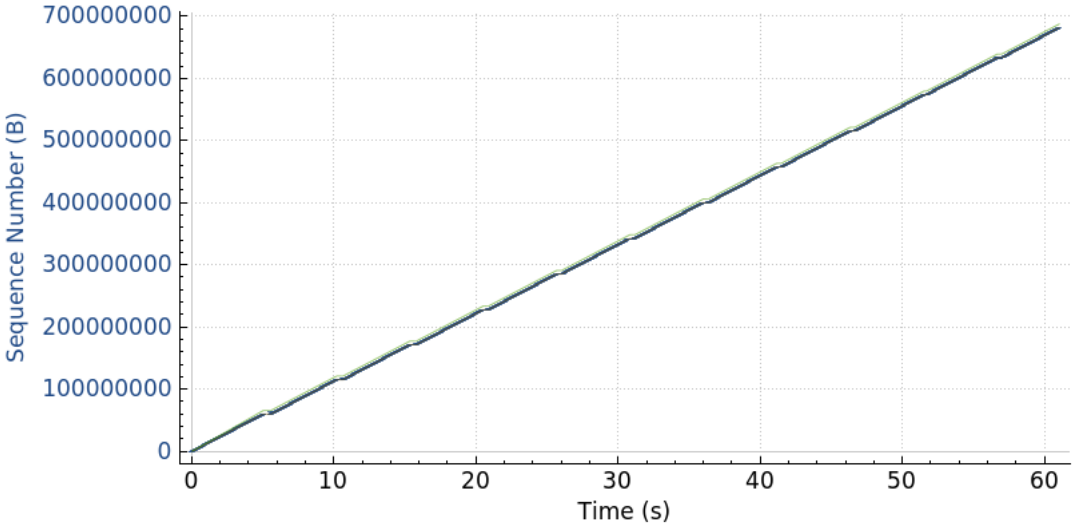

Figure 6.9: Walk TCP Sequence Numbers

To understand the performance degredation during handoffs, specific handoffs are examined more closely. Handoffs 1 and 11 are chosen, as they experience the most severe and least severe impact on RTT, respectively. Figure 6.10 depicts the RTT of TCP packets during handoff 1. Packets are lost during the handoff when the edge switch flow tables have not yet been updated. After the updates, there is a brief

period of high RTTs due to data retransmission.

The TCP sequence plot for handoff 1, shown in Figure 6.11, reveals further information. Notably, the plot indicates a period of 460 ms between the connection interruption and connection resumption. Near the end of the 460 ms interruption, a packet-in arrives to resume the connection at H2's new location. This can be seen on the plot by the two isolated packets. The following spike in sequence numbers shows duplicate ACKs before fast retransmission allows the TCP stream to recover.

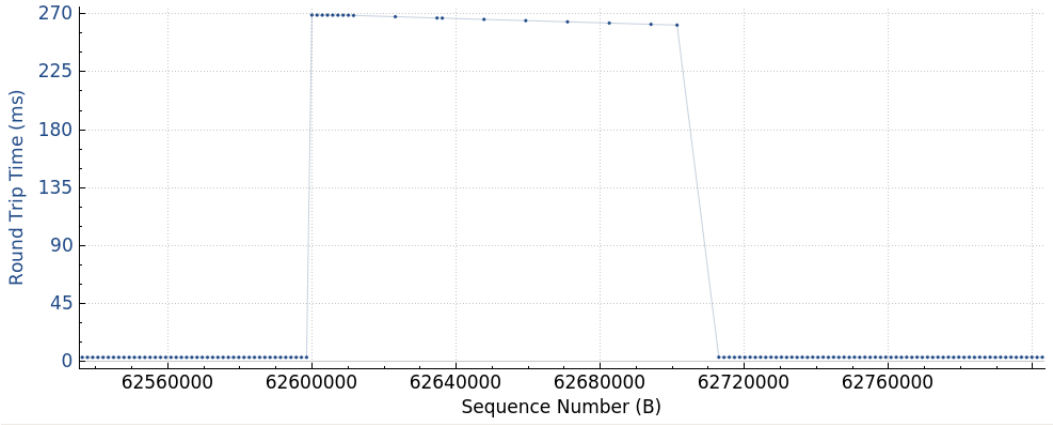

Figure 6.10: Handoff 1 TCP RTT

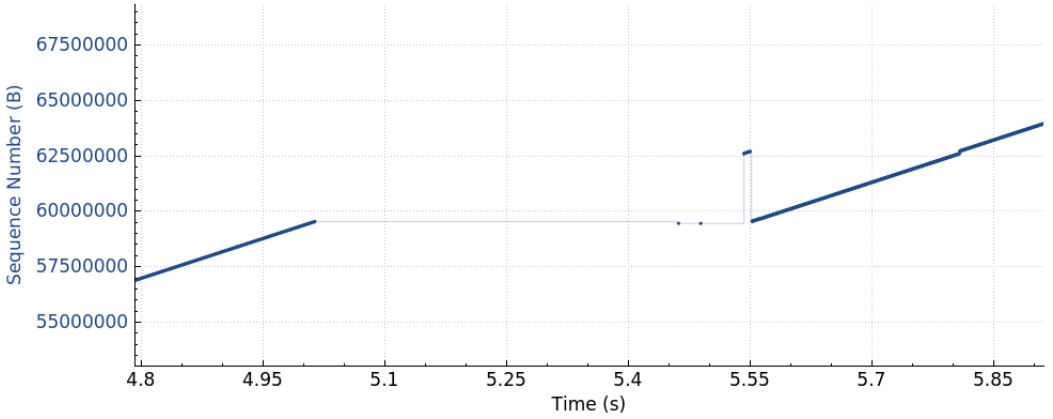

Figure 6.11: Handoff 1 TCP Sequence Numbers

Figures 6.12 and 6.13 depict the RTT and sequence numbers for handoff 11. These

plots match the plots for handoff 1, with the only difference being the duration of the delay. This delay ranges from 250 to 460 ms, durations which may be unacceptable for high-data applications like video streaming. Interestingly, the sequence number plots suggest that the majority of the disruption time may not be due to the SD-MCAN functionality but rather the delay in moving H2 on Mininet and deficiencies in POX's implementation of OpenFlow.

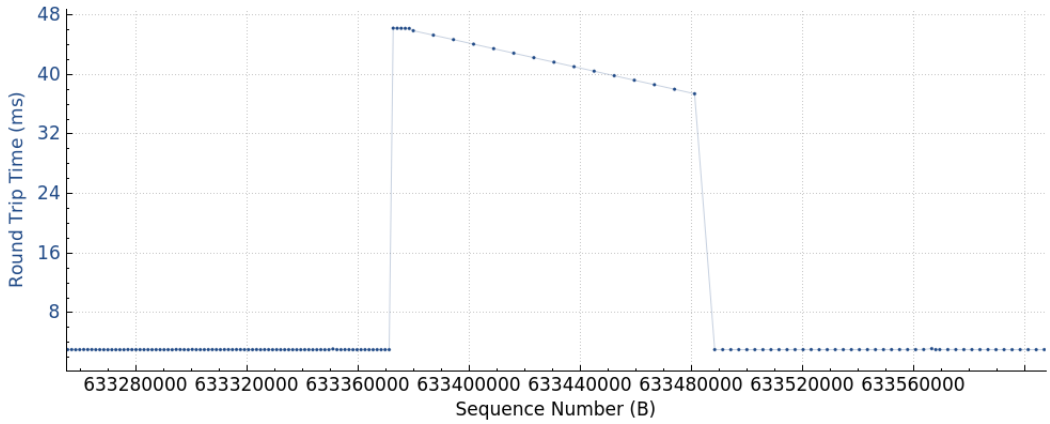

Figure 6.12: Handoff 11 TCP RTT

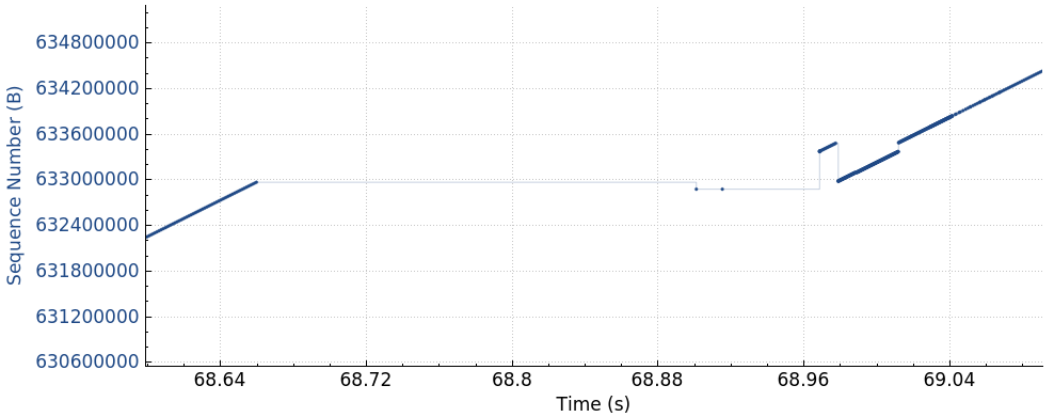

Figure 6.13: Handoff 11 TCP Sequence Numbers

To analyze the actual TCP stream disruption due only to SD-MCAN, Wireshark also sniffs on the loopback interface. This allows controller traffic to be investigated.

The controller packets are divided into two distinct periods, designated A and B. Handoff period A is calculated by subtracting the time when the first OpenFlow port status message is transmitted from the time when the controller recognizes H2's new location; this is the period during which Mininet moves the virtual host and POX's OpenFlow implementation discovers the topology change. Handoff period B is calculated by subtracting the time at which the controller receives the first packet-in from the moved H2 from the time when the last flow table modification is sent out. This period represents the time take for SD-MCAN to process the handoff and update all relevant flow tables.

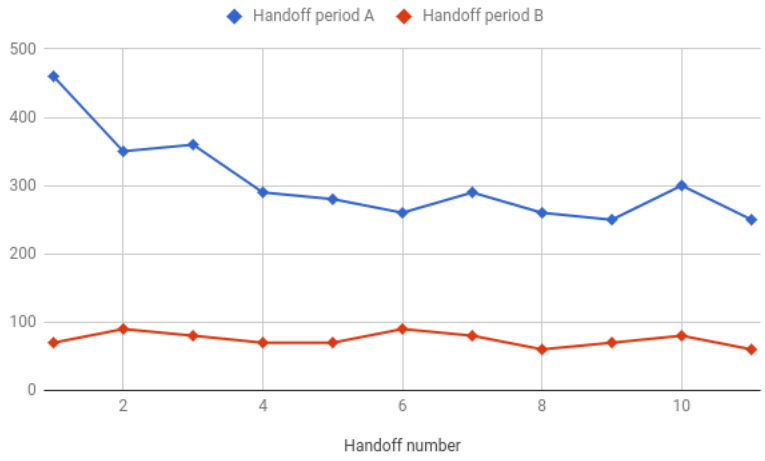

Figure 6.14: Handoff Time

Figure 6.14 shows the time taken by both periods during each of the 11 handoffs. While the total TCP interruption time ranges from 250 to 460 ms, the time taken by SD-MCAN to process the handoffs ranges only from 60 to 90 ms. Figure 6.15 and Table 6.1 provide a better picture of how much of the total handoff time is occupied by handoff periods A and B. The data shows that handoff period A accounts for around 74-87% of the handoff time. The result reveals that the majority of the TCP disruption is not due to SD-MCAN, but rather to Mininet and POX. This suggests that a production-quality version of SD-MCAN, using a Java or C-based controller,

would have handoff latencies appropriate even for demanding applications.

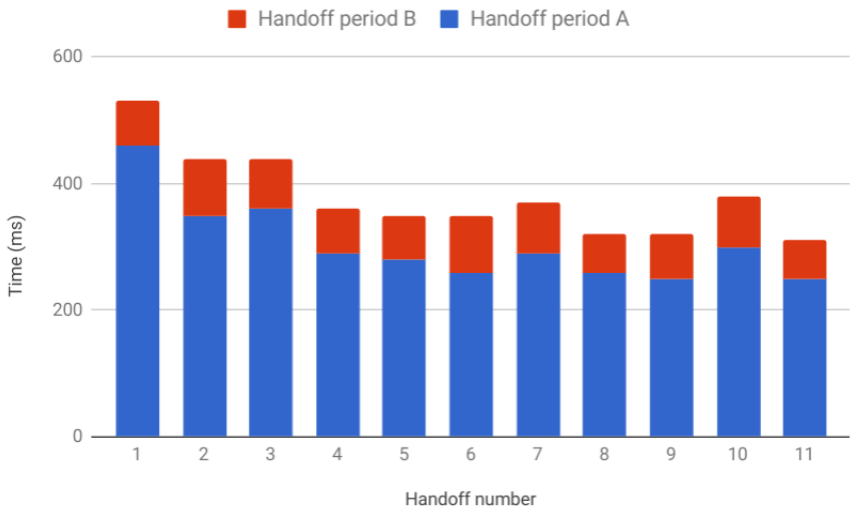

Figure 6.15: Handoff Time

Table 6.1: Percentage of Handoff Spent in Each Handoff Period

Handoff	% Period A	% Period B
1	86.79	13.21
2	79.55	20.45
3	81.82	18.18
4	80.56	19.44
5	80.00	20.00
6	74.29	25.71
7	78.38	21.62
8	81.25	18.75
9	78.13	21.88
10	78.95	21.05
11	80.65	19.35

In the previous experiment, the mobility interval is fixed at 5 seconds. While this provides a clear look at the performance impact of each handoff, it gives no information about how the frequency of handoffs impacts the overall performance of an end-to-end connection on the network. In the next experiment, H2 is walked around the LANs for 60 seconds again, but the mobility interval is varied from 5 seconds down to 1 second.

Figures 6.16 and 6.17 show the average packet loss and average TCP throughput of the H1-H2 connection at different mobility intervals. As one might predict, throughput increases with the mobility interval while packet loss drops. Notably, these plots show that significant performance degradation only occurs at short intervals which are unlikely to occur in a practical setting. Regardless of likelihood, the data proves that SD-MCAN supports handoffs over short intervals at a performance cost.

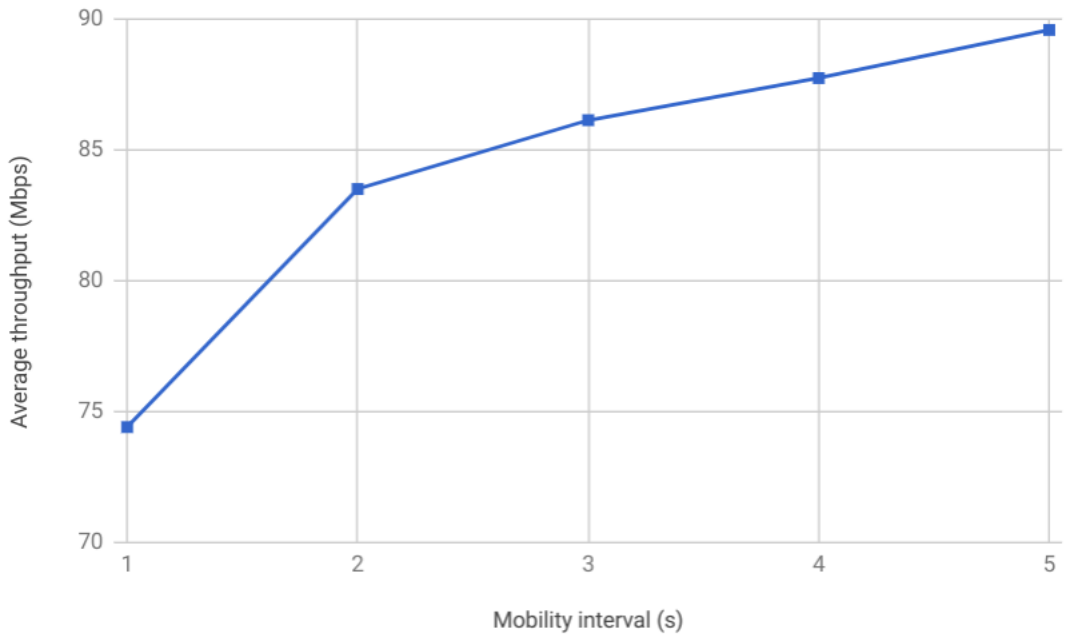

Figure 6.16: Average TCP Throughput vs. Mobility Interval

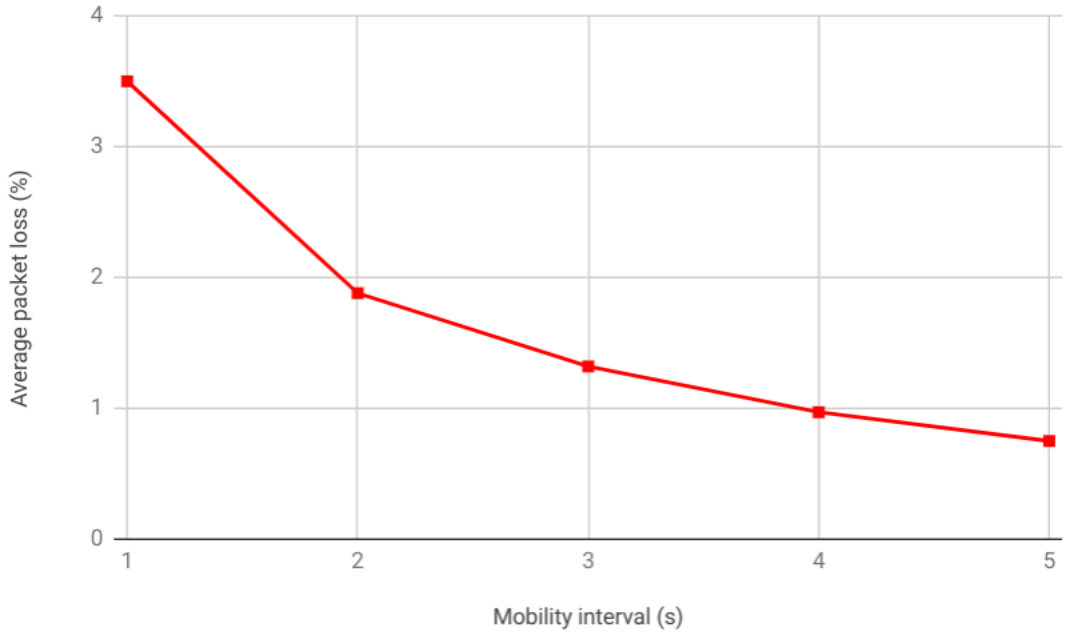

Figure 6.17: Average UDP Packet Loss vs. Mobility Interval

6.3 Experiment Set C - Scalability

This experiment set evaluates the prototype SD-MCAN implementation at a significantly larger scale. The first experiment analyzes the traffic load on the controller in a larger network of mobile hosts, while the second examines the impact of the system on the size of flow tables in the network. The same topology from experiment set B is employed here with one significant change. The original 10 switches remain controlled by a POX controller running the SD-MCAN application on port 6633, but an additional 5 switches are added on each edge to form each LAN; these new switches are connected to a second POX controller on port 6634. The second POX controller runs a built-in application that simulates the behavior of a legacy L2 switch on each switch. Figure 6.18 shows this topology, which now contains 35 switches and 39 links.

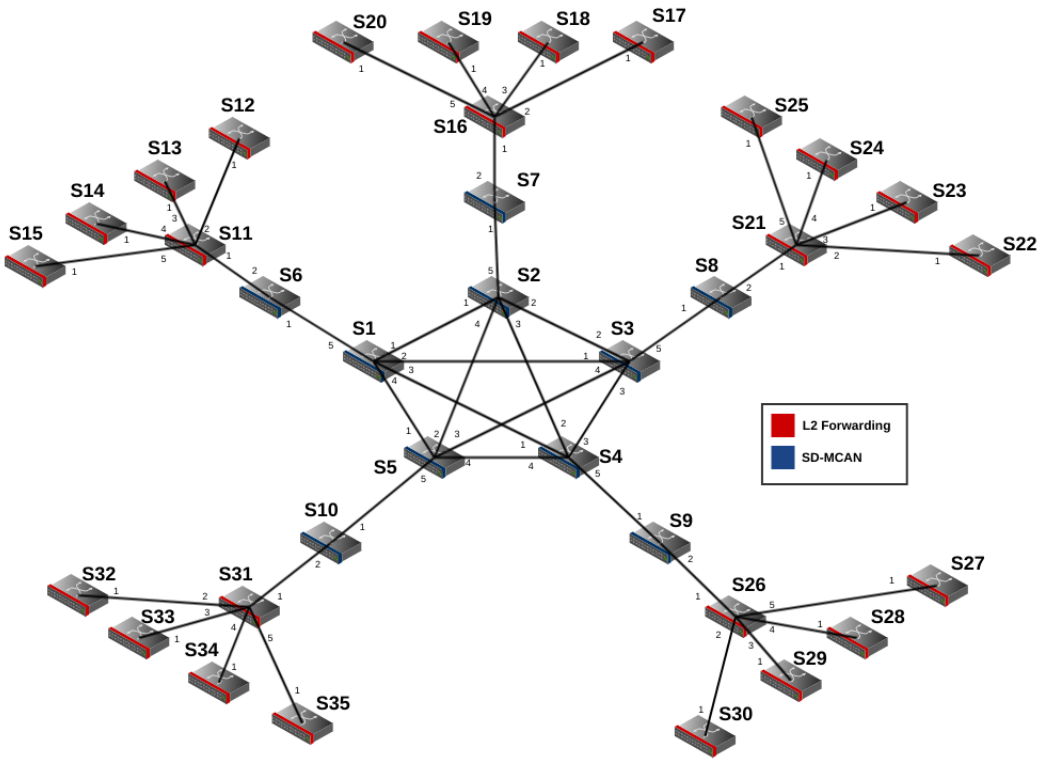

Figure 6.18: Experiment Set C Mininet Topology

In the first experiment, 100 hosts are connected to the network via a 1 Mbps/1 ms delay link, each host on a randomly selected LAN. Low bandwidth links are used to prevent the simulation from exceeding the maximum available throughput on the underlying hardware. After connecting, each host runs *dhclient* to lease an IP address before sending ICMP packets to an available host (if one exists) using the *ping* utility. After all 100 hosts connect to the network, each host is moved to a randomly selected new location on the network. As Mininet does not handle multi-threading well, each host is moved one at a time. As with experiment set B, the hosts are moved according to a specified mobility interval. As the hosts become mobile, Wireshark sniffs the loopback interface on port 6633 (the location of the SD-MCAN controller). The Wireshark capture provides information about both the input and output load on the controller. Figure 6.19 plots the controller input load (in packet-

ins per second) for each given mobility interval, while Figure 6.20 plots the output load (in flow modifications per second).

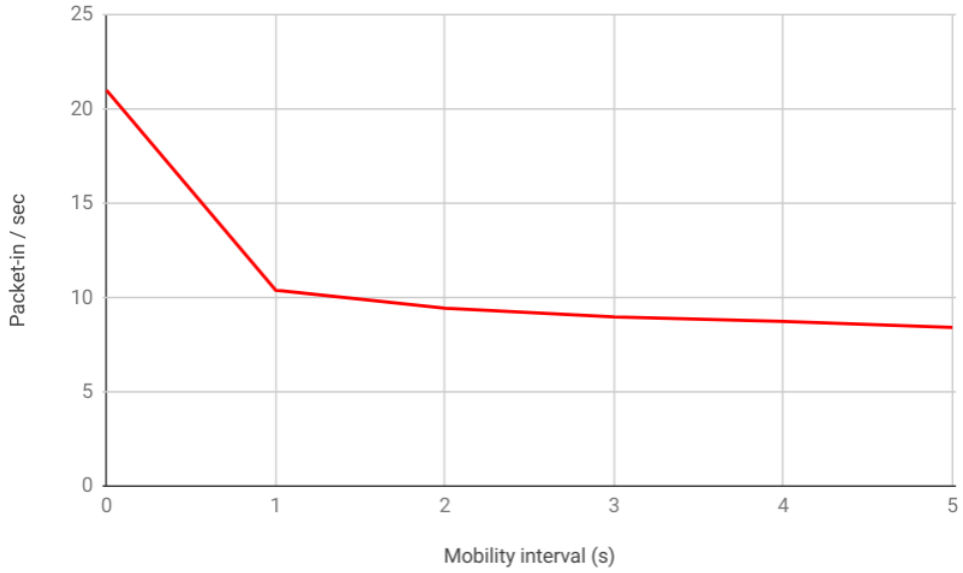

Figure 6.19: Packet-in Rate

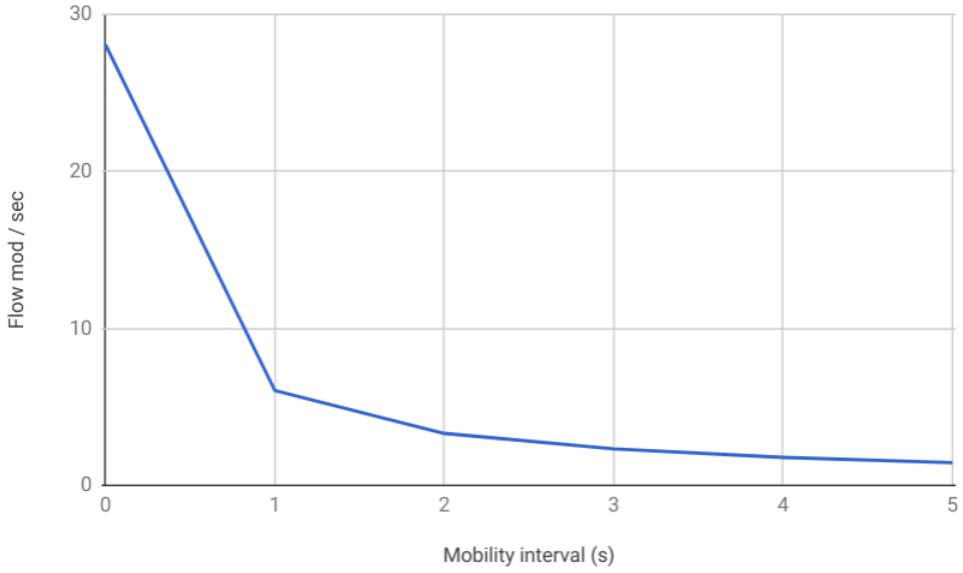

Figure 6.20: Flow Mod Rate

The first data point allows the hosts to move as fast as Mininet will allow; from

the previous experiment set this value is known to be around 10 ms. After that, hosts are added at intervals of 1 to 5 seconds. Both plots show that the mobility interval contributes little to the controller load if the interval exceeds 1 second. This trend suggests that the mobility handoffs only cause heavy load on the controller when many hosts move in a short time period (less than 1 second); otherwise, the handoffs are unlikely to cause congestion on the controller.

In Figure 6.19, the packet-in rate changes only slightly as the mobility interval increases; this indicates that the dominant source of controller packet-ins is not mobility handoffs but something else. To investigate this, the packet-ins from each run are analyzed in the Wireshark captures. As expected, packet-ins come from 4 sources: Link-layer Discovery Protocol (LLDP) packets used by POX's OpenFlow module to track the topology, DHCP packets sent to the controller when each hosts connects to a new point in the network, ARP packets used when hosts ARP hosts on other LANs or by the controller to check host liveliness, and flow table misses during handoffs.

Figure 6.21 shows the packet-ins per second at each mobility interval broken down by type. The graph shows that as the mobility interval gets longer (over 1 second), the volume of LLDP and ARP packet-ins dominates the total load on the controller. This indicates that improving the efficiency of how POX's OpenFlow module gathers topology data could lead to significant decreases in controller load. Furthermore, decreasing the host liveliness timeout in the SD-MCAN application would also lead to less load, albeit at the cost of accuracy.

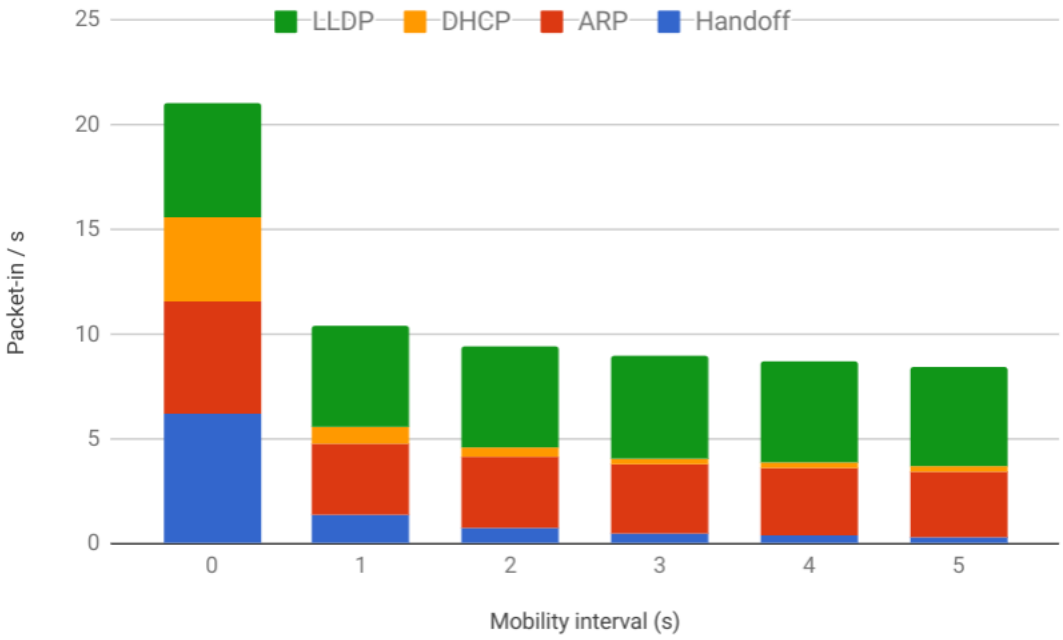

Figure 6.21: Packet-in Rate by Type

While the first experiment of this experiment set looked at controller load, the next experiment focuses on how the number of hosts on a network influences the size of the flow tables in the network's switches. The setup of this experiment resembles that of the last test. After loading the topology in Mininet, a new host is connected to a randomly selected LAN every second. After connecting each host, *dhclient* is run to lease an IP address. The connected host then pings another host on the network, selected at random, using the *ping* utility. This process continues until 1024 hosts have been connected to the network. After adding each host, a script is executed that processes the *ovs-ofctl dumpflows* command output to count the number of flows in all flow tables in the SD-MCAN switches. Figure 6.22 below shows a plot of the average number of flow table entries in both core and edge switches connected to the SD-MCAN controller given varying numbers of connected hosts.

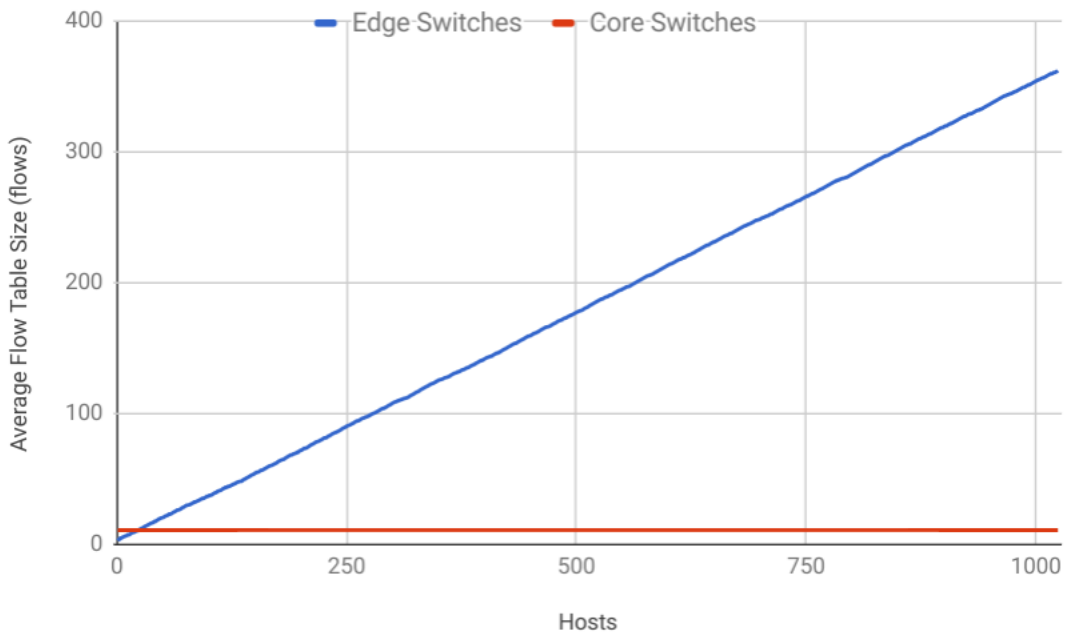

Figure 6.22: Flow Table Sizes

As expected, the number of flow entries in the core, label-switching switches remains constant regardless of the number of hosts on the network. This is because the core flow tables are updated proactively when the network topology stabilizes; host activity has no impact. Conversely, edge flow table entries are installed reactively in response to host activity on the network. The plot shows that edge flow tables scale linearly with the number of connected hosts. Clearly, the size of the edge switch flow tables limits the scalability of SD-MCAN; however, this is unavoidable as the label-pushing edge flow rules must rewrite the destination MAC addresses of the incoming packets. For SD-MCAN to be production-ready, the network must have hardware that supports large enough flow tables for the expected number of hosts.

6.4 Experiment Set D - Physical Testbed

This experiment set contains a single experiment, designed to confirm the transiency of handoff performance impact on vendor switches. To begin, a simple three-LAN topology is constructed, shown in Figure 6.23 below. Details of the switch configurations can be found in the Implementation chapter. In total, the topology features 6 switches and 8 links, each supporting 1 Gbps bandwidth; the switches connect to the controller out-of-band on a dedicated subnet. Two hosts, H1 and H2, connect to the network at switches S4 and S5, respectively. Both the hosts and the controller run Ubuntu 16.04 LTS and connect to the network via 1 Gbps interfaces. IPv6 and multicast are disabled on the host interfaces. H1 remains static, always attached to LAN1 via S4, while H2 moves around the topology acting as a mobile host.

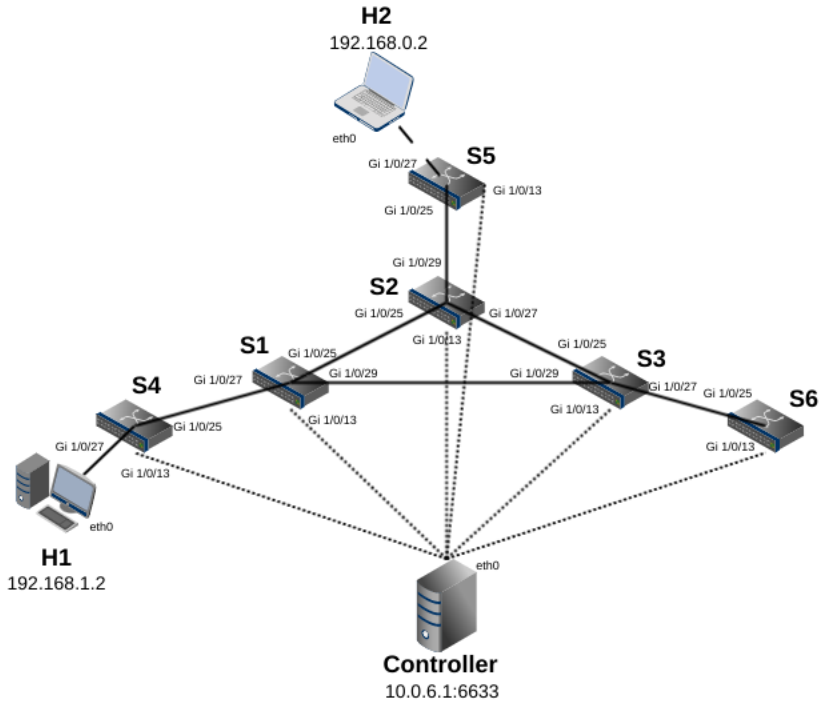

Figure 6.23: Physical Testbed Topology

As with experiment set B, this experiment set employs *Iperf* to generate UDP

traffic between the hosts. The stationary host H1 acts as the Iperf server, while the mobile host H2 acts as the client; Iperf monitors and records UDP packet loss and jitter, revealing how H2's movement impacts the network's performance. To establish a performance baseline on the physical testbed, Iperf is run between the two hosts for 60 seconds, during which time both hosts remain stationary. Figure 6.24 depicts the UDP packet loss, and Figure 6.25 shows the UDP jitter.

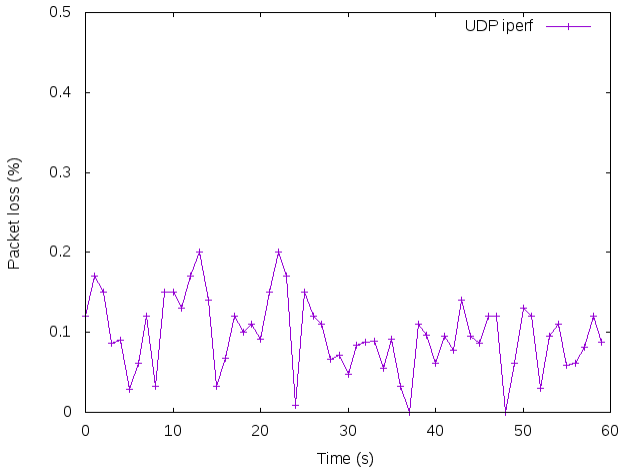

Figure 6.24: Physical Testbed Baseline UDP Packet Loss

Between the hosts, the packet loss is minimal, ranging between 0 and .2%. On the same connection, jitter ranges from .01 to .04 ms. These values are well below Cisco's acceptable packet loss and jitter thresholds and can be attributed to minor congestion on the host NICs. Figure 6.26 shows the Iperf connection summary, as reported by the client, H2. Iperf achieves 814 Mbps bandwidth between the hosts with a total packet loss of .097%.

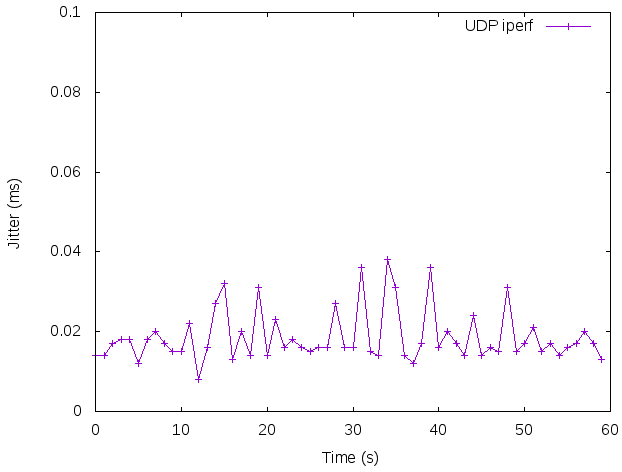

Figure 6.25: Physical Testbed Baseline UDP Jitter

```
Client connecting to 192.168.1.2, UDP port 5001
Sending 1470 byte datagrams
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[  3] local 192.168.0.2 port 56031 connected with 192.168.1.2 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-60.0 sec  5.69 GBytes   815 Mbits/sec
[  3] Sent 4156340 datagrams
[  3] Server Report:
[  3]  0.0-60.0 sec  5.68 GBytes   814 Mbits/sec   0.017 ms 4016/4156339 (0.097%)
```

Figure 6.26: Physical Testbed Baseline UDP Summary

With a baseline established, the experiment is run a again. This time, H2 walks around the topology, visiting each LAN before returning to its original LAN (making a total of 3 handoffs). Figure 6.27 shows the new UDP packet loss, while Figure 6.28 shows the new jitter. The packet loss plot shows significant spikes around each handoff (81% to 89% loss). At first glance, this appears significantly higher than the packet loss in experiment set B's simulated tests; however this discrepancy is due to the time taken to disconnect and reconnect H2 around the physical network. Iperf shows that each handoff takes about 3 seconds before the flow of packets resumes, at which point Iperf reports substantial loss. The jitter plot shows that the inter-LAN

handoffs make no impact on jitter; this is because Iperf's jitter reporting only factors in successfully transmitted packets.

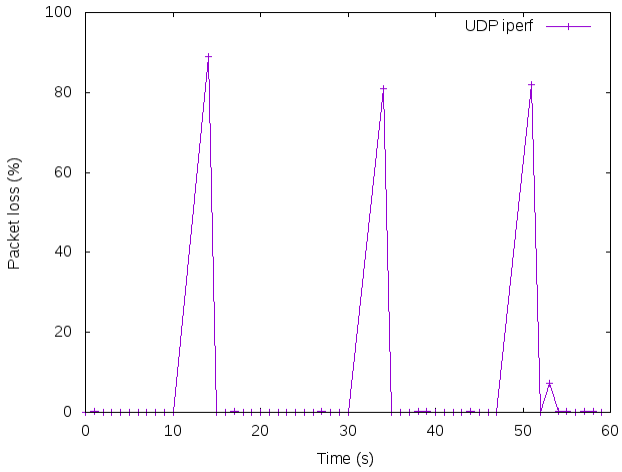

Figure 6.27: Physical Testbed Walk UDP Packet Loss

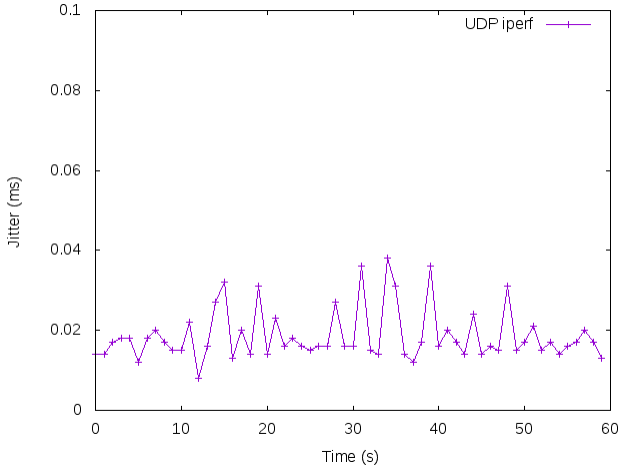

Figure 6.28: Pysical Testbed Walk UDP Jitter

Importantly, the graphs show that, as with the simulated tests, the performance degradation due to each handoff is transient, and the connection recovers quickly. The high packet loss reported after 3 seconds suggests that the speed at which a host passes between LANs will have significant impact on SD-MCAN's ability to provide fast handoffs; this observation comes as no surprise. TCP performance of SD-MCAN

on the physical testbed cannot yet be determined because the flow actions have to be restricted to prevent conflicts on the Cisco switches, causing TCP packets to be dropped by the receiving host network stack. Such tests will be possible once Cisco releases new OpenFlow software for the available Catalyst switches. While this test is limited, it functions as a proof-of-concept that SD-MCAN is deployable on vendor hardware, given the vendor properly supports the necessary OpenFlow components.

Chapter 7

FUTURE WORK

This thesis focuses on defining and validating the core functionality of the SD-MCAN architecture: providing IP mobility to CANs with support for fast handoffs. With that functionality verified, several areas of improvement remain before SD-MCAN is ready for production deployment. This thesis identifies two areas in which future research can refine the presented architecture: controller and security. This chapter provides insight on these areas of future research in the continued development of SD-MCAN into a production-ready architecture.

7.1 Controller Placement

The proposed SD-MCAN architecture employs a single controller at the network core. In [23], Heller et al. conclude that a single controller is adequate in many SDN scenarios. Indeed, experimental data from this work shows that a single controller is enough to handle all traffic in the target CAN topology. Regardless, controller placement in the SD-MCAN architecture is worth further investigation that this thesis chooses to defer to future work.

Additional research is required to determine if different controller placement in SD-MCAN would improve the reliability of the system. In [24], Hu et al. compare several controller placement selection algorithms. They find that, for certain topologies, placing both too few and too many controllers degrades performance. Future work on SD-MCAN should define metrics for evaluating the impact of controller placement on the network performance.

7.2 Security and Accessibility

SDN proves advantageous in many network scenarios, as this work has shown in CANs. While SDN can provide additional security benefits, it may also create new vulnerabilities [48]. Thus, a thorough investigation of the security implications of SD-MCAN is warranted before deployment. This section outlines potential security concerns and paths for research.

In its current iteration, SD-MCAN's only security comes throug the use of Transport Layer Security (TLS) in the secure channel between the controller and the switches. Unfortunately, the actual utilization of TLS depends on vendor support; TLS may not be used at all if the underlying switches do not support it. In [7], Benton et al. perform an analysis of the security vulnerabilities of OpenFlow. They find that widespread failure in TLS adoption leaves OpenFlow vulnerable to man-in-the-middle attacks. Without TLS, any OpenFlow-based system has no way of verifying that flow tables accurately enforce the controller's policy. A simple fix to this is to ensure that all switches and controllers on a production implementation of SD-MCAN implement TLS; however, this may come at a cost to network operators.

Benton et al. also recognize the denial-of-service (DOS) risks inherent in OpenFlow. The centralization of the controller provides benefits in respect to network management and control; however, centralized control, coupled with network programmability, also leads to increased risk of DOS attacks. Malicious users could overload the controller and flood switch flow tables with bogus flow entries. Benton's study recommends the use of multiple controllers with careful rule design to lessen the risk.

In [31], Kreutz et al. identify several threats to the security of SDN systems. They discuss forged traffic flows, switch vulnerabilities, attacks on control plane commu-

nication, and controller vulnerabilities as possible sources of failure in SDN systems. In response, they present what they call a *secure and dependable* SDN platform, advocating replication with diverse controllers. Additionally, they advocate writing self-healing mechanisms into controllers. A production SDN deployment should be robust and resistant to attack. As of this work, SD-MCAN has yet to implement these features. Future work should analyze the relevance of these potential security flaws to SD-MCAN and propose secure solutions.

Chapter 8

CONCLUSION

The types of devices, and the applications that those devices run, on college campuses indicate a clear need for mobility on campus area networks. Existing approaches like mobile IP provide mobility at the cost of inefficient routing and significant hardware updates. With the rise of SDN, decoupling control and data planes emerges as a valid solution to the mobility problem in CANs.

This work first identified four key design considerations which a successful SDN-based CAN mobility system must satisfy: compatibility, route efficiency, handoff latency, and scalability. Compatibility is a critical factor in making an SDN system deployable on a real CAN. As colleges have limited funding and may not be able to make significant changes to there preexisting networks, any SDN deployment should be deployable over the existing topology to ensure a cost-effective system. Route efficiency is necessary in ensuring that bandwidth is not wasted across the network and that users do not suffer high latency on their traffic flows. Users on CANs are often mobile; therefore any mobility solution needs to support fast handoffs, allowing users to migrate LANs without noticeable service disruptions. Finally, the SDN system should scale to satisfy the needs of the campus.

After specifying these requirements, this work proposed SD-MCAN, an OpenFlow-based architecture for enabling mobility in the CAN. Additionally, a POX-based SD-MCAN prototype was implemented to evaluate the performance of the proposed architecture. The system design allows for deployment on vendor switches with only two minor changes to the existing network. First, a controller must be added to the network, and second, the existing network devices must be updated to support OpenFlow if they do not already. Additionally, SD-MCAN mimics the behavior of

legacy routers, ensuring compatibility with existing protocols.

SD-MCAN utilizes a hybrid, label-switched routing scheme to provide both route efficiency and fast handoffs. At the network core, the controller proactively installs all core router flow entries as soon as the topology stabilizes. These core flow entries ensure that all inter-LAN packets are routed along optimal paths. Alternately, SD-MCAN reactively installs push and pop flow entries into edge routers in response to network packets. Coupled with aggressive flow removal, this reactive routing allows the controller to quickly update network paths for mobile hosts. Furthermore, short flow entry idle expiration times ensure that unused flows are quickly deleted, helping to limit the size of flow tables.

Experimental results show that the SD-MCAN prototype suffers only transient performance degradation (around 400 ms performance degradation) during host handoffs. The prototype handles host migration with minimal performance impact, packet loss <2% and throughput degradation <15%, when hosts move constantly at intervals greater than two seconds. The prototype also handles constant movement at shorter intervals, albeit with more significant performance impact (3.5% packet loss and 21% throughput degradation). While the SD-MCAN prototype is limited due to its Python implementation, analysis shows that a production quality SD-MCAN deployment is capable of handling host handoffs with <90 ms of negative performance impact, as, on average, 80% of the handoff period can be attributed to the time it takes to move the host and reconnect to the network. This suggests that SD-MCAN could support data-intensive services on mobile host devices.

Large scale experimentation shows that SD-MCAN's label-switched routing scheme keeps core router flow tables small (average 11 entries for 1024 mobile hosts); however, edge router flow tables scale more linearly with the number of hosts (average 362 entries for 1024 mobile hosts), limiting SD-MCAN's deployability on network

devices with restricted flow table sizes. While edge flow tables can grow large, experimental results show that the load on the controller remains reasonable (<21 packets per second with a host moving every 10 ms); thus, the presented SD-MCAN design is feasible on networks with devices containing adequately sized flow tables given the number of hosts on the network.

Finally, UDP tests on a physical testbed of Cisco switches in Cal Poly's networks lab show that the SD-MCAN suffers only transient performance degradation due to handoffs on vendor hardware. While the available hardware limits the prototype, the results suggest that a full-featured, production version of SD-MCAN is deployable on vendor hardware with proper OpenFlow support. Ultimately, SD-MCAN succeeds in meeting the identified criteria for IP mobility on CANs, suggesting that SDN's centralized network control is well-suited to the mobility needs of college area networks.

BIBLIOGRAPHY

[1] Cisco ios ip configuration guide, release 12.2 - configuring mobile ip [cisco ios software release 12.2], Feb 2014.

[2] Classified networking solutions policy - compliant commercial mobility technology with lower costs and higher performance, 2014.

[3] Nox. `https://github.com/noxrepo/nox.git`, 2014.

[4] Floodlight. `https://github.com/floodlight/floodlight.git`, 2017.

[5] Opendaylight. `https://github.com/daylight/controller.git`, 2017.

[6] Ryu. `https://github.com/osrg/ryu.git`, 2017.

[7] K. Benton, L. J. Camp, and C. Small. Openflow vulnerability assessment. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 151–152. ACM, 2013.

[8] C. Bernardos. Proxy mobile ipv6 extensions to support flow mobility. RFC 7864, RFC Editor, May 2016.

[9] S. Bhattacharjee, K. L. Calvert, and E. W. Zegura. An architecture for active networking. In *High Performance Networking VII*, pages 265–279. Springer, 1997.

[10] W. Braun and M. Menth. Software-defined networking using openflow: Protocols, applications and architectural design choices. *Future Internet*, 6(2):302–336, 2014.

[11] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and J. van der Merwe. Design and implementation of a routing control platform. In

Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation-Volume 2, pages 15–28. USENIX Association, 2005.

[12] K. Calvert. Reflections on network architecture: an active networking perspective. *ACM SIGCOMM Computer Communication Review*, 36(2):27–30, 2006.

[13] K. L. Calvert, S. Bhattacharjee, E. Zegura, and J. Sterbenz. Directions in active networks. *IEEE Communications Magazine*, 36(10):72–78, 1998.

[14] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking control of the enterprise. In *ACM SIGCOMM Computer Communication Review*, volume 37, pages 1–12. ACM, 2007.

[15] J. Chandrasekaran. Mobile ip: Issues, challenges and solutions. *Newark: Rutgers University*, 2009.

[16] W. Croft and J. Gilmore. Bootstrap protocol. RFC 951, RFC Editor, September 1985.

[17] J. Czyz, M. Allman, J. Zhang, S. Iekel-Johnson, E. Osterweil, and M. Bailey. Measuring ipv6 adoption. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 87–98, New York, NY, USA, 2014. ACM.

[18] S. E. Deering and R. M. Hinden. Internet protocol, version 6 (ipv6) specification. RFC 2460, RFC Editor, December 1998. `http://www.rfc-editor.org/rfc/rfc2460.txt`.

[19] R. Droms. Dynamic host configuration protocol. RFC 2131, RFC Editor, March 1997. `http://www.rfc-editor.org/rfc/rfc2131.txt`.

[20] N. Feamster, H. Balakrishnan, J. Rexford, A. Shaikh, and J. Van Der Merwe. The case for separating routing from routers. In *Proceedings of the ACM*

SIGCOMM workshop on Future directions in network architecture, pages 5–12. ACM, 2004.

[21] R. Finlayson, T. Mann, J. Mogul, and M. Theimer. A reverse address resolution protocol. STD 38, RFC Editor, June 1984.

[22] A. W. Group et al. Architectural framework for active networks. 1998.

[23] B. Heller, R. Sherwood, and N. McKeown. The controller placement problem. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 7–12. ACM, 2012.

[24] Y. Hu, W. Wendong, X. Gong, X. Que, and C. Shiduan. Reliability-aware controller placement for software-defined networks. In *Integrated Network Management (IM 2013), 2013 IFIP/IEEE International Symposium on*, pages 672–675. IEEE, 2013.

[25] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, et al. B4: Experience with a globally-deployed software defined wan. *ACM SIGCOMM Computer Communication Review*, 43(4):3–14, 2013.

[26] S. Kaur, J. Singh, and N. S. Ghumman. Network programmability using pox controller. In *ICCCS International Conference on Communication, Computing & Systems, IEEE*, number s 134, page 138, 2014.

[27] Z. K. Khattak, M. Awais, and A. Iqbal. Performance evaluation of opendaylight sdn controller. In *Parallel and Distributed Systems (ICPADS), 2014 20th IEEE International Conference on*, pages 671–676. IEEE, 2014.

[28] R. Khondoker, A. Zaalouk, R. Marx, and K. Bayarou. Feature-based comparison and selection of software defined networking (sdn) controllers. In

2014 World Congress on Computer Applications and Information Systems (WCCAIS), pages 1–7, Jan 2014.

[29] S.-M. Kim, H.-Y. Choi, P.-W. Park, S.-G. Min, and Y.-H. Han. Openflow-based proxy mobile ipv6 over software defined network (sdn). In *2014 IEEE 11th Consumer Communications and Networking Conference (CCNC)*, pages 119–125. IEEE, 2014.

[30] D. Kotz and K. Essien. Analysis of a campus-wide wireless network. *Wireless Networks*, 11(1-2):115–133, 2005.

[31] D. Kreutz, F. Ramos, and P. Verissimo. Towards secure and dependable software-defined networks. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 55–60. ACM, 2013.

[32] S. Krishnan, N. Montavont, E. Njedjou, S. Veerepalli, and A. Yegin. Link-layer event notifications for detecting network attachments. RFC 4957, RFC Editor, August 2007.

[33] D. Kumar and M. Sood. Software defined networks (sdn): Experimentation with mininet topologies. *Indian Journal of Science and Technology*, 9(32), 2016.

[34] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: Rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX, pages 19:1–19:6, New York, NY, USA, 2010. ACM.

[35] K.-H. Lee, D. Mougouei, M. K. Yeung, H. J. Park, J. H. Bae, and A. Hameed. Mobility management framework in software defined networks. *International Journal of Software Engineering and Its Applications*, 8(8):1–10, 2014.

[36] Y. Li, H. Wang, M. Liu, B. Zhang, and H. Mao. Software defined networking for distributed mobility management. In *2013 IEEE Globecom Workshops (GC Wkshps)*, pages 885–889. IEEE, 2013.

[37] M. McCauley. Pox. `https://github.com/noxrepo/pox.git`, 2013.

[38] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, Mar. 2008.

[39] B. A. A. Nunes, M. Mendonca, X. N. Nguyen, K. Obraczka, and T. Turletti. A survey of software-defined networking: Past, present, and future of programmable networks. *IEEE Communications Surveys Tutorials*, 16(3):1617–1634, Third 2014.

[40] C. Perkins. Ip mobility support. RFC 2002, RFC Editor, October 1996. `http://www.rfc-editor.org/rfc/rfc2002.txt`.

[41] C. Perkins. Ip mobility support for ipv4, revised. RFC 5944, RFC Editor, November 2010. `http://www.rfc-editor.org/rfc/rfc5944.txt`.

[42] C. Perkins, D. Johnson, and J. Arkko. Mobility support in ipv6. RFC 6275, RFC Editor, July 2011. `http://www.rfc-editor.org/rfc/rfc6275.txt`.

[43] C. E. Perkins, S. R. Alpert, and B. Woolf. *Mobile IP; Design Principles and Practices*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1997.

[44] B. Pfaff, B. Lantz, B. Heller, et al. Openflow switch specification, version 1.3. 0. *Open Networking Foundation*, 2012.

[45] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross,

A. Wang, J. Stringer, P. Shelar, et al. The design and implementation of open vswitch. In *NSDI*, pages 117–130, 2015.

[46] P. Pupatwibul, A. Banjar, A. A. Sabbagh, and R. Braun. Developing an application based on openflow to enhance mobile ip networks. In *Local Computer Networks Workshops (LCN Workshops), 2013 IEEE 38th Conference on*, pages 936–940. IEEE, 2013.

[47] E. Rosen, A. Viswanathan, and R. Callon. Multiprotocol label switching architecture. RFC 3031, RFC Editor, January 2001. `http://www.rfc-editor.org/rfc/rfc3031.txt`.

[48] S. Scott-Hayward, G. O'Callaghan, and S. Sezer. Sdn security: A survey. In *Future Networks and Services (SDN4FNS), 2013 IEEE SDN For*, pages 1–7. IEEE, 2013.

[49] A. Shalimov, D. Zuikov, D. Zimarina, V. Pashkov, and R. Smeliansky. Advanced study of sdn/openflow controllers. In *Proceedings of the 9th central & eastern european software engineering conference in russia*, page 1. ACM, 2013.

[50] H. Soliman, C. Castelluccia, K. ElMalki, and L. Bellier. Hierarchical mobile ipv6 (hmipv6) mobility management. RFC 5380, RFC Editor, October 2008. `http://www.rfc-editor.org/rfc/rfc5380.txt`.

[51] A. Tanenbaum. *Computer Networks*. Prentice Hall Professional Technical Reference, 4th edition, 2002.

[52] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A survey of active network research. *IEEE communications Magazine*, 35(1):80–86, 1997.

[53] Y. Wang and J. Bi. Software-defined mobility support in ip networks. *The Computer Journal*, 59(2):159–177, 2016.

[54] Y. Wang, J. Bi, and K. Zhang. Design and implementation of a software-defined mobility architecture for ip networks. *Mob. Netw. Appl.*, 20(1):40–52, Feb. 2015.

[55] Z. Wang, T. Tsou, J. Huang, X. Shi, and X. Yin. Analysis of comparisons between openflow and forces. *March*, 2012.

[56] L. Yang, R. Dantu, T. Anderson, and R. Gopal. Forwarding and control element separation (forces) framework. RFC 3746, RFC Editor, April 2004.

[57] M. Yu, J. Rexford, X. Sun, S. Rao, and N. Feamster. A survey of virtual lan usage in campus networks. *IEEE Communications Magazine*, 49(7), 2011.

APPENDIX: EXPERIMENT A2 FLOW TABLE PRINTOUTS

```
S1 NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=17.340s, table=0, n_packets=10, n_bytes=410, idle_age=0, priority=65000,dl_dst=01:23:20:00:00:01,dl_type=0x88cc actions=CONTROLLER:6553
 cookie=0x0, duration=17.303s, table=0, n_packets=0, n_bytes=0, idle_age=17, priority=32769,arp,dl_dst=02:00:00:00:be:ef actions=CONTROLLER:65535
 cookie=0x0, duration=17.303s, table=0, n_packets=0, n_bytes=0, idle_age=17, udp,tp_src=68,tp_dst=67 actions=CONTROLLER:65535
 cookie=0x0, duration=3.993s, table=0, n_packets=3, n_bytes=306, idle_age=1, dl_vlan=16 actions=mod_vlan_vid:17,output:1
 cookie=0x0, duration=3.992s, table=0, n_packets=0, n_bytes=0, idle_age=3, dl_vlan=19 actions=mod_vlan_vid:20,output:2
 cookie=0x0, duration=3.990s, table=0, n_packets=3, n_bytes=306, idle_age=1, dl_vlan=23 actions=mod_vlan_vid:24,output:3
 cookie=0x0, duration=3.987s, table=0, n_packets=0, n_bytes=0, idle_age=3, dl_vlan=28 actions=mod_vlan_vid:24,output:3

S2 NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=17.343s, table=0, n_packets=10, n_bytes=410, idle_age=0, priority=65000,dl_dst=01:23:20:00:00:01,dl_type=0x88cc actions=CONTROLLER:6553
 cookie=0x0, duration=17.304s, table=0, n_packets=0, n_bytes=0, idle_age=17, priority=32769,arp,dl_dst=02:00:00:00:be:ef actions=CONTROLLER:65535
 cookie=0x0, duration=17.304s, table=0, n_packets=0, n_bytes=0, idle_age=17, udp,tp_src=68,tp_dst=67 actions=CONTROLLER:65535
 cookie=0x0, duration=3.998s, table=0, n_packets=3, n_bytes=306, idle_age=1, dl_vlan=17 actions=mod_vlan_vid:18,output:3
 cookie=0x0, duration=3.995s, table=0, n_packets=3, n_bytes=306, idle_age=1, dl_vlan=22 actions=mod_vlan_vid:23,output:1
 cookie=0x0, duration=3.994s, table=0, n_packets=0, n_bytes=0, idle_age=3, dl_vlan=25 actions=mod_vlan_vid:26,output:2
 cookie=0x0, duration=3.990s, table=0, n_packets=0, n_bytes=0, idle_age=3, dl_vlan=30 actions=mod_vlan_vid:18,output:3

S3 NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=17.353s, table=0, n_packets=10, n_bytes=410, idle_age=1, priority=65000,dl_dst=01:23:20:00:00:01,dl_type=0x88cc actions=CONTROLLER:6553
 cookie=0x0, duration=17.314s, table=0, n_packets=0, n_bytes=0, idle_age=17, priority=32769,arp,dl_dst=02:00:00:00:be:ef actions=CONTROLLER:65535
 cookie=0x0, duration=17.314s, table=0, n_packets=0, n_bytes=0, idle_age=17, udp,tp_src=68,tp_dst=67 actions=CONTROLLER:65535
 cookie=0x0, duration=4.002s, table=0, n_packets=0, n_bytes=0, idle_age=4, dl_vlan=20 actions=mod_vlan_vid:21,output:3
 cookie=0x0, duration=3.999s, table=0, n_packets=0, n_bytes=0, idle_age=3, dl_vlan=26 actions=mod_vlan_vid:21,output:3
 cookie=0x0, duration=3.998s, table=0, n_packets=0, n_bytes=0, idle_age=3, dl_vlan=27 actions=mod_vlan_vid:28,output:1
 cookie=0x0, duration=3.997s, table=0, n_packets=0, n_bytes=0, idle_age=3, dl_vlan=29 actions=mod_vlan_vid:30,output:2

S4 NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=17.352s, table=0, n_packets=3, n_bytes=123, idle_age=4, priority=65000,dl_dst=01:23:20:00:00:01,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x0, duration=17.312s, table=0, n_packets=0, n_bytes=0, idle_age=17, priority=32769,arp,dl_dst=02:00:00:00:be:ef actions=CONTROLLER:65535
 cookie=0x0, duration=17.312s, table=0, n_packets=2, n_bytes=684, idle_age=2, udp,tp_src=68,tp_dst=67 actions=CONTROLLER:65535
 cookie=0x0, duration=2.042s, table=0, n_packets=2, n_bytes=196, idle_timeout=10, idle_age=1, ip,dl_dst=03:00:00:00:be:ef,nw_dst=192.168.1.2
actions=mod_dl_dst:4e:c0:c1:7f:07:34,mod_vlan_vid:16,output:1
 cookie=0x0, duration=1.972s, table=0, n_packets=3, n_bytes=306, idle_timeout=10, idle_age=1, ip,dl_vlan=24 actions=strip_vlan,output:2

S5 NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=17.358s, table=0, n_packets=3, n_bytes=123, idle_age=3, priority=65000,dl_dst=01:23:20:00:00:01,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x0, duration=17.328s, table=0, n_packets=0, n_bytes=0, idle_age=17, priority=32769,arp,dl_dst=02:00:00:00:be:ef actions=CONTROLLER:65535
 cookie=0x0, duration=17.328s, table=0, n_packets=2, n_bytes=684, idle_age=2, udp,tp_src=68,tp_dst=67 actions=CONTROLLER:65535
 cookie=0x0, duration=2.046s, table=0, n_packets=3, n_bytes=306, idle_timeout=10, idle_age=1, ip,dl_vlan=18 actions=strip_vlan,output:2
 cookie=0x0, duration=1.977s, table=0, n_packets=2, n_bytes=196, idle_timeout=10, idle_age=1, ip,dl_dst=03:00:00:00:be:ef,nw_dst=192.168.0.2
actions=mod_dl_dst:9e:e8:52:3f:4e:5e,mod_vlan_vid:22,output:1

S6 NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=17.373s, table=0, n_packets=3, n_bytes=123, idle_age=5, priority=65000,dl_dst=01:23:20:00:00:01,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x0, duration=17.334s, table=0, n_packets=0, n_bytes=0, idle_age=17, priority=32769,arp,dl_dst=02:00:00:00:be:ef actions=CONTROLLER:65535
 cookie=0x0, duration=17.334s, table=0, n_packets=0, n_bytes=0, idle_age=17, udp,tp_src=68,tp_dst=67 actions=CONTROLLER:65535
```

Figure A.1: Experiment A2 Flow Tables 1

```
S1 NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=19.544s, table=0, n_packets=11, n_bytes=451, idle_age=0, priority=65000,dl_dst=01:23:20:00:00:01,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x0, duration=19.507s, table=0, n_packets=0, n_bytes=0, idle_age=19, priority=32769,arp,dl_dst=02:00:00:00:be:ef actions=CONTROLLER:65535
 cookie=0x0, duration=19.507s, table=0, n_packets=0, n_bytes=0, idle_age=19, udp,tp_src=68,tp_dst=67 actions=CONTROLLER:65535
 cookie=0x0, duration=6.197s, table=0, n_packets=4, n_bytes=408, idle_age=2, dl_vlan=16 actions=mod_vlan_vid:17,output:1
 cookie=0x0, duration=6.196s, table=0, n_packets=3, n_bytes=306, idle_age=1, dl_vlan=19 actions=mod_vlan_vid:20,output:2
 cookie=0x0, duration=6.194s, table=0, n_packets=4, n_bytes=408, idle_age=2, dl_vlan=23 actions=mod_vlan_vid:24,output:3
 cookie=0x0, duration=6.191s, table=0, n_packets=4, n_bytes=408, idle_age=0, dl_vlan=28 actions=mod_vlan_vid:24,output:3

S2 NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=19.548s, table=0, n_packets=11, n_bytes=451, idle_age=1, priority=65000,dl_dst=01:23:20:00:00:01,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x0, duration=19.509s, table=0, n_packets=0, n_bytes=0, idle_age=19, priority=32769,arp,dl_dst=02:00:00:00:be:ef actions=CONTROLLER:65535
 cookie=0x0, duration=19.509s, table=0, n_packets=0, n_bytes=0, idle_age=19, udp,tp_src=68,tp_dst=67 actions=CONTROLLER:65535
 cookie=0x0, duration=6.203s, table=0, n_packets=4, n_bytes=408, idle_age=2, dl_vlan=17 actions=mod_vlan_vid:18,output:3
 cookie=0x0, duration=6.200s, table=0, n_packets=4, n_bytes=408, idle_age=2, dl_vlan=22 actions=mod_vlan_vid:23,output:1
 cookie=0x0, duration=6.199s, table=0, n_packets=0, n_bytes=0, idle_age=6, dl_vlan=25 actions=mod_vlan_vid:26,output:2
 cookie=0x0, duration=6.195s, table=0, n_packets=0, n_bytes=0, idle_age=6, dl_vlan=30 actions=mod_vlan_vid:18,output:3

S3 NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=19.558s, table=0, n_packets=12, n_bytes=492, idle_age=0, priority=65000,dl_dst=01:23:20:00:00:01,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x0, duration=19.519s, table=0, n_packets=0, n_bytes=0, idle_age=19, priority=32769,arp,dl_dst=02:00:00:00:be:ef actions=CONTROLLER:65535
 cookie=0x0, duration=19.519s, table=0, n_packets=0, n_bytes=0, idle_age=19, udp,tp_src=68,tp_dst=67 actions=CONTROLLER:65535
 cookie=0x0, duration=6.207s, table=0, n_packets=3, n_bytes=306, idle_age=1, dl_vlan=20 actions=mod_vlan_vid:21,output:3
 cookie=0x0, duration=6.204s, table=0, n_packets=0, n_bytes=0, idle_age=6, dl_vlan=26 actions=mod_vlan_vid:21,output:3
 cookie=0x0, duration=6.203s, table=0, n_packets=4, n_bytes=408, idle_age=0, dl_vlan=27 actions=mod_vlan_vid:28,output:1
 cookie=0x0, duration=6.202s, table=0, n_packets=0, n_bytes=0, idle_age=6, dl_vlan=29 actions=mod_vlan_vid:30,output:2

S4 NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=19.557s, table=0, n_packets=4, n_bytes=164, idle_age=1, priority=65000,dl_dst=01:23:20:00:00:01,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x0, duration=19.517s, table=0, n_packets=0, n_bytes=0, idle_age=19, priority=32769,arp,dl_dst=02:00:00:00:be:ef actions=CONTROLLER:65535
 cookie=0x0, duration=19.517s, table=0, n_packets=2, n_bytes=684, idle_age=4, udp,tp_src=68,tp_dst=67 actions=CONTROLLER:65535
 cookie=0x0, duration=2.032s, table=0, n_packets=4, n_bytes=408, idle_timeout=10, idle_age=0, ip,dl_vlan=24 actions=strip_vlan,output:2
 cookie=0x0, duration=2.040s, table=0, n_packets=4, n_bytes=392, idle_timeout=10, idle_age=0, ip,dl_dst=03:00:00:00:be:ef,nw_dst=192.168.1.2
actions=mod_dl_dst:4e:c0:c1:7f:07:34,mod_vlan_vid:19,output:1

S5 NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=19.563s, table=0, n_packets=3, n_bytes=123, idle_age=5, priority=65000,dl_dst=01:23:20:00:00:01,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x0, duration=19.533s, table=0, n_packets=0, n_bytes=0, idle_age=19, priority=32769,arp,dl_dst=02:00:00:00:be:ef actions=CONTROLLER:65535
 cookie=0x0, duration=19.533s, table=0, n_packets=2, n_bytes=684, idle_age=4, udp,tp_src=68,tp_dst=67 actions=CONTROLLER:65535
 cookie=0x0, duration=4.251s, table=0, n_packets=4, n_bytes=408, idle_timeout=10, idle_age=2, ip,dl_vlan=18 actions=strip_vlan,output:2
 cookie=0x0, duration=4.182s, table=0, n_packets=2, n_bytes=196, idle_timeout=10, idle_age=3, ip,dl_dst=03:00:00:00:be:ef,nw_dst=192.168.0.2
actions=mod_dl_dst:9e:e8:52:3f:4e:5e,mod_vlan_vid:22,output:1

S6 NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=19.579s, table=0, n_packets=4, n_bytes=164, idle_age=2, priority=65000,dl_dst=01:23:20:00:00:01,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x0, duration=19.540s, table=0, n_packets=0, n_bytes=0, idle_age=19, priority=32769,arp,dl_dst=02:00:00:00:be:ef actions=CONTROLLER:65535
 cookie=0x0, duration=19.540s, table=0, n_packets=1, n_bytes=342, idle_age=2, udp,tp_src=68,tp_dst=67 actions=CONTROLLER:65535
 cookie=0x0, duration=2.051s, table=0, n_packets=3, n_bytes=306, idle_timeout=10, idle_age=1, ip,dl_vlan=21 actions=strip_vlan,output:12
 cookie=0x0, duration=2.043s, table=0, n_packets=3, n_bytes=294, idle_timeout=10, idle_age=0, ip,dl_dst=03:00:00:00:be:ef,nw_dst=192.168.0.2
actions=mod_dl_dst:9e:e8:52:3f:4e:5e,mod_vlan_vid:27,output:1
```

Figure A.2: Experiment A2 Flow Tables 2