IMPLEMENTING MPLS WITH LABEL SWITCHING IN SOFTWARE-DEFINED NETWORKS

BY

JOHN BELLESSA

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2015

Urbana, Illinois

Adviser:

Professor Roy H. Campbell

Abstract

Label-based switching schemes, such as MPLS, have been known to be an effective mechanism in overcoming many challenges in core networks. Software-defined networking, while a much more recent development, is seen to have the potential to revolutionize networking. But some network researchers, even within the SDN community, believe, for SDN to flourish, it must adopt a more structured model with an intelligent edge and a fast but simple label switched core. This is an excellent use case for MPLS on OpenFlow. But, while there have been other implementations of MPLS in OpenFlow, they abandon the spirit OpenFlow by requiring sophisticated hardware. This thesis discusses our hybrid-OpenFlow implementation of MPLS that requires only commodity hardware in the core network. We accomplish this by compiling the MPLS labels that would have been encountered along a path through the network into a single label, which is stored in the packets' destination MAC address field.

To Lorraine, for her enduring patience and persisting love.

Acknowledgments

I first have to thank my fiancée, Lorraine, once again. Without her daily encouragement, support, and love, I would have never gotten past day one.

I would also like to thank my parents, Garrett and Marilyn Bellessa for their love and support.

I thank Professor Roy Campbell for his advice, guidance, dedication, and sense of humor.

I would like to thank those who helped on this project, Sayed Hadi Hashemi, and Shadi Abdollahian. They were and continue to be integral to its success.

I would also like to thank various members of the SRG, past and present, for their friendship, advice, and support. They are Mayank Pundir, Read Sprabery, Faraz Faghri, Reza Farivar, and Mirko Montanari.

Finally, I would be remiss if I did not thank two close friends and classmates from my undergraduate studies, Michael Hogue and Sobir Bazarbayev. It was their friendship, intellect, and dedication that gave me the hope and the will to relentlessly move forward. I am forever indebted to them.

Table of Contents

Chapter 1

Introduction

Although SDN is widely seen as a significant step forward towards a completely re-envisioned paradigm for modern packet-switched networks, current incarnations – most notably, OpenFlow – appear to fall short on these promises. This should not be regarded as a particularly controversial statement, as several early OpenFlow contributors, Casado, et al, acknowledged as much in 2012 [1]. Rule matching in SDN compatible switches is very memory intensive compared to current core network technologies which make it costly to adapt. On other hand, it does not provide sufficient flexibility to the network. SDN unnecessarily couples the host requirements to the network core behavior. All flows are identified in the whole network by parameters of the header set by the host.

In [1], the authors describe the network engineering design principle that all serious "networking logic" should exist only at the edge of the network (i.e., points of ingress and egress), while the core of the network should be kept as simple as possible. One highly effective way to achieve this is through label-based switching. We believe that this is a golden opportunity for MPLS, or some other of label-based switching, to make its mark on the future of SDN. And while some implementations exist, they require more advanced, and even optional, features of OpenFlow that may not be widely available across multiple vendors.

In this thesis, we present a hybrid-OpenFlow implementation of MPLS, built on top of a simple layer-2 label mapping technique. Because the labels do not require any modifications to packet structure, it can be deployed using only commodity switches in the core network.

Main contributions of this work are:

1. Implement label switching on commodity SDN hardware using L2 addressing as labels

2. Mapped multiple dynamic labels to static labels in real time

3. Implement MPLS supplementary protocols such as LDP, RSVP-TE

The rest of this thesis is structured as follows: Chapter 2 discusses work related to ours, Chapter 3 describes our method for implementing MPLS and general label switching in OpenFlow, Chapter 4 discusses our evaluations, and we conclude with Chapter 5.

Chapter 2

Related Works

2.1 Software-Defined Networking

In traditional networks, the logic that decides how packets should be forwarded through the network and the mechanism that is responsible for actually forwarding the packets are tightly coupled; these two constructs are typically referred to as the control-plane and the data-plane (or forwarding-plane), respectively. Additionally, the whole network structure is highly decentralized; the configuration of one device can be entirely independent of all others. These characteristics are the result of a natural evolution of networking technologies; they make the network resilient to failure, and provide certain performance guarantees. In practice, however, these qualities make large networks exceedingly difficult to manage, debug, and reconfigure.

Software-defined networking, or SDN, seeks to resolve these challenges by completely re-thinking network management. At its core, SDN has two defining characteristics: (1) the decoupling of data- and control-planes, and (2) centralized control of the entire network. From these two features emerges a third: programmability. By exposing centralized view and control over an entire network infrastructure, new functionality can be added as needed.

The key element in a software-defined network is called the SDN controller, which is also sometimes called the network operating system (NOS). SDN discovers and updates a global abstract network view to facilitate the central control logic. SDN terminology can be described bottom-up as: Forwarding Devices, Data Plane, Southbound Interface, Control Plane, Northbound Interface, and Management Plane[2].

In software-defined networking, the network OS controls the entire network state centrally via forwarding plane. Compared to traditional network, SDN enables the network operators to define the functionality after the network is deployed. New features can be added in software without modifying the switches. SDN enables new approaches to state management and new uses of packet headers [3]. An intensive research has been conducted on SDN. Commonly, mappings between virtual abstractions and physical implementations are one-to-many, e.g. a single switch abstraction is implemented using a distributed set of physical devices. [4] shows that the traditional per-packet consistency is not enough for one-to-many mapping correctness and introduces new research directions: (a) developing more advanced mapping techniques or (b) restricting the API of SDN to provide safe-to-map abstractions only. [5] develops a distributed architecture that provides a global network view to applications on physically distributed servers with high availability and scale-out. [6] extends SDN architecture to enable application aware SDN data plane by examining information beyond layer 2-4 headers.

2.1.1 OpenFlow

The most notable example of such an API is OpenFlow [7]. Initially, OpenFlow aspired only to academia; it was originally proposed as a way to enable experiments in production networks without causing the threat of outages to critical networks. Upon its release, however, its potential recognized almost immediately and has gained significant interest from both academia and industry. For example, Google uses OpenFlow to manage traffic within and between data centers [8].

OpenFlow itself is actually a flow-level switch management protocol. The OpenFlow specification defines an API that, at its most basic, allows for switches to ask an external controller how to handle packets, and for the controller to respond with instructions. These instructions are in the form of "flow rules", which consist of two parts: a flow description and a set of actions. A flow description is essentially a 12-tuple of header fields from L2-L4, any of which can be wildcards. The actions of a flow rule describe what the switch should do when it receives a packet matching that rule's description; this is typically specifying out which port the switch should send the packet, though there is a wide variety of possible actions.

2.2 Label Switching

QoS (Quality of Service) entails providing better services to selected traffic, without disrupting other flows [9]. The central idea behind QoS is to provide differentiated services based on application needs. However, IP itself does not provide any sort of QoS capability as it always routes through the shortest path to the destination. Multilayer switching overcomes this limitation by providing ISPs more fine-grained control over network traffic within the core network. The fundamental building blocks for any multilayer switching solution are the separation of the control and data planes, and the label-swapping forwarding algorithm [10]. The former entails that multilayer label switching solutions comprise of two functional units - the control unit and the forwarding unit. The control unit uses some standard routing algorithm to update entries in a forwarding table, while the forwarding unit itself is used to make routing decisions in a packet. Label-swapping forwarding algorithms use labels, where a label is short, fixed-length value carried in the packets header to identify a Forwarding Equivalence Class (FEC) [10]. An FEC encapsulates a set of packets that should be routed along the same path, regardless of what their final destination may be. The algorithm requires that packets are classified into labels at the edges of the node, as the labels are then used within the core to route packets [11]. MPLS is a standard presented by the IEFT that seeks to standardize a label switching protocol while incorporating features from various proprietary vendors [10]. MPLS provides flexibility to the ISP in determining which path is taken by a packet. One of the significant advantages of MPLS is that it empowers ISPs to deliberately engineer the flow of traffic within the network at a more fine-grained level - resulting in a network that is more efficiently operated, supports more predictable service, and can offer the flexibility required to meet constantly changing customer expectations. [10]

2.3 Label Switching with OpenFlow

Although SDN has been considered as a significant step forward in some aspects, several of the early OpenFlow evangelists, point out several severe limitations in [1]. On top of all, SDN does not fulfill the promise of simplified hardware. Rule matching in SDN compatible switches is very memory intensive (TCAM) compared to other core network technologies such as MPLS. On other hand, it does not provide sufficient flexibility to the network. Adding a functionality to OpenFlow needs to be implemented in all network devices including core, distributed, and access switches which is not the case with commodity networks. One of the possible reasons for this problem is that SDN unnecessarily couples the host requirements to the network core behavior. All flows are identified in the whole network by parameters of the header set by the host.

2.3.1 MPLS Extensions for OpenFlow

Many attempts from academia have been made to combine SDN with MPLS. Work by Kempf, et al [12], extends canonical OpenFlow 1.0 architecture by adding MPLS actions in virtual port table consisting of `push_mpls`, `pop_mpls`, `swap_mpls`, `decrement_ttl`, `copy_bits`. [13] extends GMPLS, a MPLS successor in optical circuit switched domain, and implements an extended OpenFlow controller. The edge packet switches that interconnect the optical circuit switched domain to the packet switched domains are equipped with tunable WDM interfaces and are connected to an add/drop port of the ingress/egress nodes of the optical domain.

2.3.2 Fabric Networks

The notion of a “fabric network” is one possible solution to the current limitations of SDN [1]. This work adopts the idea of label switching concept and separates addressing and controlling the core network from access network. Ingress / egress edge switches s idea not only reduced the required memory in core switches by simplifying forwarding rules, but let edge and core networks involved separately. For example, migration from IPv4 to IPv6 can be done on edge network, without requiring any hardware or software updates in core network.

2.3.3 Layer-2 Labels

One of the biggest challenges with fabric networks is compatibility with current generation of hardwares, controllers, and protocols . It need dramatic changes to the OpenFlow, switches, and SDN controllers. In a meanwhile some projects such as [14] or [15] try to implement part of the fabric network using existing technologies.

Shadow MACs [14], a project done at IBM, uses layer-2 (L2) addressing to implement the fabric network. In this project, they exploit the fact that commodity switches has larger memory and more efficient implementation for L2 address matching than OpenFlow header matching. Therefore, the fabric network labels are saved as L2 destination addresses. Current switches are able to automatically match this address with L2 memory upon packet arrivals. This, if executed properly, L2 label switching can, in effect, achieve some of the desired out comes of fabric networks. This makes the idea of repurposing the L2 destination field for switching labels very appealing.

One limitation of this work is only one label can be assign to a packet, unlike MPLS in which multiple stacked labels can be used in a packet. The work in [15] address this problem by verifying that all paths has an assigned label, and possibly merge some labels in case of having more labels than available memory on switches.

Chapter 3

Method for Implementing MPLS-like Label Switching in SDN

As mentioned above, our goal is to bring the same capability as MPLS to SDNs running commodity hardware. To achieve this goal we need to map several concepts to the SDN.

3.1 Architecture

Similar to MPLS architecture, the proposed method works on a core network. Access to this network is through the edge switches, which tag incoming packets with a label, and remove the label from outgoing packets. Core switches in the core network use the label solely for switching. The architecture is shown in Figure 3.1.

When a packet reaches an edge switch that does not have a matching flow rule, the switch requests instructions from the controller. The controller will then determine the path that this packet, and other matching packets, should take through the network; the controller also creates a corresponding label and then installs the label in the switches along the packet's path.

3.2 Layer-2 Labels

One challenge for a label switching implementation is determining how the label will be tagged onto packets. There are only so many places a label can be added to a packet. But, regardless, there are some options that, at first glance, seem to be natural choices. First of all, why not simply use actual MPLS labels? Initially, MPLS labels seem to be a good option, especially given that the ability to use MPLS labels has been part of the OpenFlow specification since version 1.1 [16]. However, this is feature is purely optional. Thus, we cannot reliably count on switches – especially commodity switches

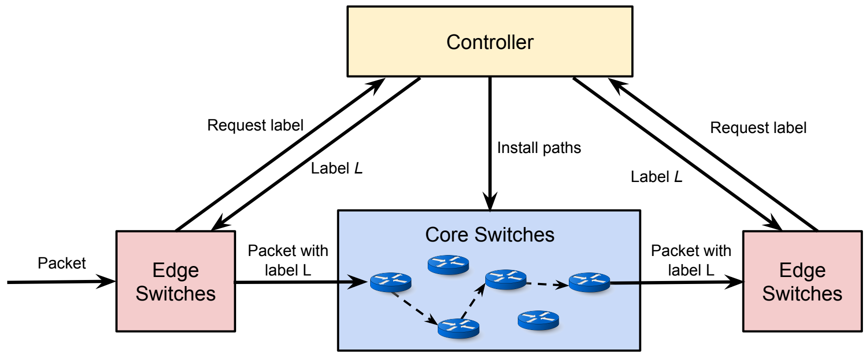

Figure 3.1: Edge switches notify the controller of unrecognized packets. The controller determines the flow's path, generates a label, and installs rules to send packets with matching label along the path.

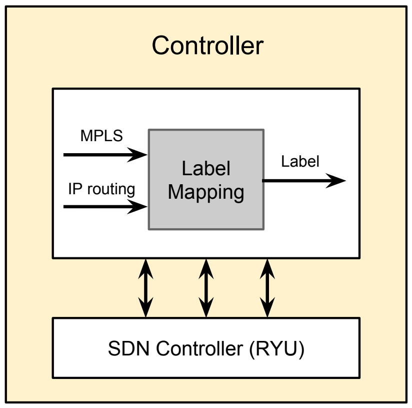

Figure 3.2: The label mapping module is implemented on top of the Ryu controller.

– to be able to manipulate MPLS tags, even those implementing OpenFlow 1.1. This is also the case for VLAN tags.

Another option, which we do in fact use, is to store the switching label in the packet's destination MAC address field, as was also done in the Shadow MACs project [14]. While, initially, this may seem less intuitive, it turns out to be quite a convenient choice for several reasons. First, most commodity switches feature ASICs, or some other highly optimized mechanism, for matching packets on the destination MAC field, meaning that it is an extremely fast operation. Most switches have a hardware look-up table for MAC addresses. These MAC tables tend to be significantly larger than TCAM memory, which is used for routing tables, or OpenFlow rules. The second reason for using the destination MAC address field is that, since all forwarding decisions are determined solely by the switching label, layer-2 addressing in not used in the core network. Therefore, the destination MAC address field is a natural choice for store the switching label.

However, this choice does have some limitations. While, perfectly useful for a general label switching scheme, the specified functionality of MPLS can be directly replicated. In particular, MPLS allows for an arbitrary number of labels to be pushed onto a packet's "label stack". To overcome this problem, we introduce the concept of "Label Mapping". The idea is to convert all labels that will be assigned to a particular packet into a static label. This static label will be used later by our method. This process is described in Section 3.4.

3.3 Forwarding

Labels are used to direct a packet through a specific path. This path is implemented as sets of forwarding entries installed in core switches, and a few OpenFlow rules on the edge switches.

To implement forwarding tables on core switches we considered two different approaches:

1. **MAC Address Table:** In this method, each forwarding rule is added to the switch's MAC address table as a static ARP entry. This entry contains the value of the label, and the destination port number on the switch.

2. **TCAM:** In this method, each forwarding rule is added to TCAM memory as a OpenFlow entry. This entry has the label value as the l2 destination and wildcard for other fields. Also the corresponding action of the rule is to send the packet to a specified port on the switch.

While both methods behave similarly, there are some differences between them, with each having benefits and drawbacks. As our implementation is meant to be used in an OpenFlow environment, TCAM would appear to not only be the obvious choice, but the *only* choice. This is because typical OpenFlow switch implementations only use TCAM for storing rules; and even if a "rogue" implementation were to leverage the MAC table, there is no mechanism in the OpenFlow standard to give a controller the option to use it. [1]

However, using the MAC table offers enough benefits that it is worth at least exploring. As discussed above, all switches have highly optimized abilities to match a packet's destination MAC address against the entries in the MAC table. But, while this may even be faster than accessing data in TCAM, it will not be orders-of-magnitude faster. What really makes the MAC table such an attractive option is that it is dramatically less expensive, in terms of both up-front costs, as well as long-term power consumption.

Using the MAC table also comes with its own set of drawbacks. First and foremost, as we discussed above, there is no ability for an OpenFlow controller to manipulate a switch's MAC table out-of-the-box, meaning we would be required to find a way to side-load labels into the switches' MAC tables. The second drawback is the lack of support for partial wildcard look-ups. While not an issue for our purposes, this does preclude some potentially useful strategies found in other recent work [15].

After weighing the issues, we ultimately decided to pursue the MAC table option for core fabric switches. We felt that this option most closely aligns with our goal for supporting as much commodity hardware as possible, as it would allow for far more label-matching rules on significantly cheaper switches with smaller TCAMs. However, as we will discuss in Section 4.2.1, this may not be as significant as we first initially expected.

[1]As an aside, from the perspective of OpenFlow designers, this is perfectly sensible. TCAM allows for constant-time look-ups of arbitrary data, including wildcards. It is precisely what one needs in order to give OpenFlow the flexibility and generality that it requires.

3.4 Label Updating

Unlike MPLS, the proposed method will not need to have “Label Distribution Protocol” or “Resource Reservation Protocol with Traffic Engineering” to automatically setup labels. Since SDN has a global view of the network, a new label can be assigned to a new path in the controller. After that the necessary forwarding rules will be installed on the switches via the controller.

3.5 Label Mapping

MPLS forwarding occurs with the use of MPLS labels, which are organized into a stack. There are two significant differences between IP and MPLS based routing schemes.

1. IP based routing occurs using IP address based lookup, while MPLS based routing occurs based on a stack of labels. The top label is used to determine the next hop using the Lable Forwarding Information Base. The LFIB is fundamentally a forwarding table that uses a label to determine the next hop.

2. MPLS based routing also supports label operations on the stack of labels.

When packets are received at the ingress router, the ingress router can append a stack of labels to the packet. These labels are then used to engineer the flow of packets through the network. At every node, the label at the top of the stack is observed, and certain operations are performed on the stack. The fundamental operations permitted are push, pop and swap. These operations are illustrated in Figure 3.3. Then, the top of the stack (after all operations have been performed) is observed and used to find the next hop to which the packet is forwarded. At the egress, the labels are all popped and the packet is forwarded using IP again.

The problem with this approach of stacking labels is that it is not supported in SDN. In SDN, stacked labels are not supported and routing decisions are made by a single lookup in a forwarding table. To solve for this incompatibility, we propose a compatibility mechanism in the control plane

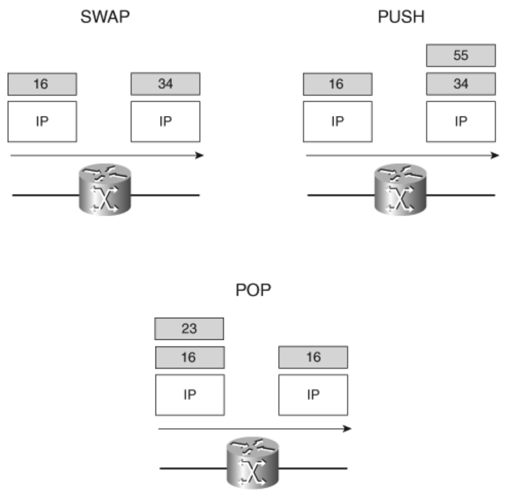

Figure 3.3: An illustration of the different label operations supported by MPLS nodes.

that takes in a description of the MPLS rules, creates an internal representation of the network graph and processes it so that in the end, every path through the graph is identified by a single label, and not a stack of labels. We term the graph processing aspect of this as a Label Flattening Operation.

We propose an algorithm based on Depth First Search (DFS). DFS is a well-known algorithm in graph theory for path discovery and tree generation. Label Matching can be generated online based on DFS after user request. Running DFS from source switch on MPLS label switching can generate an acyclic destination tree, in which each path is uniquely defined by a path label. The path label can be an aggregation of all the MPLS labels along the path.

Initially, the source broadcast path discover message of its own ID to all its neighbours. Upon receiving the message, the node goes through all its outgoing edges with corresponding label match and actions. The action can be add, remove or swap. For add and swap action, the node appends the corresponding MPLS label at the end of its discover message (initially sender's ID), and forwards the new message to corresponding edge. For remove action, the node appends a remove label and forwards the message to corresponding edge. DFS algorithm guarantees the outcome path is loop-free and also the most efficient based on some user defined measurements.

The generated path label is unique. Start from the source, any two paths that have the same path label must have the same source ID, since the source

ID appends at the beginning of path label. Note that each label uniquely defines an outgoing edge from the source, thus any path begins with the label must go through the same edge. Then, consider any node along the path, any path between the node and the source has unique label. Likewise, any path label between the node and the destination is unique. Because the appending operation is commutative, we can show the uniqueness based on induction above.

MPLS paths can have a loop, in which the packet is sent to the same node, but with different states. For example, a package can be sent to a node for processing and then be sent back. In DFS algorithm, each path stores visited nodes to generate loop-free paths. To support loops in the MPLS path, the discover message contains an additional vector of state variables, containing the state of all visited nodes, set to inactive by in default. The processing nodes can change the state variables into active, such that the path can go back to those visited nodes.

Algorithm 1 Algorithm that flattens the MPLS network graph so that every path in the network is defined by a single label, instead of a stack of labels.

```
1: procedure FlattenLabelStack(G, LabelStack, StartingNode)
2:     CurrentNode = StartingNode
3:     NewLabel = NewUniqueLabel()
4:     while LabelStack is not empty do
5:         LabelStack = CurrentNode.performLabelActions(LabelStack)
6:         NextHop = CurrentNode.lookupLBIF(LabelStack.top())
7:         CurrentNode.updateLBIF(NewLabel, NextHop)
8:         CurrentNode = NextHop
```

The pseudocode for our proposed algorithm given in Algorithm 1. The procedure *FlattenLabelStack* takes the label stack and the ingress router responsible for appending it on a message, and maps the label stack to a new singular label. The algorithm then traverses the graph using the label stack and the predefined MPLS rules to trace the unique path identified by the label stack. As it traces the unique path, the algorithm also updates the Label Forwarding Information Base with the new label so that by the end, the new label and the label stack will route through the same path in the core network.

The runtime of our proposed solution depends on the number of unique label stacks assigned by the ingress routers U and the total number of nodes in the system N. For now, one major restriction in our algorithm is that the MPLS graph is assumed not to have cycles. In other words, it is assumed that a packet will never visit the same node again. Therefore, the runtime to flatten a label will, in the worst case, take $O(n)$. Given that we have to flatten U label stacks, the total runtime is $O(UN)$. However, this rule is unrealistic and often packets are routed to the same node, albeit with a different label stack. Therefore, future iterations of our algorithm will seek to generalize the algorithm to handle such cycles in the node.

The algorithm can be best illustrated by an example. For a simple graph comprising of 6 nodes, the results of the algorithm processing can be seen. Note, the nodes shapped as double circles are ingress and egress nodes, while the rest are nodes in the core network. The ingress nodes are responsible for assigning labels, while the labels are all popped in the egress nodes and the packet is forwarded based on IP beyond them. The original graph is shown in Figure 3.4. At every node, the original graph excepts a stack of labels $x : xs$, where x is the top label in the stack and xs are the tail elements. Based on the top label, the node performs a combination of the predefined actions on the label stack. In this particular example, the node 4 performs a pop operation, thereby removing the top label in the stack. Once the operations are performed, the node uses the top label in the new stack to determine the next hop switch.

The new graph is illustrated in Figure 3.5. In this graph, the ingress router assigns only a single label to each packet. There is no stacking of labels and no need for label operations at each node. Rather, the packet is routed throughout the core, from the ingress to the egress, using only the single label that was assigned to it by the ingress router.

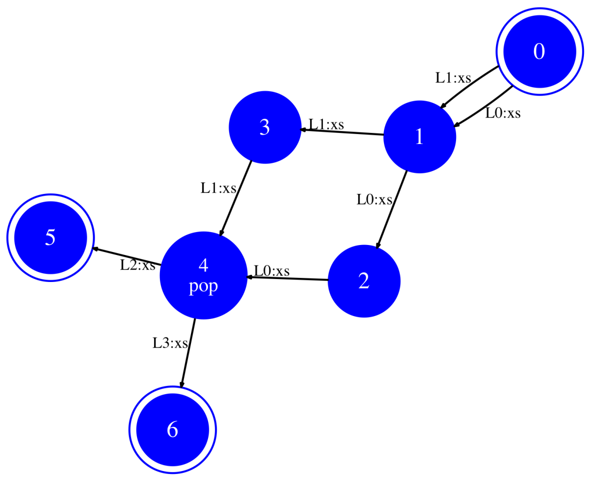

Figure 3.4: The original network graph. Here, routing is based on the top of the stack label and stack operations like push, pop and swap can be performed at every node.

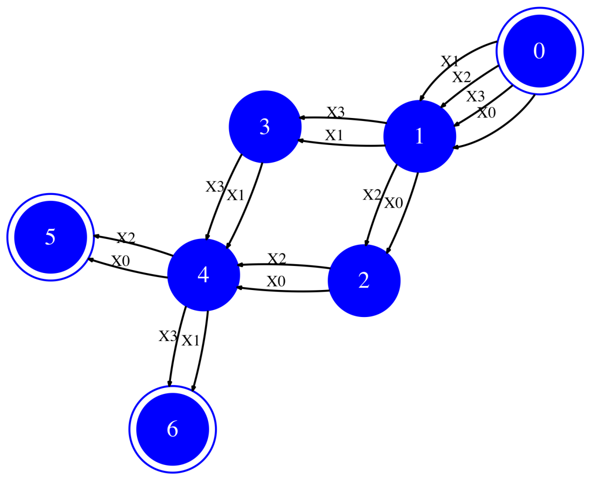

Figure 3.5: The modified network graph. Here, routing is based on a single label and no stack operations are supported.

Chapter 4

Evaluations

In this section, we describe how we evaluated both our implementation as well as the underlying concepts.

4.1 Dataset

To evaluate our implementation, we wished to examine how it affected actual production data center traffic. In order to do this, we used a set of packet traces made available by Benson, et al., from their study of network traffic characteristics in production data centers [17]. [1] The raw data that they gathered is available at [18]. The specific packet traces we used came from both the UNI1 and UNI2 datasets.

The packet capture has traffic from 500 servers, as well as several thousand external hosts. All packets were captured from a single span port.

4.1.1 Flow Data

Flow Collection

While they provide a various forms of fascinating data, we were most interested in observing flow-level behavior. That is, we look specific "conversations" between network hosts/ports.

While basic flows were not readily available, we were able to extract that information from the packet capture file. To do this, we wrote a Python script, using Scapy [19]. For every packet, we extracted the specific information we

[1] This is a very popular dataset when needing to simulate data center network traffic. As of late-April, 2015, their publication has over 500 citations on Google Scholar.

needed (i.e., timestamp, source MAC/IP/port, destination MAC/IP/port, TCP or UDP, and packet size), and compiled these into CSV files.

Isolation of Local Traffic

One particular challenge we had was that there were nearly 8,000 unique IP addresses represented in the dataset, but we were only able to simulate approximately 1,000-2,000 hosts. To resolve this, we chose to focus specifically on local-only traffic. However, there was no information indicating which IP addresses were internal and which were not, nor did we have access to any subnetting information.

So, to identify the local traffic, we considered that the vast majority of non-local traffic entering the network would come for a relatively few number of gateways. After analyzing the flow data we had extracted, we found this intuition to be accurate: Two MAC addresses on the network were associated with 6,723 of the 7,876 IP addresses on the network. By eliminating the flows to or from these IP addresses, our simulation needed to only consider a much more manageable 1,153 hosts.

While it is not specifically addressed by the authors, we presume that the discrepancy between the number of servers known to be present on the network (500) and the number of local hosts we identified (1,153) are due to virtual machines being hosted on the physical servers.

4.1.2 Topology

The switching topology, as shown in Figure 4.1, was made available with the packet traces. We used this topology information to reconstruct a mininet network of virtual switches.

Information about how hosts were connected to the network, and thus the full host topology, were not made available. We worked around this by randomly assigning hosts to edge switches. The full network topology, including hosts, is shown in Figure 4.2.

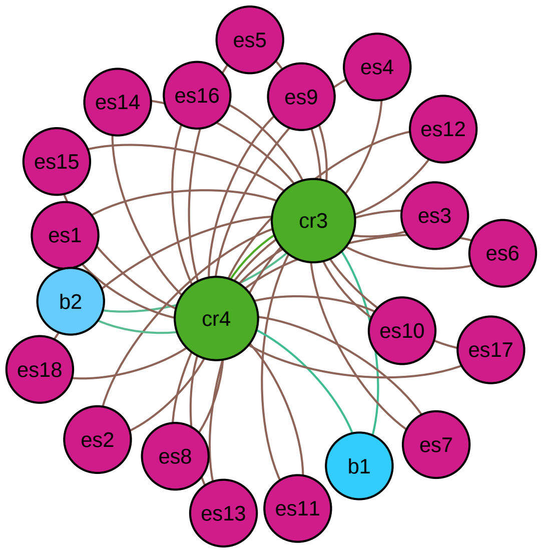

Figure 4.1: Switch topology. Core devices are represented by nodes beginning with `cr` indicate, edge switches are represented by nodes beginning with `es`, and gateways are represented by nodes beginning with `b`.

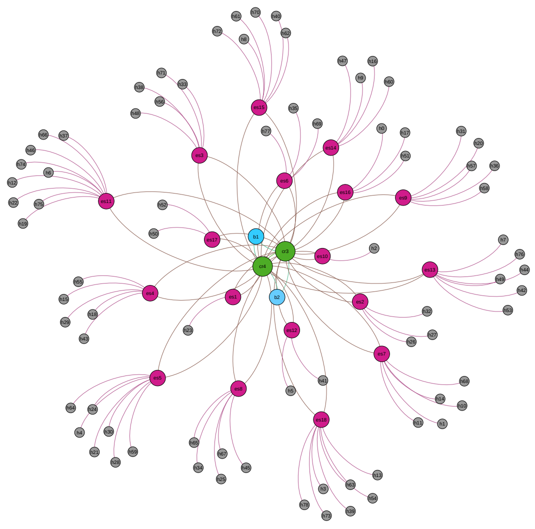

Figure 4.2: Full topology, including network hosts.

4.2 Experiments

4.2.1 Rule Distribution Requirements

Our initial experiments were designed to understand how label-matching rules would be distributed throughout the network, and how the maximum size of the flow table will affect the operation of the network.

Simulator Design

To conduct this experiment, we implemented a very simple simulator that allows us to keep track of the location of all packets, as well as the statistics on each switch, at any given moment over the course of the simulation.

Our simulator takes, as input a list, of "packets", in the form $(timestamp, label)$, and a list of rules, with the form $(label \rightarrow switch_1, switch_2, ..., switch_n)$, indicating the path that packet with a given label would take through the network.

The simulator then steps through the "execution" of the network in configurable time units. These units correspond how long a switch takes to process and forward a packet. At each time unit, if a new packet's timestamp has been reached, it is placed at its initial switch, according to its label. If the switch has a rule associated with the packet's label, it will be allowed to move forward to its next switch on the next time step. However, if the switch does *not* have a matching rule, and additional latency is introduced before the packet is able to move forward; this simulates the required for the switch to request the rule from the controller, and for the controller to respond.

In order to make the simulation more realistic, we enforce a hard limit on the number of flow rules that can be installed on a switch at any given time. When a flow rule must be added to an already full flow table, some eviction policy must be enacted. For this experiment, we simply evicted a random rule.

After all packets have made their way from source to destination, the simulator reports how many packets each switch processed, and how many of them required a request to the controller.

Experimental Design

We ran this experiment with a subset of our data that amounted to 966,000 packets with 404 unique labels over a 110-second timespan. The rule list was generated using an LDP-based scheme. The simulator was run over 100 iterations with the maximum number of rules allowed to be in a switch's flow table as the only variable. When a switch's flow table was filled, we used a random eviction policy to make room for the next flow rule.

Results

The results from this experiment were pleasantly surprising. We found that, as expected, a smaller flow table resulted in high-levels of "churn": the table would quickly fill, and, as rules that had been evicted to make room were eventually required again, the number of "flow misses" grew very high. This is particularly true of higher-volume switches. For example, Table 4.1, which displays the number of flow misses at each switch for selected iterations of the experiment. Consider Switch 2, which represents one of the core switches in the topology, as seen in Figure 4.1, and every packet in the simulation is forwarded by it.[2] Since Switch 2 is responsible for every packet that crosses the network, it has by far the highest churn rate.

What is surprising, however, is how few entries were actually required to dramatically decrease the number of flow misses. This is demonstrated in Table 4.1, but more effectively in Figure 4.3, which displays the average hit rate (i.e., $1 - miss\_rate$) for every class of device for each iteration. [3] The edge switches and gateways achieve a nearly 100% hit rate with only a few flow entries. Even Switch 2 achieves a nearly 100% hit rate at around 50 flow entries.

This is a promising sign for the long-term viability of our solution. It also, perhaps, casts doubt on our initial reasoning for choosing to implement label matching in the MAC tables of core switches, instead of the more traditional TCAM, as was discussed in Section 3.3.

[2]It should be noted that Switch 3 represents the other core switch, but due to the simple nature of the algorithm that set paths through the network, no packets ever traverse it.

[3]Switch 3 is excluded from the core switch calculation, as it is effectively *not* in the network.

Switch	Flow Table Size (number of entries)							
ID	5	15	25	35	45	55	65	75
0	125756	36	36	36	36	36	36	36
1	10	11	11	11	11	11	11	11
2	646061	307660	157911	81297	36446	7169	97	97
3	0	0	0	0	0	0	0	0
4	1	1	1	1	1	1	1	1
5	4	4	4	4	4	4	4	4
6	24377	20	20	20	20	20	20	20
7	54666	18	18	18	18	18	18	18
8	4	4	4	4	4	4	4	4
9	1846	7	7	7	7	7	7	7
10	17	17	17	17	17	17	17	17
11	2	2	2	2	2	2	2	2
12	6	6	6	6	6	6	6	6
13	10633	13	13	13	13	13	13	13
14	3	3	3	3	3	3	3	3
15	3	3	3	3	3	3	3	3
16	16993	13	13	13	13	13	13	13
17	5602	11	11	11	11	11	11	11
18	2	2	2	2	2	2	2	2
19	15217	15	15	15	15	15	15	15
20	7	7	7	7	7	7	7	7
21	4373	9	9	9	9	9	9	9
Average	39373	13385	6875	3544	1595	322	15	16

Table 4.1: This table shows the number of total flow misses at each switch, given the size of the flow table.

4.2.2 Number of Rules

Another set of experiments that we conducted were designed to determine how many rules were present in the network, as a function of the number of hosts visible to the network. This experiment was conducted in mininet with a the full dataset being used.

Figure 4.4 depicts the results when the rules are generated using the LDP scheme, and Figure 4.5 depicts the results when the rules are generated using RSVP-TE. In both cases, what we find is that the number of rules in both the core and gateway switches remain constant, while the number of rules in the edge switches grow linearly with the number of hosts.[4]

[4]Note: The x-axes of both graphs are on a log_{10} scale.

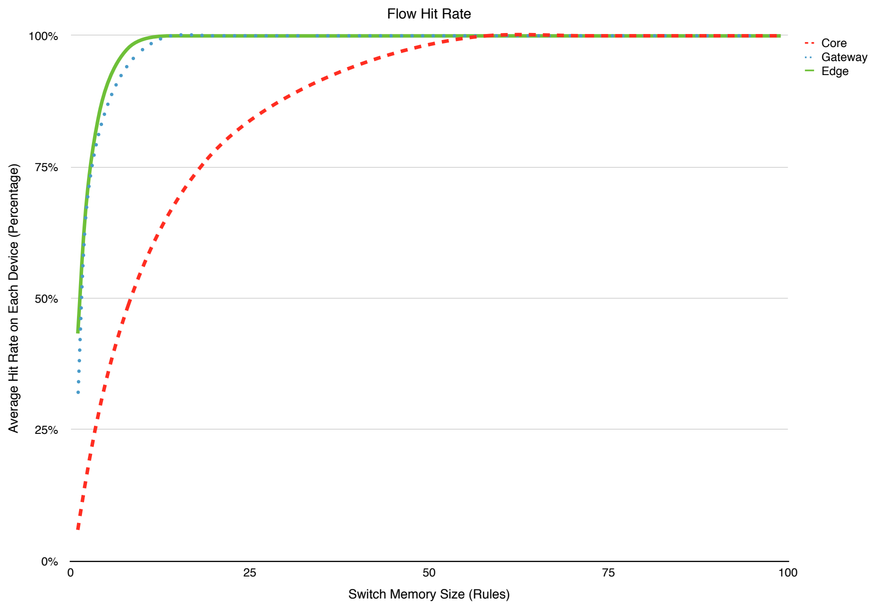

Figure 4.3: Flow rule hit rates quickly approach 100% as the number of flow entries available to a switch increases.

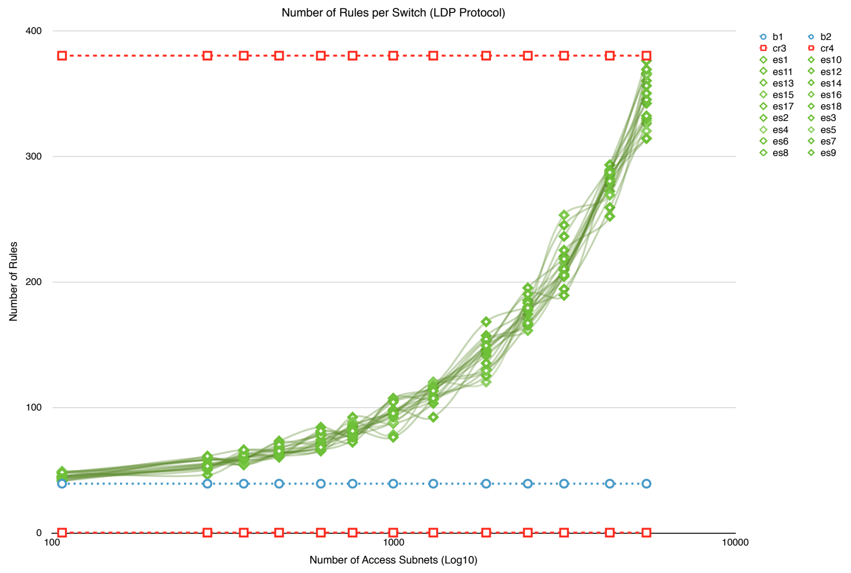

Figure 4.4: Number of rules in each switch as a function of access subnets present, when using LDP-based distribution model.

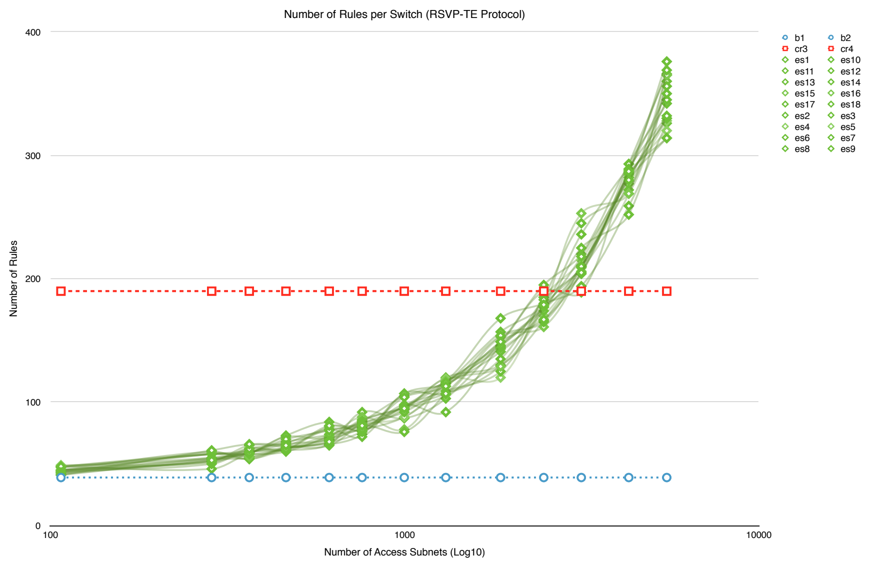

Figure 4.5: Number of rules in each switch as a function of access subnets present, when using RSVP-based distribution model.

Chapter 5

Conclusion

This thesis has described out implementation of MPLS functionality in an OpenFlow controlled network, capable of running in on simple commodity hardware. We accomplish this by mapping the MPLS labels that would be pushed into packets down to a single static label that is stored in the packet's destination MAC address field remains unaltered as the packet travels between edges of the network.

We were inspired to pursue this goal by the Fabric paper by Casado, et al [1]. While not fully achieving the design that they outlined, our work demonstrates a basic framework for achieving similar designs without requiring a complete up-ending of the existing technology.

The results from our evaluations are very promising, demonstrating that we are able to efficiently forward large amounts of traffic through the core network with only modest hardware, in terms of available functionality. In fact, we saw such a diminishing amount of flow churn with only a marginal increase in available flow rules, that we may be forced to rethink one of our basic assumptions: TCAM would be too expensive to be used efficiently in the core network. We have seen, however, that even a relatively modest number of available flow entries were able to achieve a near 100% hit rate.

References

[1] M. Casado, T. Koponen, S. Shenker, and A. Tootoonchian, "Fabric: a retrospective on evolving sdn," in *Proceedings of the first workshop on Hot topics in software defined networks*. ACM, 2012, pp. 85–90.

[2] D. Kreutz, F. M. Ramos, P. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[3] 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*. ACM, 2010, p. 19.

[4] S. Ghorbani and B. Godfrey, "Towards correct network virtualization," in *Proceedings of the third workshop on Hot topics in software defined networking*. ACM, 2014, pp. 109–114.

[5] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow et al., "Onos: towards an open, distributed sdn os," in *Proceedings of the third workshop on Hot topics in software defined networking*. ACM, 2014, pp. 1–6.

[6] H. Mekky, F. Hao, S. Mukherjee, Z.-L. Zhang, and T. Lakshman, "Application-aware data plane processing in sdn," in *Proceedings of the third workshop on Hot topics in software defined networking*. ACM, 2014, pp. 13–18.

[7] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[8] 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," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 3–14.

[9] C. Systems, “Quality of service networking,” 2012. [Online]. Available: http://docwiki.cisco.com/wiki/Quality_of_Service_Networking

[10] J. Networks, “Multiprotocol label switching enhancing routing in the new public network.”

[11] J. T. Johnson, “Mpls explained,” 2007. [Online]. Available: http://www.networkworld.com/article/2297171/network-security/mpls-explained.html

[12] J. Kempf, S. Whyte, J. Ellithorpe, P. Kazemian, M. Haitjema, N. Beheshti, S. Stuart, and H. Green, “Openflow mpls and the open source label switched router,” in *Proceedings of the 23rd International Teletraffic Congress*. International Teletraffic Congress, 2011, pp. 8–14.

[13] S. Azodolmolky, R. Nejabati, E. Escalona, R. Jayakumar, N. Efstathiou, and D. Simeonidou, “Integrated openflow–gmpls control plane: an overlay model for software defined packet over optical networks,” *Optics express*, vol. 19, no. 26, pp. B421–B428, 2011.

[14] K. Agarwal, C. Dixon, E. Rozner, and J. Carter, “Shadow macs: scalable label-switching for commodity ethernet,” in *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking, ser. HotSDN*, vol. 14, 2014, pp. 157–162.

[15] A. Schwabe and H. Karl, “Using mac addresses as efficient routing labels in data centers,” in *Proceedings of the third workshop on Hot topics in software defined networking*. ACM, 2014, pp. 115–120.

[16] *OpenFlow Switch Specification Version 1.1*, Open Networking Foundation Std. [Online]. Available: http://archive.openflow.org/documents/openflow-spec-v1.1.0.pdf

[17] T. Benson, A. Akella, and D. A. Maltz, “Network traffic characteristics of data centers in the wild,” in *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*. ACM, 2010, pp. 267–280.

[18] T. Benson, “Quality of service networking.”

[19] P. Biondi, 2014. [Online]. Available: http://bb.secdev.org/scapy/wiki/Home