ROMA TRE
UNIVERSITÀ DEGLI STUDI

Roma Tre University
Ph.D. in Computer Science and Engineering

Improving flexibility, provisioning, and manageability in intra-domain networks

Gabriele Lospoto

Cycle XXVIII

Improving flexibility, provisioning, and manageability in intra-domain networks

A thesis presented by
Gabriele Lospoto
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
in Computer Science and Engineering

Roma Tre University
Department of Engineering

Spring 2016

COMMITTEE:
Prof. Giuseppe Di Battista

REVIEWERS:
Prof. Olivier Bonaventure, Université Catholique de Louvain
Prof. Stefano Giordano, Università di Pisa

To my family

Allora Gesù disse loro: "Gettate la rete dalla parte destra della barca e troverete". La gettarono e non potevano più tirarla su per la gran quantità di pesci. Allora Simon Pietro salì nella barca e trasse a terra la rete piena di centocinquantatrè grossi pesci. E benché fossero tanti, la rete non si spezzò.

(Gv 21, 6;11)

So Jesus said to them, "Cast the net over the right side of the boat and you will find something." So they cast it, and were not able to pull it in because of the number of fish. So Simon Peter went over and dragged the net ashore full of one hundred fifty-three large fish. Even though there were so many, the net was not torn.

(John 21, 6;11)

Acknowledgments

First of all, I would like to sincerely acknowledge my advisor, Giuseppe Di Battista: he gave me the opportunity to enter in the research world. He was more than an advisor: he always passed down to me all his passion and his deep knowledge about this work. I will always be grateful for giving me the opportunity to grow up, as a man as well as a researcher.

A wholehearted thank goes to Massimo "Max" Rimondini: he has been my second advisor, and he always gave me valuable advices. Working with him was always a great pleasure, as well as a honor. He was also a great friend, and I always had the chance to count in him during these years and I am also sure I will have the opportunity to count in him in the future.

A special thank goes to Marco Chiesa. When I worked on my master thesis, he was a PhD student and I strongly collaborated with him in that period. During our small talks, he told me about the PhD student's life with a contagious enthusiasm that significantly contributed to make me choose to become a PhD student.

I would like to thank Stefano Vissicchio, whom I worked with during my stay at UCL. I had the opportunity to appreciate that details make research higher.

I would like to thank Professor Olivier Bonaventure and Professor Stefano Giordano for carefully reviewing this thesis: I really appreciated their suggestions.

A "smoky" thank to Roberto di Lallo and Vincenzo Roselli for sharing very productive breaks in the terrace.

I would like to thank all people who have been part of our research group, making our lab enjoyable. I could just say "Angelini *et al.*", but this time I prefer to mention each of them: Patrizio Angelini, Massimo Candela, Marco Di Bartolomeo, Giordano Da Lozzo, Valentino Di Donato, Fabrizio Frati, Federico Griscioli, Luca Lanziani, Habib Mostafaei, Alessandro Manfredi, Alessandro Mancini, Andrea Olivieri, Maurizio Patrignani, Maurizio Pizzonia, Claudio Squarcella.

A very brotherly thank to fr. Paolo. Walking beside me, he showed me that nothing is impossible: having the bravery to choose is enough.

Finally, but not less important, I would like to thank all my family for their tireless support. There have been hards years, but they constantly incited me to not lose the hope! For this, and many other reasons, thanks to my dad Gianni, my mom Patrizia, my sister Francesca, and my (almost) brother-in-law Vincenzo.

Contents

Introduction

Internet has a hierarchical structure, in which network devices (e.g., routers) are grouped into logical areas. Each area is called Autonomous System (AS) and it is typically under the control of a single organization, also called Internet Service Provider (ISP). An ISP takes decisions in terms of routing protocols to use for forwarding traffic. The focus of this thesis is on problems inside the network of an ISP, a context that is commonly referred to as intra-domain network. Routing inside intra-domain networks is typically realized combining several protocols, which collaborate in order to provide services, even basic like connectivity. This makes protocols specialized on a specific task (e.g., guarantee certain levels of quality of service) giving to network administrators the possibility to choose among different protocols in order to accomplish a specific activity (e.g., computing paths in the network). Such coexistence surely aims at improving flexibility, but it also leads to several problems in terms of manageability of the network, making the provisioning of services more difficult. The main goal for an ISP consists in having a fine-grade control over the routing paths in the network, which gives to the ISP itself the opportunity of exploiting its network efficiently; at the same time, making the management of the network simpler, for example reducing both the configuration and troubleshooting effort, is also a desirable target.

In this thesis, four different contributions are presented, aimed at: 1) making routing flexible; 2) simplifying the provisioning of the services; 3) making the manageability of the network sustainable for network administrators. The emphasis is on how flexibility and manageability can be improved using both distributed and centralized approaches. On the other hand, provisioning can be significantly improved by using a centralized paradigm, like Software-Defined Networking (SDN): having a single logic place in which to locate the entire control for a service allows network administrators to also decrease the configuration effort in the setup of that service, as well as to have very high levels of flexibility.

Adopting the novel approaches proposed by SDN raises new challenges: after showing the benefits brought by it, interoperability issues are addressed, as well as a review of the most relevant state of the art contributions on SDN, checking whether they are applicable on real OpenFlow-enabled switches.

The rest of the thesis is organized as follows. The chapters from 2 to 4 describe respectively three contributions aimed at making the routing as flexible as possible, as well as reducing configuration and troubleshooting problems; chapter 5 describes contribution in terms of how the most relevant SDN state of the art is not applicable on current OpenFlow-enabled switches.

The first contribution is referred to a distributed control plane aiming to improve flexibility and manageability. This control plane relies on approaches like multipath and source routing in order to provide to network administrators tools that make the manageability of the network simpler; moreover, exploiting multipath and source routing, the control plane also improves flexibility in the network.

The second contribution is a solution aiming at improving provisioning, as well as flexibility and manageability. The idea behind this contribution is the following: can OpenFlow, the most used protocol enabling SDN, replace many protocols in order to provide a service? Starting from a well known and widely used service in production networks, Virtual Private Networks, a *rethinking* activity has been performed, in order to make its provisioning and manageability faster and simpler, by reducing the number of protocols involved in setting up this service and, consequently, the configuration effort. Indeed, setting up a Virtual Private Network is not trivial: many protocols are typically involved, each of which has a specific task; being flexible is not simple as well, because the interaction among protocols is realized by tuning some configuration parameters; also the management of the service itself is hard, because the configuration of a Virtual Private Network is scattered among many devices; this makes troubleshooting hard as well. As a consequence of using just one protocol based on a centralized approach like OpenFlow, and observing that it typically has a complete view of the network, a new centralized specification language for setting up Virtual Private Network is also defined.

The third contribution addresses an interoperability problem. Indeed, today's networks strongly rely on the Address Resolution Protocol (ARP), a protocol for building the association between IP and MAC addresses; this protocol is also executed by end-systems, that do not play any active role in the operation of an SDN, like computers; hence an interoperability mechanism is needed, allowing OpenFlow-enabled switches to correctly handle ARP packets. This protocol generates broadcast traffic that typically traverses local networks: the simpler and most adopted solution for handling this kind of traffic using OpenFlow is to re-implement the ARP protocol behavior. However, this choice does not take into account the power of SDN. Exploiting the SDN approach, a solution that prevents broadcast ARP packets from passing through the local SDN-enabled networks has been proposed; in particular, these packets can be bounded at the edge of the network, making the handling of ARP traffic more efficient.

The fourth contribution is an investigation on the readiness of real devices to run centralized protocols like OpenFlow. The idea is very simple: OpenFlow 1.0.0 is the first standard, and it was released in 2009; the most recent, OpenFlow 1.5.1, has been released around the half of 2015; is the SDN state of the art applicable using real devices? Moreover, which is the level of compliance of the real network devices with the OpenFlow standard after six years? The former question arises because many evaluations and experiments of SDN papers have been conducted in a simulated environment. We reviewed the most relevant state of the art talking about SDN, aiming at understanding if they are applicable. We also investigated features in terms of functions and performance of those devices.

In the last chapter conclusions and research directions will be reported.

Chapter 1

Routing in Intra-Domain Networks

In this chapter we introduce some basic concepts, aiming to provide preliminary building blocks, useful to have a more complete view over the addressed problems and the proposed solutions. First of all, we propose a brief overview about different approaches which routing protocols rely on; after that, an introduction on Software-Defined Networking (SDN) and OpenFlow specification [1] is proposed.

1.1 Routing Protocol Approaches

Routing protocols may rely on two different and complementary approaches: distributed and centralized. A routing protocol based on a distributed approach typically works as follows. Each network device cooperates with the other ones in order to produce, accordingly with a certain algorithm, a set of rules, allowing traffic to be forwarded in the network. A network device (e.g., a router) with a distributed routing protocol on board has two logical levels: *control plane* and *data plane*. The first one implements algorithm used to compute paths in the network; it can be thought as the *brain* of the network itself. The data plane has in charge the task of forwarding traffic relying on the rules computed by the control plane. All today’s routing protocols are based on the distributed approach: an example is OSPF [2], the most used protocol in intra-domain networks. These protocols are massively used because they are fast to recover from network changes, guaranteeing very high levels of robustness in the network. A network administrator interacts with these protocols through configuration files, that allows him/her to tune several parameters in order to affect the behavior of the control plane. Unfortunately, this interaction system, joint to the coexistence of many protocols in today's network, amplifies problems like flexibility, provisioning and manageability.

Flexibility problems are referred to the complexity in taking the control of how traffic is forwarded. Indeed, paths used to carry traffic in the network are computed by algorithms that are typically built-in inside the firmware of devices, making difficult to implement custom forwarding policies. The interactions with routing protocols are always realized interacting with configuration files: exploiting them, a network administrator can influence the behavior of the protocols; nevertheless he/she will not be able to exactly determine how traffic will be forwarded, leaving to the protocol itself the task of computing paths accordingly to the strategies implemented in the algorithms.

Provisioning problems arise when a network administrator wants to set up a service. Provisioning is referred to the ability of configuring services in a simple and fast way. In a tradtional network a network administrator must interact with different protocols, potentially replicating the configuration on many devices. This task is time consuming and the configuration effort is not negligible. For the same reasons, also *manageability* becomes a relevant problem: in presence of a misbehavior in the network, investigating the possible causes could not be a trivial activity. Moreover, modifying the configuration for a service may result hard to perform.

An alternative approach to the distributed one is the centralized. This approach relies on a single entity having in charge the task of controlling the whole network, and instructing each device about the forwarding. This approach has been set aside for a long time, but in the past years it is back under the name of Software-Defined Networking (SDN).

1.2 Software-Defined Networking

Software-Defined Networking (SDN) is routing architecture [3] based on a centralized routing paradigm that is collecting the attention of the entire network research community, as well as many vendors. In this approach, software implementing the control plane, called *controller*, is moved into a dedicate hardware (e.g., a computer), leaving on board of the network devices the data plane level. A controller interacts with devices through an ad-hoc protocol: *OpenFlow* [1] is the most used protocol enabling SDN in the networks. A controller has in charge many tasks; it must: i) receive topology and state information from all network devices and reconstruct the entire topology;

ii) compute routing tables for each network device in the network in order to forward traffic; iii) distribute them to each network device.

Centralized approach offered by SDN potentially has several advantages: first of all, it allows network administrators to precisely determine how forwarding is realized: in fact, SDN gives the opportunity to write software in order to model the behavior of the network, instead of interacting with many configuration files. Moreover, exploiting the centralized network’s view at the controller, the configuration is not replicated on many devices, for example reducing the effort of troubleshooting activities. Unfortunately, SDN also brings some disadvantages, especially in terms of robustness. Indeed a new relevant research field is arising: Hybrid SDN [4, 5, 6], namely the possibility to let traditional distributed routing protocols and SDN coexist in the same network. For example in [4] authors aim to take advantage from both the flexibility offered by SDN and the robustness abilities of the distributed protocols, pointing out how a centralized decision point is a profitable choice for having a flexible control over the network; in this thesis it is also argued that a central decision point improves the provisioning of the services and the manageability of them, and – consequently – of the network.

The OpenFlow Protocol

OpenFlow is a specification of a logical architecture for an SDN-enabled switch (*datapath*) and of a protocol for the communication between such a switch and a controller platform. It is by far the most widely adopted specification, to the point that even vendors that developed alternative implementations of SDN customized to support proprietary functions (e.g., Cisco’s onePK [7]) also offer OpenFlow support as a compatibility plug-in.

In this section we summarize the fundamental elements of the OpenFlow specification that are useful to understand our device testing methodology. Several versions of the specification have been published since its appearance in 2009, confirming that it has now reached a considerable level of maturity: in this summary we refer to the most recent version, 1.5.1 [8]. The specification describes three key concepts: datapath ports, various kinds of tables, and the datapath-controller communication protocol.

The configuration of a datapath often includes a declaration of the physical ports that operate in OpenFlow mode, namely that are part of an instance of (virtual) OpenFlow datapath. According to the specification, at least two kinds of ports are exposed to an OpenFlow datapath instance: an abstraction of each physical port where the port number, its features, and its status can be accessed via OpenFlow data structures and messages; and a set of reserved ports, that are used to accomplish special actions or invoke OpenFlow-specific functionalities. Support for some of the reserved ports is mandatory: for example, this is the case for ports ALL (used to forward a copy of a packet on all the interfaces but the one through which it was received) and CONTROLLER (used to send a packet to the controller). Support for other reserved ports is optional: for example, this applies to the NORMAL port.

According to the specification, an OpenFlow datapath must implement different kinds of tables: the standard flow tables, a group table, and a meter table. First of all, it is possible to apply an arbitrary bitmask to certain packet headers to match only a subset of the bits of a field value. This is particularly useful, for example, when matching IP subnets. Moreover, among the actions declared as mandatory by the specification, there is a "group action", which allows to perform several actions on multiple copies of the same packet. Match conditions and actions can also operate on registers, called "metadata", that are used to pass information between flow tables. Flow entries have a priority, and every flow table also has a lowest-priority special *table-miss* flow entry, which determines the action that the datapath should undertake on packets that were not matched by any of the entries in the flow table (in the absence of a table-miss flow entry, packets should just be dropped). Each entry in the flow table may have counters that determine how many packets and bytes matched that entry: note that the support of these counters is declared as optional. Besides the flow tables, an OpenFlow datapath also maintains a *group table*, whose implementation is mandatory. This table is used to store groups of actions that can be referenced in the action part of a flow entry. Depending on the type of the group, all or only one the involved actions are executed on matching packets. Groups are therefore used during our tests to check the ability of a datapath to forward copies of a packet out of multiple ports. Finally, an OpenFlow datapath also maintains a mandatory *meter table*, that defines per-flow meters usable for classifying, rate limiting, or dropping different types of traffic.

Concerning datapath-controller communication, the forwarding is accomplished according to match-action rules that the controller installs in *flow tables* on the datapaths using FlowMod messages, either proactively (before any traffic is exchanged) or reactively (after a PacketIn message is received from a datapath that has no flow entries to handle a specific packet). If required, the controller may also ask a datapath to originate a packet, using a PacketOut message.

Chapter 2

Intra-Domain Routing with Pathlets *

Internal routing inside the network of an Internet Service Provider (ISP) affects the performance of lots of services that the ISP offers to its customers and is therefore critical to adhere to Service Level Agreements (SLAs), achieve a top-quality offer, and earn revenue. Existing technologies (most notably, MPLS) offer limited (e.g., with RSVP-TE), tricky (e.g., with OSPF metrics), or no control on internal routing paths. Recent research results address these shortcomings, but miss a few elements that would enable their application in an ISP's network.

We introduce a new hierarchical control plane, based on pathlet routing [9], designed to operate in the network of an ISP and offering several nice features: it enables steering of network paths at different levels of granularity, allowing the ISP to have a flexible handling of those paths, since our control plane relies on a multi-path approach; it is scalable and robust; it supports independent configuration of specific network regions and differentiation of Quality of Service (QoS) levels; it can nicely coexist with other control planes and is independent of the data plane used in the ISP's network. Besides formally introducing the messages and algorithms of our control plane, we propose an experimental scalability assessment and comparison with OSPF, conducted in the simulation framework OMNeT++.

*Part of the material presented in this chapter is based on the following publication: M. Chiesa, G. Lospoto, M. Rimondini, G. Di Battista. Intra-Domain Routing with Pathlets. In *Computer Communications*, 46:76-86. 2014.

2.1 Introduction

In a never-ending effort to offer top-quality services, Internet Service Providers (ISPs) strive to distribute traffic loads in their networks with clever strategies that not only ensure satisfaction of Service Level Agreements (SLAs), but also realize competitive performance levels that earn them market shares and, therefore, revenue. Fine-grained control of internal routing paths is essential to achieve these goals, and several technologies (e.g., OSPF, RSVP) have been introduced and widely deployed to gain this control. However, their complexity of setup, scarce predictability of dynamic behavior, and limited degree of control of routing paths pushed the research community to seek for alternative solutions along approaches like source routing, multipath routing, and hierarchical routing. To the extent of our knowledge, none of these solutions yet succeeded in combining these approaches to obtain routing control while supporting other features that ISPs yearn for, like configuration simplicity, robustness, compatibility with deployed routing mechanisms, and Quality of Service (QoS) differentiation, to mention a few.

In this chapter we propose the design of a new control plane for internal routing in an ISP's network which aims at achieving these goals. Our control plane supports control of routing paths at different levels of granularity, envisions several kinds of routing policies, and allows computation of multiple paths for resilience and, possibly, QoS differentiation. It reacts efficiently to topological changes and administrative reconfigurations, enables administrators to independently configure different network portions, and it can be incrementally deployed. We build our control plane on top of pathlet routing [9], one of the most convenient approaches introduced so far to tackle an ISP's requirements. By integrating this contribution and combining it with other suitably adapted approaches from the literature, we define a complete pathlet-based routing solution that is applicable to intra-domain routing, filling a gap that, as far as we know, is still open.

In the control plane we propose, routers exchange path fragments called *pathlets* and are grouped into *areas*: within a single area routers exchange all information about the available links, in a much similar way to what a link-state routing protocol does; when announced outside the area, such information is summarized in a single pathlet that goes from an entry router for the area directly to an exit router, without revealing routing choices performed by routers that are internal to the area. This special pathlet, which we call *crossing pathlet*, is considered outside the area as if it were a single link. An area can enclose other areas, thus forming a hierarchical structure with an arbitrary number of levels.

The rest of the chapter is organized as follows. In Section 2.2 we review the state of the art on routing mechanisms that could match the requirements of ISPs. In Section 2.3 we introduce a model for a network where nodes are grouped in a hierarchy of areas. Based on this model, in Section 2.4 we define the mechanisms for the creation and dissemination of pathlets in the network. We then describe in Section 2.5 how network dynamics are handled, including the specification of the messages of our control plane and of the algorithms executed by network nodes to update routing information. In Section 3.3 we elaborate on the practical applicability of our control plane in an ISP's network in terms of possible deployment technologies and illustrate an incremental deployment scenario. In Section 2.7 we present an experimental assessment of the scalability of our approach and compare its performance with those of OSPF, using the OMNeT++ simulation framework. Conclusions and plan for future work are presented in Section 2.8.

2.2 Related Work

Many contributions in the literature propose methodologies, algorithms, and protocols that address the scalability, robustness, and controllability requirements faced by an ISP in managing its network. Commonly adopted approaches to satisfy these requirements include source routing, hierarchical routing, and multipath routing. For example, hierarchical routing has for long been known to be provably effective in reducing the size of routing tables [10]. On the other hand, multipath routing is widely used in sensor networks [11], where reachability of the various nodes must be guaranteed even under frequent connectivity variations.

However, none of the contributions we are aware of succeeds in proposing a complete routing solution that fits the requirements of an ISP in an intra-domain scenario: either they apply to inter-domain routing, where the degree of control offered by the available technologies, as well as the goals that ISPs are interested in pursuing, are different than those already proposed in the literature, or they fail to address some basic requirements, most notably simplicity of setup or compatibility with already deployed configurations and technologies. We now review the state of the art on the most relevant control plane mechanisms, using Table 2.1 as a reading key to classify the contributions we mention.

In terms of technologies, OSPF [2] is the state of the art for interior routing

"main" — 2016/5/6 — 12:57 — page 12 — #22

	Source routing	**Hierarchical routing**	**Multipath routing**
MIRO [13]	Limited	No	Yes
Path Splicing [14]	Limited	No	Yes
NIRA [15]	Yes	No	No
Landmark [16]	No	Yes	Yes
Slick Packets [17]	Yes	Limited	Yes
BGP Add-Paths [18]	No	No	Yes
YAMR [19]	No	Limited	Yes
HLP [20]	No	Limited	No
ALVA [21]	No	Yes	Limited
MACRO [22]	No	Yes	No
HDP [23]	Limited	Yes	No

Table 2.1: A classification of the state of the art according to the adoption of some relevant routing techniques.

and has a wide deployment base. However, it offers limited control of routing paths, because they can only be affected by assigning costs and it is very hard to influence a single path without affecting others; it imposes restrictions on the configuration of areas, because they must adhere to a precise structure with a single backbone and multiple stubs/transits; it is not designed to support source routing; and it has limited options to handle multiple alternative paths, typically consisting in a set of possible load balancing policies. Although not a true routing protocol, RSVP [12] has been conceived with traffic engineering in mind, and yet it shares many of the shortcomings mentioned for OSPF.

MIRO [13] is a routing solution that supports the negotiation of multiple routing paths to satisfy the diverse requirements of end users, but no complete control can be enforced on these paths. A similar drawback is shared by path splicing [14], a mechanism designed to realize fault tolerance (see also [24]): it exploits multipath routing to ensure connectivity between network nodes as long as the network is not partitioned, but actual routing paths are not exposed and cannot therefore be controlled. The route discovery mechanism envisioned in NIRA [15] makes routing paths more controllable, but this solution is designed only for an inter-domain routing architecture, like MIRO, and it relies on a constrained address space allocation, a hardly feasible choice for an ISP that is taken also by Landmark [16]. Slick packets [17] achieves a combination of fault tolerance and source routing, obtained by encoding in the forwarded packets a directed acyclic graph of different alternative paths to reach the destination. Besides the intrinsic difficulty of this encoding, this solution inherits the limits of the dissemination mechanisms it relies on: NIRA or pathlet routing (discussed below). BGP Add-Paths [18] and YAMR [19] also address resiliency by announcing multiple routing paths selected according to different criteria, but they only adopt multipath routing, they offer very limited or no support for hierarchical routing, and they have some dependencies on the BGP technology. A completely different approach is taken by HLP [20], which proposes a hybrid routing mechanism based on a combination of link-state and path-vector protocols. In this paper the authors present an in-depth discussion of the routing policies that can be implemented in such a scenario. Although HLP matches more closely our approach, it is not conceived for internal routing in an ISP's network, it constrains the hierarchical network structure to reflect inter-ISP agreements, and it has limits on the configurable routing policies. A similar hybrid routing mechanism called ALVA [21] offers more flexibility but, like Macro-routing [22], it does not explicitly envision source routing and multipath routing. HDP [23] is a variant of this approach that, although natively supporting Quality of Service and traffic engineering objectives, is closely bound to MPLS and accommodates source routing and multipath routing only in the limited extent allowed by this technology.

Some contributions, like LIPSIN [25], adopt a completely different routing approach based on Bloom filters to gain efficiency. However, these solutions are more oriented to multicast forwarding and do not offer a complete control on routing paths because they are based on a probabilistic model. Pathlet routing [9] is definitely the contribution that is closest to our control plane approach, because it introduces a data plane that supports a very flexible handling of routing paths. Its most evident drawback is the lack of a completely defined mechanism for the dissemination of pathlets, which the authors only hint at. Our control plane approach, which is based on pathlet routing, shares some routing principles with those adopted in wireless sensor and mobile networks: among the others, the existence of clusters (which we call areas) and the selection of routing paths based on some quality metrics. However, there are some differences. For example, while energy constraints, handover mechanisms, and evolution of network clusters are not a concern in the scenario we consider, we provide any vertex in the network with enough information to perform source routing, removing the need for central nodes that hold forwarding information (e.g., Cluster Heads).

As a general remark, previous contributions highlight how path-vector protocols typically support complex information hiding and path manipulation policies, whereas link-state protocols typically offer fast convergence with a low overhead. Therefore, a suitable combination of the two mechanisms, which is considered in our approach, should be pursued to inherit the advantages of both.

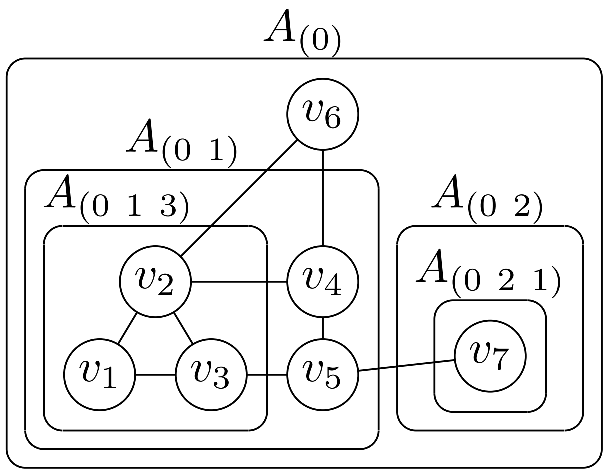
2.3 A Hierarchical Network Model

We now describe the hierarchical model we use to represent the network. In addition, we describe how vertices in the network are assigned label stacks, used to realize a hierarchical network structure, and define on these stacks a few operators that are used to construct and propagate routing data; we then introduce the concept of area and of border vertices, which are in charge of summarizing internal routing information for an area.

We model a network topology as an undirected graph $G = (V, E)$, with vertices in V representing routers and edges in $E = \{(u, v) | u, v \in V\}$ representing links between routers. An example of network is in Figure 2.1. We assume that any vertex in the graph is interested in establishing a path to a set of *destination vertices* $D \subseteq V$ that represent routers that announce network destinations. Note that, although suitable to capture the physical topology of an ISP's network, this representation can also be adopted for overlay networks.

In order to limit the propagation of routing information that is only relevant in certain portions of the network and to improve scalability, we group vertices into structures called areas. Each vertex $v \in V$ is assigned a stack of labels $S(v) = (l_0\ l_1\ \dots\ l_n)$ that are taken from a set L, describing the area that v belongs to. We assume that l_0 is the same for every $S(v)$. An example of assignment of label stacks to vertices, and the corresponding assignment of vertices to areas, is shown in Figure 2.1.

We now define some operations on label stacks that allow us to formally introduce the notion of area and that will be useful in the rest of the chapter. Given two stacks $\sigma_1 = (l_1\ l_2\ \dots\ l_i)$ and $\sigma_2 = (l_{i+1}\ l_{i+2}\ \dots\ l_n)$, we define their concatenation as $\sigma_1 \circ \sigma_2 = (l_1\ l_2\ \dots\ l_i\ l_{i+1}\ l_{i+2}\ \dots\ l_n)$. Assuming that $()$ denotes the empty stack, we have that $\sigma \circ () = () \circ \sigma = \sigma$. We say that a stack σ_2



$L = \{0, 1, 2, 3\}, l_0 = 0$

$S(v_1) = S(v_2) = S(v_3) = (0\ 1\ 3)$
$S(v_4) = S(v_5) = (0\ 1)$
$S(v_6) = (0)$
$S(v_7) = (0\ 2\ 1)$

Figure 2.1: A sample network where vertices have been assigned to areas, represented by rounded boxes. The corresponding assignment of label stacks to vertices is shown below the network.

strictly extends a stack σ_1, denoted by $\sigma_1 \sqsubset \sigma_2$, if there exists a non-empty stack $\bar{\sigma}$ such that $\sigma_2 = \sigma_1 \circ \bar{\sigma}$, namely σ_2 is longer than σ_1 and σ_2 starts with the same sequence of labels as σ_1. We say that σ_2 *extends* σ_1, denoted by $\sigma_1 \sqsubseteq \sigma_2$, if $\bar{\sigma}$ can be empty. Referring to Figure 2.1, we have that $S(v_5) \sqsubset S(v_3)$, because there exists a stack $\bar{\sigma} = (3)$ such that $S(v_3) = S(v_5) \circ \bar{\sigma}$.

We call *area* $A_\sigma \subseteq V$ a set of vertices whose stack extends σ, namely such that $\forall v \in A_\sigma : \sigma \sqsubseteq S(v)$. In particular, $A_{(l_0)} = V$. Considering Figure 2.1, the assignment of label stacks to vertices defines areas $A_{(0)}$, $A_{(0\ 1)}$, $A_{(0\ 1\ 3)}$, $A_{(0\ 2)}$, and $A_{(0\ 2\ 1)}$ (note that $A_{(0\ 2)} = A_{(0\ 2\ 1)}$). From the definition of area follows this property:

Property 1. *A vertex $v \in V$ with stack $S(v)$ belongs to all the areas in $\{A_\sigma | \sigma \sqsubseteq S(v)\}$.*

We now consider a few interesting consequences of the definition of area. First, an area cannot be empty, because it is defined by the stack of at least one vertex. Moreover, by Property 1, specifying the stack $S(v)$ for a vertex v implicitly assigns v to all areas A_σ such that $\sigma \sqsubseteq S(v)$. Thus, areas can be conveniently defined by simply specifying the label stacks for all vertices. In addition, areas can contain other areas, forming a hierarchical structure like the one in Figure 2.1. However, areas cannot overlap partially, i.e., if a vertex $v \in V$ belongs to both A_{σ_1} and A_{σ_2}, then either $\sigma_1 \sqsubseteq \sigma_2$ or $\sigma_2 \sqsubseteq \sigma_1$.

In order to limit the propagation of routing information, in our control plane the portion of network topology that falls within an area A_σ is summarized when disseminated outside A_σ. To support this summarization we introduce two additional operators on label stacks.

Given two stacks $\sigma_a = (a_0\ \dots\ a_i\ \dots\ a_n)$ and $\sigma_b = (b_0\ \dots\ b_i\ \dots\ b_m)$ such that $a_0 = b_0$, $a_1 = b_1$, …, $a_i = b_i$ for some $i \le \min(m, n)$ and $a_{i+1} \ne b_{i+1}$ if $i < \min(m, n)$, we define $\sigma_a \bowtie \sigma_b = (a_0\ \dots\ a_i)$ and $\sigma_a \rightarrowtail \sigma_b = (a_0\ \dots\ a_k)$, where $k = \min(i+1, n)$. For example, $(0\ 2\ 1) \bowtie (0\ 1) = (0\ 1) \bowtie (0\ 2\ 1) = (0)$, $(0\ 2\ 1) \rightarrowtail (0\ 1) = (0\ 2)$, and $(0\ 1) \rightarrowtail (0\ 2\ 1) = (0\ 1)$. Note that $\bowtie$ is commutative, whereas $\rightarrowtail$ is not. We also assume that $() \bowtie \sigma_b = \sigma_a \bowtie () = ()$ and $\sigma_a \rightarrowtail () = () \rightarrowtail \sigma_b = ()$. As an intuitive interpretation, $A_{\sigma_a \bowtie \sigma_b}$ is the most nested area that contains both A_{σ_a} and A_{σ_b}, namely the area within which routing information that is relevant only for vertices in $A_{\sigma_a} \cup A_{\sigma_b}$ is supposed to be confined. For example, in Figure 2.1 information exchanged by v_5 and v_7 will be confined within area $A_{S(v_5) \bowtie S(v_7)} = A_{(0)}$. Similarly, $A_{\sigma_a \rightarrowtail \sigma_b}$ is the least nested area that includes all vertices in A_{σ_a} but not those in A_{σ_b}, which is the area that vertices in A_{σ_a} declare to be member of, and which they summarize information for, when sending messages to neighboring vertices in A_{σ_b}. In Figure 2.1, v_7 summarizes and sends routing information to v_5 as a member of area $A_{S(v_7) \rightarrowtail S(v_5)} = A_{(0\ 2)}$.

A vertex $u \in A_\sigma$ incident on an edge (u, v) such that $v \notin A_\sigma$ is called a *border vertex for A_σ*, and is in charge of propagating outside A_σ a summary of the internal routing information of A_σ. For example, in Figure 2.1 v_2 is a border vertex for $A_{(0\ 1\ 3)}$. A vertex can be a border vertex for more than one area (Property 1), while a neighbor of a border vertex needs not be a border vertex itself. In Figure 2.1, v_2 is also a border vertex for $A_{(0\ 1)}$, while v_6 is not a border vertex. Derived from the definition of border vertex, we can state the following property:

Property 2. *Area $A_{(l_0)}$ does not have border vertices.*

2.4 Dissemination of Routing Information

We now illustrate how routing information is disseminated over the network. In order to do so, we first define pathlets and describe how they are created and propagated. We then introduce conditions on label stacks and routing policies that further regulate this propagation.

Pathlet

Vertices in graph G exchange path fragments called pathlets [9] in order to learn paths towards the destination vertices. We present an enhanced definition of a pathlet that is slightly different from [9]. A *pathlet* π is a t-uple $\langle FID, v_1, v_2, \sigma, \delta \rangle$ where all fields are assigned by vertex v_1: FID is an integer number, unique at v_1, that identifies the pathlet, and is called *forwarding identifier*; $v_1 \in V$ is the *start vertex*; $v_2 \in V$ such that $v_2 \neq v_1$ is the *end vertex*; σ is a stack of labels from $L \cup \{\perp\}$ called *scope stack*, used to restrict the areas where π should be propagated ($\perp$ is a special label used to mark pathlets representing network links); and δ is a (possibly empty) set of network destinations (e.g., network prefixes) available at v_2. A pathlet π describes a way to reach v_2 from v_1, without revealing the sequence of traversed vertices. By concatenating learned pathlets, a vertex can therefore construct a path towards a destination vertex. A concatenated pathlet can, in turn, constitute a new path fragment that can be propagated to other vertices as a single pathlet. Pathlets can also be labeled with Quality of Service indicators, that characterize the performance of the network portion they traverse. For the sake of readability, in the descriptions of Sections 2.4 and 2.5 we omit specifying these indicators. The relationship between pathlets and QoS will be further discussed in Section 3.3.

Packet forwarding

We now provide hints about the pathlet-based forwarding mechanism presented in [9]. Understanding this mechanism is required to determine the forwarding state information that each vertex has to maintain to support the operation of the data plane.

In pathlet routing, each data packet carries in a dedicated header a sequence of *FID*s: each *FID* in this sequence indicates a pathlet that the packet should be routed along to reach the destination. When a vertex u receives a packet, it considers the first FID in the sequence contained in its header: this FID, referenced as f in the following, identifies a pathlet π that is known at u and that has u as start vertex.

Observe that, in the general case, pathlet π may lead to an end vertex that is not adjacent to u. Also, a pathlet does not contain the detailed specification of the routing path to be taken to reach the end vertex. Hence, before forwarding the packet, vertex u has to replace f with another sequence of FIDs that indicates the pathlets to be used to reach the end vertex of π. We assume that the forwarding state of u contains a mapping $\mathit{fids}_u(FID)$ between each FID with start vertex u and a (possibly empty) sequence of FIDs. At this point, u has to pick a neighboring vertex to forward the packet to. Since also this information is missing in pathlet π, it must be kept locally at u. We assume that the forwarding state of u also contains the *next-hop* vertex $nh_u(FID)$, namely the vertex that immediately comes after u along π. Both $fids_u$ and nh_u are computed by the control plane, as explained in the following.

Atomic, crossing, and final pathlets

We distinguish among three types of pathlets: atomic, crossing, and final. An *atomic pathlet* $\pi = \langle FID, v_{\text{start}}, v_{\text{end}}, \sigma, \delta\rangle$ corresponds to a single edge between two neighboring vertices v_{start} and v_{end} in G and is such that σ ends with the special label $\perp$. δ contains the network destinations possibly available at v_{end}. Atomic pathlets are used to disseminate information about the network topology and are propagated only inside area $A_{S(v_{\text{start}})\ltimes S(v_{\text{end}})}$. Besides serving as a distinguishing mark for atomic pathlets, the special label $\perp$ simplifies the description of pathlet dissemination mechanisms, by homogenizing several special cases. Bidirectional network links are represented by two opposite atomic pathlets. For example, an atomic pathlet in the network in Figure 2.1 is $\pi_{45,\perp} = \langle 3, v_4, v_5, (0\ 1\ \perp), \emptyset\rangle$, used by v_4 to reach v_5.

A *crossing pathlet* π for area A_σ is a pathlet between a start and an end vertex that are border vertices for area A_σ. Crossing pathlets always have $\delta = \emptyset$ and do not contain label $\perp$ at the end of scope stack. Crossing pathlets are one of the fundamental building blocks of our control plane, because they offer to vertices outside A_σ the possibility to traverse A_σ without knowing its internal topology. Referring to Figure 2.1, two examples of crossing pathlets that v_2 may know are $\pi_{25} = \langle 1, v_2, v_5, (0\ 1), \emptyset\rangle$, which allows to traverse $A_{(0\ 1)}$, and $\pi_{23} = \langle 12, v_2, v_3, (0\ 1\ 3), \emptyset\rangle$, which allows to traverse $A_{(0\ 1\ 3)}$. A set of propagation conditions, described later in this section, constrains the propagation of crossing pathlets outside the area of the vertex that composed them. Since a vertex can be a border vertex for several areas, different pathlets with the same start and end vertices (but with different scope stacks and *FID*s) can act as crossing pathlets for different areas.

Last, a *final pathlet* π is a pathlet between a start vertex that is the border vertex of an area A_σ and an end vertex that is a destination vertex in A_σ. A final pathlet has $\delta \neq \emptyset$ and does not contain $\perp$ in the scope stack. An example of final pathlet that may be known by v_5 in Figure 2.1 is $\pi_{51} = \langle 11, v_5, v_1, (0\ 1), \{d\}\rangle$, where d is a destination available at v_1.

Notice that it is possible to create an atomic, a crossing, and a final pathlet between the same pair of neighboring vertices: each of these pathlets will have a different role and scope of propagation. For example, since v_4 and v_5 are border vertices for $A_{(0\ 1)}$, besides the atomic pathlet $\pi_{45,\perp}$ there can be a crossing pathlet $\pi_{45} = \langle 2, v_4, v_5, (0\ 1), \emptyset\rangle$ between the two vertices. Even if they both correspond to a single link, $\pi_{45,\perp}$ and π_{45} are two different pathlets: in fact they have different *FID*s and, unlike $\pi_{45,\perp}$, the scope stack of π_{45} does not end with $\perp$.

Pathlet creation

We now describe how atomic and crossing pathlets are created (similar mechanisms are applied for final pathlets). By *create* we mean that a vertex defines a pathlet where it appears as the start vertex, assigns a locally unique *FID* to it, and keeps it in a local data structure (see Section 2.5). We also use the term *composition* to refer to the creation of crossing and final pathlets.

First of all, each vertex $u \in V$ for each $(u, v) \in E$ creates an atomic pathlet $\langle FID, u, v, \sigma \circ (\perp), \delta\rangle$ such that $\sigma = S(u) \bowtie S(v)$, and δ contains the network destinations at v. The scope stack σ is chosen to restrict propagation of each atomic pathlet up to the most nested area that contains both u and v. For each atomic pathlet, vertex u also updates its forwarding state by setting $nh_u(FID) = v$ and $fids_u(FID) = ()$.

By concatenating atomic pathlets, a border vertex u can create pathlets between non-neighboring vertices that can be used to traverse the areas that u belongs to as if they consisted of a single link. In order to support summarization of routing information, u appears to each neighbor $x \notin A_{S(u)}$ as a member of area $\bar{A} = A_{S(u) \rightarrowtail S(x)}$. Therefore, u creates crossing (and final) pathlets for each such $\bar{A}$. In Figure 2.1, v_2 creates crossing and final pathlets for $A_{(0\ 1)} = A_{S(v_2) \rightarrowtail S(v_6)}$ to be offered to v_6 and crossing and final pathlets for $A_{(0\ 1\ 3)} = A_{S(v_2) \rightarrowtail S(v_4)}$ to be offered to v_4. To describe the creation of crossing pathlets, we introduce a set $chains(\Pi, u, v, \sigma)$ that contains all the possible concatenations of pathlets from a set Π, where each concatenation starts at u and ends at v, and where the scope stack of each pathlet extends σ. Formally, $chains(\Pi, u, v, \sigma)$ is the set of all the finite and cycle-free sequences $(\pi_1\ \pi_2\ \dots\ \pi_n)$ of pathlets in Π, where each sequence is such that $\pi_i = \langle FID_i, w_i, w_{i+1}, \sigma_i, \delta_i\rangle$, $\sigma \sqsubseteq \sigma_i$, $\pi_{i+1} = \langle FID_{i+1}, w_{i+1}, w_{i+2}, \sigma_{i+1}, \delta_{i+1}\rangle$, and $\sigma \sqsubseteq \sigma_{i+1}$, with $i \in \{1, \dots, n-1\}$, $w_1 = u$, $w_{n+1} = v$. For each neighbor x, a border vertex $u \in A_\sigma$ populates a set $crossing_u(\Pi, \sigma)$, with $\sigma = S(u) \rightarrowtail S(x)$. Set $crossing_u(\Pi, \sigma)$ contains a pathlet $\pi = \langle FID, u, w, \sigma, \delta\rangle$ for each border vertex $w \neq u$ for A_σ and for each sequence $(\pi_1\ \pi_2\ \dots\ \pi_n)$ in set $chains(\Pi, u, w, \sigma)$. FID is chosen to be unique at u and δ is set to $\emptyset$. Assuming that $\pi_i = \langle FID_i, u_i, v_i, \sigma_i, \delta_i\rangle$, the forwarding state of u is updated by setting $fids_u(FID) = (FID_2\ FID_3\ \dots\ FID_n)$ and $nh_u(FID) = nh_u(FID_1) = v_1$. Because of the way in which $chains(\Pi, u, w, \sigma)$ will be used in the following, we assume without loss of generality that π_1 is always an atomic pathlet. As an example from Figure 2.1, let $\Pi = \{\pi_1, \pi_2\}$, where $\pi_1 = \langle 2, v_2, v_4, (0\ 1\ \perp), \emptyset\rangle$ and $\pi_2 = \langle 3, v_4, v_5, (0\ 1\ \perp), \emptyset\rangle$: v_2 may compose and put in its set $crossing_{v_2}(\Pi, (0\ 1))$ a pathlet $\langle 1, v_2, v_5, (0\ 1), \emptyset\rangle$ corresponding to the sequence of atomic pathlets $(\pi_1\ \pi_2)$ taken from set $chains(\Pi, v_2, v_5, (0\ 1))$. v_2 will set $fids_{v_2}(1) = (3)$ and $nh_{v_2}(1) = v_4$.

Final pathlets, put in a set $final_u(\Pi, \sigma)$, are created in a much similar way, except that they are composed towards vertices in $A_\sigma \cap D$ and δ is set to the set δ_n of network destinations of the last component pathlet in the sequence.

Discovery of border vertices

In order to compose crossing pathlets for an area, a border vertex u must identify other border vertices for the same area. Thanks to the following property, u can achieve this by just considering the pathlets it receives.

Property 3. *A vertex $u \in A_\sigma$ that receives two pathlets $\pi_1 = \langle FID_1, v_1, w_1, \bar{\sigma}_1, \delta_1\rangle$ and $\pi_2 = \langle FID_2, v_2, w_2, \bar{\sigma}_2, \delta_2\rangle$ that have a (start or end) vertex v in common and that satisfy the following features can conclude that v is a border vertex for A_σ. (I) $\delta_2 = \emptyset$; (II) $\bar{\sigma}_1 = \sigma_1 \circ (l)$, $\bar{\sigma}_2 = \sigma_2 \circ (\perp)$, $l \in L$, $\sigma_1 \neq ()$, $\sigma_2 \neq ()$; (III) $\sigma_2 \sqsubset \sigma = \sigma_1$; (IV) $v_1 \neq v_2$ or $w_1 \neq w_2$; and (V) $v \in \{v_1, w_1\}$, $v \in \{v_2, w_2\}$ (that is, both π_1 and π_2 start or end at v).*

Proof. The fact that $v \in \{v_1, w_1\}$ implies that $v \in A_{\sigma_1}$: in fact, if $l = \perp$, then π_1 is an atomic pathlet whose scope stack is $\sigma_1 = S(v_1) \bowtie S(w_1)$; since we know that

$S(v_1) \bowtie S(w_1) \sqsubseteq S(v)$, by Property 1 we can conclude that $v \in A_{\sigma_1}$. Otherwise, if $l \neq \bot$, then π_1 is either a crossing pathlet for some area $A_{\sigma_1 \circ (l)}$ or a final pathlet; in both cases, being an endpoint of π_1, v must belong to $A_{\sigma_1 \circ (l)}$, hence to A_{σ_1} (by Property 1 again). Since $\sigma_1 = \sigma$, we can conclude that $v \in A_\sigma$. On the other hand, from the scope stack $\sigma_2 \sqsubset \sigma$ of the atomic pathlet π_2 we know that v has some neighbor that is not in A_σ: this allows u to conclude that v is a border vertex for A_σ. □

According to this property, a vertex $u \in A_\sigma$ can use Algorithm 1, called DiscoverBorderVertices(u, σ, Π), to discover other border vertices for A_σ based only on known pathlets in Π.

Algorithm 1 Algorithm that a vertex $u \in A_\sigma$ can use to discover remote border vertices for A_σ based only on the known pathlets in Π.

```
1: function DiscoverBorderVertices($u$, $\sigma$, $\Pi$)
2:     B is the set of remote border vertices for $A_\sigma$ discovered so far
3:     $B \leftarrow \emptyset$
4:     for each pair $(\pi_1, \pi_2)$ of pathlets with $\pi_1 = \langle FID_1, v_1, w_1, \bar{\sigma}_1, \delta_1 \rangle$ and
       $\pi_2 = \langle FID_2, v_2, w_2, \bar{\sigma}_2, \delta_2 \rangle$, such that $v_1 \neq v_2$ or $w_1 \neq w_2$, and $\exists v$ such that
       $v \in \{v_1, w_1\}$ and $v \in \{v_2, w_2\}$ do
5:         if $\exists \sigma_1 \neq ()$ such that $\bar{\sigma}_1 = \sigma_1 \circ (l)$, $l \in L$, and $\exists \sigma_2 \neq ()$ such that
           $\bar{\sigma}_2 = \sigma_2 \circ (\bot)$ and $\sigma_1 = \sigma$ and $\sigma_2 \sqsubset \sigma$ then
6:             $B \leftarrow B \cup \{v\}$
7:         end if
8:     end for
9:     return $B$
10: end function
```

Routing policies

We allow two kinds of policies: *filters*, which restrict the propagation of pathlets, and *pathlet composition rules*, which affect the creation of crossing and final pathlets. A filter at a vertex u may specify a neighboring vertex v and a triple $\langle w_1, w_2, \sigma \rangle$: this would prevent u from propagating to v those pathlets whose start vertex, end vertex, and scope stack match the triple. A pathlet composition rule may induce a border vertex v to compose only the best ranked among all the possible crossing (and final) pathlets, according to some optimality metric. For example, v could compose a single crossing pathlet corresponding to a shortest sequence of concatenated pathlets. Even better, pathlets can be ranked according to performance levels of the network portion they traverse, which results in favoring best performing paths. Alternatively, pathlets can be ranked according to their nature of atomic or crossing pathlet, and transit through areas could be discouraged. We argue that such policies can simplify network configuration tasks and improve scalability in a pathlet-enabled network. In fact, policies can be implemented completely independently for each area and, in case a best ranked pathlet becomes unavailable, a vertex could switch to an alternative sequence of pathlets transparently (without sending any messages).

Pathlet dissemination

In this section, we now explain how all created pathlets are disseminated to other vertices in G based on their scope stacks. Consider any pathlet $\pi = \langle FID, u, v, \sigma, \delta\rangle$ and let $\sigma = \bar{\sigma} \circ (l)$ (because of the way in which pathlets are created, such $\bar{\sigma} \neq ()$ and $l \in L \cup \{\perp\}$ must exist). A vertex w can propagate π to a neighboring vertex x either if $x = u$ or if π's scope stack does not satisfy any of the following *propagation conditions*:

1. $S(w) \bowtie S(x) \sqsubset \bar{\sigma}$: prevents propagation of pathlets outside the area in which they have been created. As a consequence, atomic pathlets with scope stack $\bar{\sigma} \circ (\perp)$ are only propagated inside area A_σ;
2. $\sigma \sqsubseteq S(w) \bowtie S(x)$: prevents propagation of crossing and final pathlets inside the area of the vertex that created them;
3. $\sigma = S(n) \rightarrowtail S(x)$: prevents $w \notin A$ from propagating crossing and final pathlets for A inside A;
4. $x = v$: prevents sending to x a pathlet that is useless for x.

For convenience, given a vertex w that is assigned label stack $S(w) = \sigma_w$, we define $N(w, \sigma_w, \pi)$ as the set of neighbors to which w can propagate a pathlet π according to the propagation conditions and to the routing policies.

Complete example

To show a complete example of creation and dissemination of pathlets, consider again the network in Figure 2.1 and let v_6 host network destination d.

In the following we assume that there are no filters applied, that the pathlet composition rule allows composition of all possible sequences of pathlets (although we only show some of them), and that *FID*s are randomly (yet uniquely) assigned integer numbers. The atomic pathlet $\pi_{24,\perp} = \langle 2, v_2, v_4, (0\ 1\ \perp), \emptyset \rangle$, created by vertex v_2, is propagated by v_2 to v_3 because $S(v_2) \bowtie S(v_3) = (0\ 1\ 3) \not\sqsubset (0\ 1)$, $(0\ 1\ \perp) \not\sqsubseteq (0\ 1\ 3)$, $(0\ 1\ \perp) \neq S(v_3) \rightarrowtail S(v_2) = (0\ 1\ 3)$, and $v_3 \neq v_4$; it is also propagated to v_1 for the same reasons. Instead, $\pi_{24,\perp}$ is not propagated by v_2 to v_6 because $S(v_2) \bowtie S(v_6) = (0) \sqsubset (0\ 1)$ (the first propagation condition applies), and it is not propagated by v_2 to v_4 because the end vertex of $\pi_{24,\perp}$ is v_4 itself. For similar reasons, $\pi_{24,\perp}$ is further propagated by v_3 to v_5, but in turn v_5 does not propagate it to v_7. Therefore, the visibility of $\pi_{24,\perp}$ is restricted to vertices inside $A_{(0\ 1)}$. In a similar way, v_5 creates the atomic pathlets $\pi_{53,\perp} = \langle 2, v_5, v_3, (0\ 1\ \perp), \emptyset \rangle$ and $\pi_{54,\perp} = \langle 3, v_5, v_4, (0\ 1\ \perp), \emptyset \rangle$, while v_4 creates the atomic pathlet $\pi_{46,\perp} = \langle 3, v_4, v_6, (0\ \perp), \{d\} \rangle$. The reader can easily find how these atomic pathlets are propagated. As a border vertex of $A_{(0\ 1\ 3)}$, v_3 will also propagate to v_5 at least a crossing pathlet $\pi_{32} = \langle 1, v_3, v_2, (0\ 1\ 3), \emptyset \rangle$ for area $A_{S(v_3) \rightarrowtail S(v_5) = (0\ 1\ 3)}$. Once pathlets have been disseminated, v_5 has learned about a set of pathlets Π and can create a crossing pathlet for area $A_{S(v_5) \rightarrowtail S(v_7) = (0\ 1)}$ that can be offered to v_7. For example, v_5 can pick sequence $(\pi_{53,\perp}\ \pi_{32}\ \pi_{24,\perp})$ from $\mathit{chains}(\Pi, v_5, v_4, S(v_5) \rightarrowtail S(v_7))$ and create in its set $\mathit{crossing}_{v_5}(\Pi, S(v_5) \rightarrowtail S(v_7))$ the crossing pathlet $\pi_{54} = \langle 1, v_5, v_4, (0\ 1), \emptyset \rangle$. Propagation of this pathlet by v_5 to v_4 is forbidden by the second propagation condition, because $(0\ 1) \sqsubseteq S(v_5) \bowtie S(v_4) = (0\ 1)$, and also by the fourth propagation condition, because v_4 is also the end vertex of π_{54}; π_{54} will however be propagated by v_5 to v_7 because $S(v_5) \bowtie S(v_7) = (0) \not\sqsubset (0)$, $(0\ 1) \not\sqsubseteq (0)$, $(0\ 1) \neq S(v_7) \rightarrowtail S(v_5) = (0\ 2)$, and $v_7 \neq v_4$. To provide an alternative path, v_5 can create another crossing pathlet $\pi'_{54} = \langle 9, v_5, v_4, (0\ 1), \emptyset \rangle$, corresponding to the sequence consisting of the single atomic pathlet $(\pi_{54,\perp})$, and propagated in the same way as π_{54}. Last, v_7 will also create an atomic pathlet $\pi_{75,\perp} = \langle 8, v_7, v_5, (0\ \perp), \emptyset \rangle$. At this point, v_7 has two ways to construct a path from itself to vertex v_6, which contains destination d: it can concatenate pathlets $\pi_{75,\perp}$, π_{54}, and $\pi_{46,\perp}$ or pathlets $\pi_{75,\perp}$, π'_{54}, and $\pi_{46,\perp}$. The availability of multiple choices supports quick recovery in case of fault and allows v_7 to select the pathlet supporting the most appropriate Quality of Service.

2.5 A Control Plane for Pathlet Routing: Messages

We now describe how the dissemination mechanisms illustrated in Section 2.4 are implemented in terms of messages exchanged among vertices and algorithms executed to update routing information. Details about algorithms to handle network dynamics and actions undertaken by each vertex in conseguence of the reception of a message can be found in Appendix A.

Message Types

Each message in our control plane can carry the following fields: `s`: a stack of labels; `d`: a set of network destinations; `p`: a pathlet; `f`: a *FID*; `a`: a boolean flag (which tells whether a vertex has "just been activated"); `o`: the vertex that first originated the message (we implicitly assume that this field is present in every message). Messages can be of the following types, with fields in square brackets:

Hello (s, d, a) periodically sent for neighbor greetings and never forwarded to other vertices; it carries the label stack `s` and the set of destinations `d` of the sender vertex. `a` is set to true when this is the first message sent since the sender vertex was activated (powered on or rebooted), and is used to allow neighbors to synchronize with complete information about the current network status.

Pathlet (p) disseminates a pathlet `p`.

Withdrawlet (f, s) withdraws the availability of a pathlet with FID `f`, scope stack `s`, and start vertex `o`. This message can only be originated by the vertex that had previously created and disseminated the pathlet.

Withdraw (s) withdraws the availability of all pathlets with scope stack `s`, and start vertex `o`.

With the exception of **Hello**, messages also have a `src` field specifying the vertex that has sent the message and used to avoid forwarding the message back to the sender (a technique known as *split horizon*). All message types but **Hello** also have a timestamp field `t` that is set by the message originator to the current clock when sending a newly created message. Unless there are exceptions to their natural assignment, in the following we omit specifying how the origin, source, and timestamp fields are set.

Upon the reception of those messages at each vertex, it undertakes different actions; algorithms describing in detail those actions are reported in Appendix A.2. In the following section, we report the details of all routing information which our algorithms rely on.

Routing Information Stored at each Vertex

Each vertex $u \in V$ keeps the following control plane information locally (note that this is not a complete view of all the routing paths): for each neighbor v, the label stack $S_u(v)$ and the set of network destinations $D_u(v)$ currently known for v; a set Π_u of *known pathlets*, consisting of atomic pathlets created by u and pathlets that u has received from its neighbors; for every area A_σ for which u is a border vertex, a set $B_u(\sigma)$ of vertices $v \in A_\sigma$, $v \neq u$, that are also border vertices for A_σ, and sets $C_u(\sigma)$ and $F_u(\sigma)$ of the crossing and final pathlets for A_σ composed by u; a set H_u, called *history*, that tracks the most recent piece of information known by u about every pathlet, being it positive or negative (withdrawal): u sends information in H_u to newly appeared neighbors, to immediately synchronize them with the current network status. Each item of H_u is a t-uple $\langle \mathit{FID}, v, \sigma, t, \mathit{type} \rangle$ where: the FID and the start vertex v identify a pathlet π with scope stack σ; t is the timestamp of the most recent information that u knows about π (it may be the time instant of when π has been composed or deleted by u, or the timestamp contained in the most recent message received by u about π); and $\mathit{type} \in \{+, -\}$ determines whether the last known information about π is positive (π has been composed by u or a **Pathlet** message has been received about π) or negative (π has been deleted by u or a **Withdrawlet** message has been received about π).

For each pathlet $\pi \in \Pi_u$, u keeps an expiry timer $T^p_u(\pi)$, that specifies how long the pathlet is to be kept in Π_u before being removed. When a new pathlet is created by u, its expiry timer is set to the special value $T^p_u(\pi) = \oslash$, meaning that the pathlet never expires. Timer $T^p_u(\pi)$ is used to prevent indefinite growth of Π_u: in fact, there may be cases when u never receives a **Withdrawlet** message for a no longer usable pathlet. For example, consider the network in Figure 2.1 and suppose that v_2 composes and announces a crossing pathlet $\pi_{25} = \langle 1, v_2, v_5, (0\ 1), \emptyset \rangle$ for area $A_{(0\ 1)}$. If link (v_2, v_6) fails, v_6 has no way to receive a **Withdrawlet** for π_{25}, because only v_2 can originate this message and the propagation conditions prevent it from being forwarded inside area $A_{(0\ 1)}$. However, v_6 can no longer use π_{25} for any concatenations and therefore has no reason to keep this pathlet in its set Π_{v_6}: π_{25} can indeed be automatically removed after timer $T^p_{v_6}(\pi_{25})$ has expired.

Also the history H_u could grow indefinitely, because an entry is stored and kept in H_u even for each deleted or withdrawn pathlet. Therefore, each vertex must also keep a *history timer*, which determines how long negative entries (i.e., with $\mathit{type} = -$) in the history H_u of u are kept before being automatically purged from H_u. Positive entries (with $\mathit{type} = +$), on the other hand, never expire.

We now specify the strategy with which the history is updated when creating or deleting pathlets. Every time a pathlet $\pi = \langle \mathit{FID}, u, v, \sigma, \delta \rangle$ is created by a vertex u, the history H_u of u is automatically updated with a positive entry $\langle \mathit{FID}, u, \sigma, T, + \rangle$, where T denotes the time instant of the creation. If an entry for the same FID and start vertex u already existed in H_u, that entry is replaced by this updated version. When pathlet π is no longer available, H_u is updated with a negative entry $\langle \mathit{FID}, u, \sigma, T, - \rangle$, where T denotes the time instant at which π has become unavailable. This entry replaces any previously existing entry referring to the same pathlet π. Observe that, in order to correctly forward data packets that still contain the FIDs of withdrawn pathlets in their headers, u waits for a timeout to happen before clearing the forwarding state corresponding to π, which means that FID cannot immediately be reassigned to a new pathlet. Here we have only described history updates due to locally created or deleted pathlets: updates triggered by received messages will be presented in more detail in Appendix A.2.

We point out that algorithms in Appendix A.2 strongly rely on these routing information.

2.6 Applicability Considerations

Setup of our control plane just involves specifying areas and filters and picking pathlet composition rules in a set of ready-to-use alternatives. We believe this makes it easy for a network administrator to adopt it in an ISP's network. We now discuss possible application scenarios and further requirements that can be accommodated in our model.

Our control plane is completely independent of the data plane that carries its messages. However, the disseminated routing information can only be exploited if a data plane that can handle pathlets is available. Such a data plane should be able to carry sequences of FIDs, specified in packet headers to route them along pathlets (see Section 2.4), and to apply push/pop operations on these sequences. Given that we are considering the internal network of an ISP, we argue that implementing the data plane, as well as our control plane, on top of Multi Protocol Label Switching (MPLS) is a quite natural choice.

It is unrealistic for an ISP to change the internal routing protocol in the whole network in a single step. We therefore envision an incremental scenario where a pathlet-enabled zone and a legacy (e.g., IP) zone coexist in the same network. In this scenario, routers at the boundary of the two zones can disseminate IP destinations in the pathlet zone in the form of final pathlets, and announce pathlet destinations in the IP zone in the form of IP prefixes. Moreover, these routers can compose crossing pathlets to allow pathlet routers to traverse an IP zone as if it were a single area, and forwarding of pathlet-based packets in the IP zone can be achieved by tunneling such packets in IP: with a suitable encoding of pathlets in AS paths, the Border Gateway Protocol (BGP) could be used as the signaling protocol to establish these tunnels.

Our control plane can compute multiple paths between the same pair of routers. Besides improving robustness, this feature can be exploited to support different Quality of Service levels: pathlets can in fact be labeled with performance indicators (delay, packet loss, jitter, etc.) characterizing the quality of the paths they exploit. When composing pathlets, a router can update these performance indicators to reflect those of the component pathlets: for example, the delay of a composed pathlet can be set to the sum of the delays of the component pathlets. In this way, routers can select pathlets (for composition, or to reach some network destination) based on the QoS requirements for a specific traffic flow.

A relatively recent trend in computer networks consists in separating the logic of the control plane of a device from the components that take care of the actual traffic forwarding: this approach is known as Software Defined Networking. The specification of OpenFlow [26], the most widely adopted standard realizing this approach, includes support for matching, pushing, and popping MPLS labels in data packets. Thus, our control plane can be implemented as an OpenFlow controller that manages MPLS labels as required by the pathlet forwarding mechanism (see Section 2.4). On the other hand, in terms of network architecture, the authors of [27] suggest that a set of devices managed by a single controller can be seen as a single logical device, in turn having its own controller: we argue that we can inherit this approach, and its scalability and manageability advantages, by assigning a separate OpenFlow controller to each area.

2.7 Experimental Evaluation

In order to verify the effectiveness and scalability of our approach, we performed several simulations, trying to answer the following questions: (i) *how does our control plane perform when the network grows in size?* and (ii) *how does the performance of our control plane compare with those of OSPF?* As an answer to question (i), we show that the number of exchanged messages and created pathlets at a vertex grows linearly with the network size, while as an answer to question (ii) we show that our control plane has competitive performance in various different network scenarios involving a single area or multiple areas.

For the simulations we used OMNeT++ [28], a component-based C++ simulation framework based on a discrete event model. We built a prototype implementation of our control plane based on the IP implementation available in the INET framework OMNeT++ package [29]: messages of our control plane are exchanged encapsulated in IP packets with a dedicated protocol number in the IP header. Our implementation comprises 3 rules for the composition of pathlets between every pair of border vertices for a certain area: `All`, that allows composition of all possible pathlets; `Avoid`, that prevents usage of crossing pathlets in the composition (i.e., prevents traversing internal areas); and `One`, that allows composition of a (randomly chosen) single pathlet. Due to the huge amount of generated pathlets, indeed hardly useful for any real application, we used `All` only in preliminary experiments. We then performed more significant experiments on topologies generated by an ad-hoc piece of software that creates a hierarchy of areas and adds routers and links randomly to the areas. The topology generation was driven by the following parameters: length N of the label stacks (we assigned to all the routers a stack of the same length); number R of routers having the same stack; number A of areas contained in each area; fraction B of the routers within an area that act as border routers for that area; probability P of adding an edge between two border vertices or, as a special case, between vertices with the same label stack. The topology generator proceeds by recursively creating areas, starting from a single area that comprises all vertices. The complete procedure is described in Algorithm 13 in Appendix A.3.

To perform experiments with topologies of increasing size, we set $N = 4$, $R = 10$, and $B = 0.2$, and varied A between 3 and 4 (correspondingly having $3^{4-1} \times 10 = 270$ and $4^{4-1} \times 10 = 640$ routers, respectively) and P in the range $[0.1, 1]$ (thus randomly varying the number of links). We settled on these values after several test runs used to assess the scalability limits of OMNeT++. Link delays were uniformly chosen in the range $[10\text{ms}, 50\text{ms}]$ and

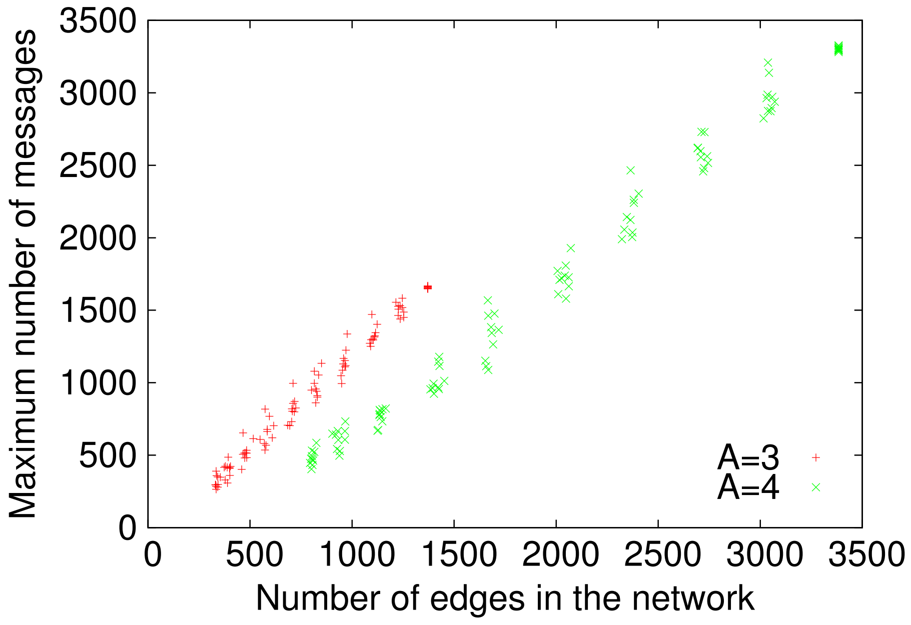


Figure 2.2: Maximum number of messages (**Hello**, **Pathlet**) sent by a router for pathlet composition rule `One`.

fixed throughout each simulation. We ran each simulation on 10 randomly generated topologies for each combination of parameters and collected the number of messages sent by each router, the number of crossing pathlets composed by each router, and the time required by our protocol to converge, namely to reach a state when no further information needs to be updated and propagated. We measured the convergence time in terms of OMNeT++ simulation time.

Scalability

The plots in Figg. 2.2 and 2.3 show some results of our scalability tests, performed with the pathlet composition rule `One`. Data points in these plots tend to be clustered in small groups: each group corresponds to the 10 runs launched on as many randomly generated topologies with a fixed set of pa-

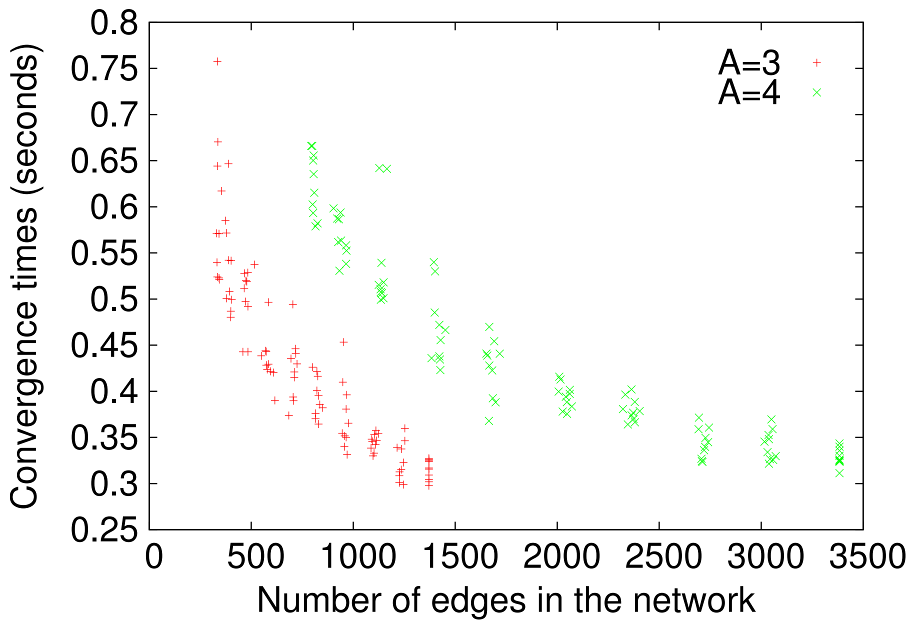


Figure 2.3: Convergence time of our control plane for pathlet composition rule `One`.

rameters. In Figure 2.2 it can be seen that the maximum number of messages sent by a router grows linearly with the number of edges in the network, regardless of the number of routers determined by parameter A. In Figure 2.3 we show that convergence times decrease as the complexity of the network grows. We ascribe this behavior to the fact that a denser network allows messages to be disseminated more quickly: in fact, they can reach relevant routers along shorter paths. We consider this as an evidence of the scalability of our protocol. We experienced very similar convergence times even with the `Avoid` pathlet composition rule: however, we are not reporting here the results of these simulations because they impacted the scalability limits of OMNeT++ and therefore showed much less clear trends.

We also performed another class of experiments to assess the impact of the appearance of a new router in the network. For this purpose, we focused on one of the topologies generated with $N=3$, $R=10$, $B=0.2$, $A=3$, and $P=0.1$:

	Number of messages	**Convergence time (ms)**	**Involved routers**
Internal vertex (1)	62	300	10
Internal vertex (2)	233	330	29
Border vertex for 1 area	221	310	30
Border vertex for 2 areas	486	370	90
Border vertex for 3 areas	1 101	480	270

Table 2.2: Impact of the appearance of a new vertex on the network.

this topology had 270 routers and 332 links. We then selected a sample of internal (non-border) and border vertices in this topology and, for each vertex, ran a simulation with the pathlet composition rule `One` where the selected vertex was activated only after the network had converged. In Table 2.2 we show the effect of the appearance of the new router in terms of total number of exchanged messages, convergence time, and number of routers that sent at least one message. As a term of comparison, consider that between $15\,190$ and $17\,423$ messages were exchanged in total during each simulation. The impact of newly appeared internal vertices depends of course on their position within the area but, since they do not compose pathlets, it is in general less evident than that of border vertices. Instead, the impact of newly appeared border vertices increases with the number of areas for which they are border vertices. Note that, since $R = 10$ and $B = 0.2$, we had exactly two border vertices per area. Therefore, in our experiments there were areas for which pathlets could only be composed after the late activation of the selected border vertex.

Comparison with OSPF

We also compared the performance of our control plane with those of OSPF, the state-of-the-art protocol for intra-domain routing. We ran both our pathlet routing prototype and the OSPF implementation included in the INET framework on a common sample of topologies, with suitable configuration adaptations to make them compatible with the two protocols. In a topology consisting of a backbone area, with 15 routers and 16 links, and 4 stub areas, each with $11-15$ routers and $12-16$ links, we observed that our control plane exchanged many more messages (which is natural because several crossing pathlets are composed for each area), yet convergence times were comparable: 439ms for OSPF and 517ms for our control plane. We then checked the scalability in a single-area scenario: we generated 10 random topologies with $N = 1$, $R = 80$, $A = 1$, and $P = 0.2$, having 80 vertices each and between 580 and 680 edges. Especially in this scenario, our control plane exhibited very competitive convergence times, in the range $[259\text{ms}, 273\text{ms}]$, when compared to the OSPF times, in the range $[664\text{ms}, 699\text{ms}]$. We recall that in this scenario pathlet composition rules have no influence (see Property 2).

Our prototype implementation, including the topology generator, is available at [30].

2.8 Open Problems

The challenge of defining a new control plane at ISP-scale leaves several problems open. For example, in terms of control plane operation, at present several pathlets must be (re)announced if a destination (dis)appears on the network, but this could be avoided or at least contained with approaches like the locator-ID separation pursued by [31]. The handling of stack change events could be improved to further limit the network portion affected by the event. Routing policies could be enriched to accommodate further requirements that we have not considered yet. And it is still unclear what is the best option to deal with dynamic changes in QoS levels without refreshing existing pathlets.

Chapter 3

Rethinking Virtual Private Networks in the Software-Defined Era [*]

Multi Protocol Label Switching (MPLS) Virtual Private Networks (VPNs) have seen an unparalleled increasing adoption in the last decade. Although their flexibility as transport technology and their effectiveness for traffic engineering are well recognized, VPNs are difficult to set up and manage, due to the complexity of configurations, to the number of involved protocols, and to the limited control and predictability of network behaviors. On the other hand, Software-Defined Networking (SDN) is a consolidated, yet still emerging paradigm by which the control plane logic of a network device is implemented by an arbitrarily programmed software that runs outside the device itself. We conjugate the effectiveness of traditional VPNs with the programmability of SDN, proposing a novel and improved realization of MPLS VPNs based on SDN. With our approach, provisioning and setup of VPNs are accomplished by using a simple and flexible configuration language. Management and troubleshooting are facilitated because only a minimal set of technologies (notably, just MPLS) is retained. Control and predictability of network behaviors are enhanced by the centralized coordination enforced by

[*]Part of the material presented in this chapter is based on the following publications: G. Lospoto, M. Rimondini, B. G. Vignoli, G. Di Battista. Rethinking Virtual Private Networks in the Software-Defined Era. In *Proc. IM*, IFIP/IEEE, 2015, and G. Lospoto, M. Rimondini, B. G. Vignoli, G. Di Battista. Making MPLS VPNs Manageable through the Adoption of SDN. In *Proc. IM*, IFIP/IEEE, 2015.

the SDN controller. Besides illustrating our proposed approach and specifying the configuration language, we describe a prototype implementation of a controller and the outcome of tests we conducted in several configuration scenarios.

3.1 Introduction

Virtual Private Network (VPN) services based on Multi Protocol Label Switching (MPLS) play a key role in the business of Internet Service Providers (ISPs). They are offered to a wide population of customers to interconnect their geographically distributed sites and continue to have an increasing market share since the late 90s [32, 33]. Despite their maturity, they still have several critical aspects, implied by the plethora of networking technologies and protocols involved in their operation (MP-BGP, OSPF, LDP, etc.). First, this makes VPNs difficult to provision, configure, manage, and troubleshoot. Second, it is hard to cast a predictable behavior from the complex interactions of these technologies, especially considering that configurations are distributed on several devices. Third, they offer limited or inconvenient methods to control routing and traffic engineering decisions. In contrast, ISPs seek for simplicity of provisioning and configuration, trouble-free management, flexibility in accommodating increasingly complex customer requirements, controllability, and predictability.

This chapter shows that SDN can give a fundamental contribution to tackle the mentioned shortcomings of MPLS VPNs. In principle, VPNs can be realized with SDN by implementing a controller that replicates the configuration interface and operation of the technologies involved in traditional VPNs. However, this approach has very limited advantages (perhaps only the centralized configuration), and inherits the drawbacks of the involved technologies. An alternative approach offering improved flexibility is to specify the configuration in the form of code fragments within the controller, for example by exploiting an API designed for the setup of VPNs: virtually arbitrary network behaviors and routing policies could be enforced in this way, but the highly increased configuration complexity may be barely tolerable by network administrators.

We leverage SDN to propose a completely different approach that preserves the current features of MPLS VPNs while bringing about a smooth provisioning, setup, and management experience for ISPs. We achieve this by introducing a simple and flexible language for a convenient centralized specification of VPN configurations. This specification is then directly translated into flow entries to be installed on datapaths. To the benefit of manageability, we drop most of the technologies currently involved in MPLS VPNs, retaining only those that are strictly required and reducing their usage to a bare minimum (we even abandon label switching). Controllability and predictability are also enhanced by the datapath coordination action enforced by the SDN controller. We have implemented a prototype of an OpenFlow SDN controller based on our approach and used it to perform extensive experiments in combination with Open vSwitch, one of the leading implementations of OpenFlow-compliant datapaths that is also used on commercial hardware such as Corsa Technology Inc. and Quanta Computer Inc. devices, as well as some Intel-based platforms.

The rest of the chapter is organized as follows. In Section 3.2 we review contributions related with MPLS VPNs and SDN. In Section 3.3 we describe our approach compared with traditional MPLS VPNs, and argue on its practical applicability. In Section 3.4 we describe our VPN configuration language and how a VPN specification is translated into flow entries. In Section 3.5 we present the outcome of our experiments. Last, concluding remarks and lines for future research are in Section 3.6.

3.2 Related Work

MPLS VPNs are a long-established and extensively documented technology (for a comprehensive introduction, see [34]). However, their complexity of setup is a matter of fact (see, e.g., [35]) and is also confirmed by the existence of several ad-hoc management and monitoring tools such as [36, 37, 38, 39]. Configuration automation solutions like, e.g., [40], can substantially cut the probability of errors but do not solve the underlying configuration complexity. In addition, current technologies for MPLS VPNs may exhibit inconsistent behaviors: for example, certain visibility constraints between VPNs may be difficult or even impossible to enforce (see [41]).

On the other hand, SDN is an approach supported by many vendors as well as by technical and research communities, including the Open Networking Foundation, the Open Networking Research Center (ONRC), the IETF, and the IEEE. Its most widely adopted realization is OpenFlow [1], whose most complete specification is 1.5.0 [8].

Despite the converging interest on SDN, there are really few attempts to import its flexibility in the implementation of VPN services. The most active research line is led by the ONRC [42]: in [43] they share our concerns on the inconveniences of the technologies currently used for VPNs and they implement an SDN-based MPLS traffic engineering solution that eliminates the need for intra-domain routing protocols. In [44] they demonstrate the feasibility of reimplementing several features of MPLS VPNs using an SDN controller. Similar results are presented in [45]. However, these contributions tend to focus more on traffic engineering aspects, are rather tied to the existing way of realizing VPNs, do not tackle the configuration difficulty problem, and lack a rigorous specification of SDN-related technical aspects. Although this research line is still alive, we are not aware of any progress on these contributions or deployments in operational networks.

We exceed existing contributions by rethinking VPNs with manageability as a driving principle: our approach simplifies the configuration by introducing a language that addresses only domain-specific concepts (e.g., customer sites, routing policies), and takes advantage of SDN to reduce the involved technologies to a very small set, thus abating or even eliminating the need for a complex VPN management system. While we do not pretend to replace sophisticated management suites, we believe the simplicity and clear specification of our solution can make it appealing for an ISP that is transitioning to SDN.

3.3 A Novel Approach to Realize VPNs with SDN

In this section we briefly review the current implementation of MPLS VPNs, discuss its inconveniences, and describe how we leverage SDN to address them and offer additional features.

Architecture and Technologies of VPNs Today

An MPLS VPN is a service that an ISP can offer to support the communication between geographically distributed sites of a customer network. Each site accesses the service by an interconnection between a *Customer Edge* (CE) router deployed in the customer's network and one or more *Provider Edge* (PE) routers which act as entry points to the ISP's network. Internal routers within the ISP are called *Provider* (P) routers and support exchange of MPLS-encapsulated traffic between different sites of the same customer. Traffic between different customers is kept separate by tagging packets with different MPLS labels, each identifying a VPN.

The setup of MPLS VPNs involves the following steps:

1. Setup of IP addresses for loopback interfaces of PE routers.

2. Setup of an internal routing protocol (e.g., Open Shortest Path First (OSPF) [2]) to ensure reachability between the loopback interfaces of different PE routers. This may involve basic traffic engineering parameters like link weights.

3. Setup of the Border Gateway Protocol (BGP) [46] to distribute routing information about customer networks among PE routers. The IP prefixes of each customer site are propagated on a full mesh of iBGP peerings established between PE routers (route reflectors can be used to improve scalability). Prefixes belonging to different customers are differentiated using the Route Distinguisher (RD) address qualifier offered by the MultiProtocol BGP (MP-BGP) [47] extensions, thus allowing address space overlaps between different customers. Optionally, prefixes can be tagged with a Route Target (RT) extended community to implement customized VPN topologies or allow communication between different VPNs. Optional NAT translation rules can be set up on some PE routers to grant customers with overlapping address spaces access to shared services or to the Internet.

4. Setup of the Label Distribution Protocol (LDP) [48] to assign MPLS labels. These labels identify paths between PE routers computed by the internal routing protocol.

5. Setup of MPLS forwarding. Packets going from a customer site to another travel in the ISP's network encapsulated in MPLS with two labels: one (inner) identifying the VPN and another (outer) identifying the label-switched path towards the PE router to which the target customer site is attached. Optionally, additional traffic engineering mechanisms (e.g., based on RSVP-TE) can also be deployed.

It is evident that even a simple VPN setup involves several technologies, exposing to the risk of configuration errors and complicating management and troubleshooting significantly. This is exacerbated by the fact that configurations are scattered on tens or even hundreds of devices, making their interplay extremely difficult to determine, especially in the presence of dynamic events (e.g., link failures). In addition, current technologies offer limited flexibility and control: for example, influencing routing paths by tuning OSPF weights is a hard task, sometimes accomplished by trial and error, and some integrity constraints just cannot be enforced (see, e.g., [41]).

Our Approach

We propose an SDN-based approach to realize VPNs that offers an extremely simplified configuration interface and preserves only the building blocks of a VPN that are strictly required to achieve the desired functionalities, thus gaining flexibility, controllability, and predictability.

In our approach CE routers are still standard IP routers set up with a default route pointing to the PE router they are connected to. On the other hand, we assume that all nodes in the ISP's network are SDN-enabled devices (datapaths) coordinated by an OpenFlow controller. Flexibility of management is improved in this scenario because the role of a network node (PE or P router) can be changed by just intervening on the SDN controller, without the need for costly firmware or hardware replacements. Because of its adequacy and widespread availability, we keep MPLS as the encapsulation technology for packets traversing the ISP's network.

We designed an SDN controller that orchestrates all datapaths in the ISP's network to set up VPNs without the need for additional technologies. Several configuration elements, as well as signaling and routing protocols, are no longer required, thus ruling out many subtle interactions and making the outcome of configurations more predictable. Referring to the steps in Section 3.3, we make the following simplifications:

1. PE routers still need an IP address, but it is only used to contact the SDN controller over IP: their forwarding behavior is instead completely managed by the SDN controller.

2. Internal routing protocols are no longer required, because the SDN controller knows the network topology and can compute routing paths between PE routers as needed.

3. MP-BGP is also dropped, because address space overlaps between VPNs are handled by distinct flow entries installed on the datapaths. Any special routing policies (including, e.g., balancing on multiple uplinks) can be part of the controller's operational logic, therefore RDs and RTs are not needed either. NAT address translation rules are translated into flow entries as well.

4. LDP is also abandoned, because the assignment of MPLS labels is completely managed by the SDN controller.

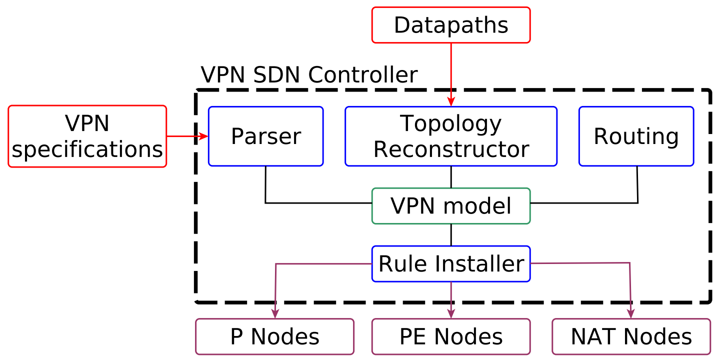


Figure 3.1: Architecture of our SDN controller for MPLS VPNs.

5. We retain MPLS to forward packets from a PE router to another, but labels are pushed and popped by flow entries installed by the SDN controller based on VPN configurations and, possibly, on traffic engineering mechanisms.

In our approach, the setup of all VPNs is described in a single configuration specification, thus making it readily accessible. This specification contains information about each customer site, including the IP subnets it hosts (possibly learned by means of a routing protocol), the PE router it is attached to, and an optional specification of a PE router that is the egress point to the Internet. It also contains associations between customer sites and VPNs and a selection of the routing policy to be adopted for each VPN. Optionally, it contains NAT address translation rules and further policies for the communication between customer sites. We describe these information in detail in Section 3.4.

Our SDN controller consists of the components depicted in Figure 3.1, which realize the following functionalities.

Configuration parsing and basic setup

The controller fetches two kinds of configuration information. Global settings consist of a mapping between datapath IDs and symbolic names, used to build a datapath inventory and to conveniently reference devices in the rest of the configuration. VPN configurations, specified using the language in Section 3.4, instruct the controller about the VPN scenario to be realized. The controller continuously watches these configurations for possible changes.

Topology reconstruction

The controller builds a complete representation of the network topology and keeps it up to date as network events (datapath/link failures or additions) occur. This activity is accomplished by exploiting the Link Layer Discovery Protocol (LLDP) [49] in the standard way envisioned by OpenFlow. When a link change event is triggered, the following functionality is activated.

Computation of routing paths

The controller (re)computes routing paths between PE routers as needed whenever a topology change is detected. If multiple paths are available between two PE routers, it selects a best path for each VPN, based on the routing policy specified in the configuration (e.g., shortest path). Although not within the focus of this paper, this policy may realize traffic engineering by taking into account the dynamic state of the network (e.g., link utilization).

Flow entries installation

The above components concur to maintain an internal representation of the VPN scenario that is used by another component to appropriately deploy OpenFlow rules. In particular, VPN configurations are translated into flow entries to be installed on datapaths, as described in Section 3.4. Flow entries are always installed proactively, namely before any traffic is exchanged (we briefly discuss in Section 3.4 how to improve scalability by installing some entries in reaction to observed network traffic). If a configuration change occurs, we assume that the controller clears all flow tables of all datapaths and immediately installs new flow entries according to the new configuration.

Unlike current MPLS VPNs, where a different MPLS label is used for each link and forwarding between PE routers happens by label switching, we use a fixed label along all the path. This is very beneficial for manageability, because the same label has the same semantic at any point in the ISP's network, and performance, because packet headers need not be rewritten. Similarly to some existing implementations, forwarding decisions on P routers are not affected by a packet's input port. Complex routing policies such as distinct routing paths per VPN or per service class, multipath routing, or destination-based load balancing can still be achieved by a suitably structured partitioning of the label space.

Applicability Considerations

Our approach can be readily implemented in existing networks provided that datapaths support release 1.1 of the OpenFlow specification, (i.e., the earliest one that introduced support for handling MPLS labels). At the time of writing, several major vendors including Corsa Technology [50] Brocade [51], Huawei [52], Arista Networks [53], and NEC [54] already introduced OpenFlow 1.3 support in their product offering. Other vendors, as well as renowned silicon producers like Broadcom and Marvell, have OpenFlow 1.3 support in their roadmap (see, e.g., [55, 56]) or will do in the foreseeable future. Big market players such as Cisco and Juniper propose customized SDN implementations that offer similar functionalities under a different interface, but they also support open standards by means of OpenFlow compatibility modules (see, e.g., [57]). Recent releases of the OpenFlow specification are usually quickly adopted in leading software switch implementations such as Open vSwitch [58], Lagopus [59], and Switch Light [60], which are also designed to run on bare metal and can leverage packet processing libraries such as Intel's DPDK [61] in order to achieve production-level performance.

Due to hardware constraints, the size of flow tables is often very limited. While this is a realistic concern, in our opinion several arguments can mitigate it. First of all, even the most limited devices, which can accommodate few thousands of flow entries (see, e.g., [62]), can support the typical amount of MPLS VPNs operated by a small-sized ISP. Moreover, vendors move towards adopting more powerful silicon capable of handling tens of thousands of flow entries [63], and a suitable combination of optimization methods, like e.g. [64], and smart hardware solutions (see, e.g., [65, 66]) can push this limit up to millions of flow entries, practically ruling out scalability problems. Finally, such problems are insignificant for software switches, whose scalability limits can be overcome by affordable hardware upgrades, and whose performance are continuously pushed towards production level (see, e.g., [67]).

The transition from an existing operational network offering MPLS VPN services to an OpenFlow-enabled network should not be a concern either: there exist solutions, like RouteFlow [68], conceived to support the coexistence of OpenFlow and standard routing protocols. On the other hand, a project within the Open Networking Foundation is releasing use cases and guidelines to accomplish such transition (see, e.g., [69]). In addition, our approach can be applied regardless of whether controller-switch communication is realized out of band or in band. Finally, existing techniques to realize the SDN controller with a distributed architecture can be adopted to improve fault tolerance (see, e.g., [70], which also addresses network partitioning).

3.4 A Configuration Language for VPNs

In this section we describe our VPN specification language. Although well-established configuration languages (e.g., YANG [71]) already exist, as far as we know this the first one specially designed for VPNs: as such it addresses domain-specific concepts (e.g., customer sites) as opposed to technical aspects (e.g., modules, namespaces, etc.), making the configuration much simpler. In this section we also explain how a configuration is turned into flow entries.

Configuration Primitives

First of all, a preamble in the configuration specifies, for each datapath, an association between its ID and a symbolic name.

Listing 3.1: Datapath's specification.

```
<datapath name="datapath_name" dpid="DPID" />
```

For example `<datapath name="pe4" dpid="4" />` is an instance of the above statement. Following that, VPN configurations are specified by the following elements.

Specification of the customer sites

It defines the IP subnets owned by each site (which may be automatically learned by means of a routing protocol like BGP), the PE router and port it is connected to, and, optionally, a PE router that acts as an egress point towards the Internet for that site.

Listing 3.2: Customer site's specification.

```
<customer-site name="customer_site_name">
  <network subnet="IP_subnet" />
  <connection pe="PE_name" port="PE_port" />
  <default pe=egressPE_name" />
</customer-site>
```

In the following, an example of this statement.

```
<customer-site name="BigCompany_Rome">
  <network subnet="10.0.0.0/16" />
  <network subnet="10.172.0.0/16" />
  <connection pe="pe_rm" port="3" />
  <default pe="default_pe" />
</customer-site>
```

Specification of the VPNs to be set up in the network

For each VPN, it consists of a set of participating customer sites and of the policy to be applied to compute routing paths between PE routers (at present we envision only the shortest path policy, but more elaborate traffic engineering policies along the lines of [44] are of course possible).

Listing 3.3: VPNs' specification.

```
<vpn name="VPN_name">
  <site id="site_name" />
  <routing type="policy" />
</vpn>
```

A possible instance of this statement is reported in the following.

```
<vpn name="BigCompany">
  <site id="BigCompany_Rome" />
  <site id="BigCompany_Tokyo" />
  <routing type="shortest-path" />
</vpn>
```

Specification of NAT address translation mechanisms

They are required when multiple VPNs with overlapping address space need to access the Internet or any shared services. Each customer site can use a different mechanism, and each mechanism comprises one or more translation rules that apply to single IP addresses or IP subnets.

Listing 3.4: Address translation mechanisms' specification.

```
<nat egresspe="PE_name" egressport="PE_port">
  <mapping vpn="VPN_name" site="site_name">
```

```
    <rule private="private_IP" public="global_IP">
  </mapping>
</nat>
```

In this specification `egresspe` and `egressport` indicate the PE router and port that are connected to the Internet. Each `mapping` statement determines that the following IP address translation rules apply to traffic coming from a specific `site` of a specific `vpn` and directed towards the Internet. Each `rule` specifies that a packet originated from a `private` IP address must have its source address translated to a `public` IP address when sent to the Internet (packets received in response must have their destination address translated accordingly). `private` and `public` can be single IP addresses (*static NAT*). Alternatively, they can be two IP subnets of equal size, in which case IP addresses in the `private` subnet are mapped to IP addresses in the `public` subnet (*dynamic NAT*). Finally, `private` can be a wildcard '*' and `public` can be an IP subnet, in which case any IP address belonging to the considered `vpn` and `site` is dynamically mapped to an available IP address in the `public` subnet (*full NAT*) when traffic is sent to the Internet.

A complete example of NAT specification is reported in the following.

```
<nat egresspe="default_pe" egressport="1">
  <mapping vpn="BigCompany" site="BigCompany_Rome">
    <rule private="10.0.100.8" global="211.10.20.4" />
    <rule private="10.0.200.0/24" global="211.10.40.0/24" />
  </mapping>
  <mapping vpn="BigCompany" site="BigCompany_Tokyo">
    <rule private="*" global="190.25.0.0/16" />
  </mapping>
</nat>
```

In this example an IP address of site `BigCompany_Rome` in VPN `BigCompany` is statically mapped to another IP address when traffic from that site is sent to the Internet. Moreover, for the same site the controller will establish a mapping between addresses in subnet `10.0.200.0/24` and addresses in subnet `211.10.40.0/24`. Last, the controller will dynamically remap any IP addresses belonging to site `BigCompany_Tokyo` of VPN `BigCompany` to an available address in subnet `190.25.0.0/16`.

NAT translation rules are optional: a PE router can be the Internet egress router for a VPN even if no NAT rules are configured. Of course, if multiple

VPNs with an address space clash need Internet access, response traffic from the Internet may be handled unpredictably in the absence of NAT rules.

Specification of policies

With our language it is also possible to express complex routing policies. For example, allowing communication between customer sites in different VPNs is as simple as adding both sites to a new VPN created ad hoc. In a traditional MPLS VPN, this would require a selective tagging of IP prefixes with RTs and a leakage of tagged routes from one VPN to the other. Forbidding communication between customer sites is also possible, but a bit more difficult: to prevent traffic between customer sites S_1 and S_2 belonging to the same VPN V_1, one of the two sites, say S_1, can be moved to a separate VPN V_2. In addition, all the sites of V_1 except S_2 must then be added to VPN V_2 to permit communication with S_1. In order to make such configurations easier to specify and to support more flexible routing policies, we introduce an additional primitive.

Listing 3.5: Policies' specification

```
<policy vpn="VPN_name">
  <deny from="from_subnet" to="to_subnet" />
  <allow from="from_subnet" to="to_subnet" />
</policy>
```

This specification blocks or allows traffic originated by IP addresses in subnet `from_subnet` and directed to IP addresses in subnet `to_subnet` within VPN `VPN_name`. With this primitive it is possible to selectively control communication between specific subnets of customer sites.

In the following, we report an example of policies' specification.

```
<policy vpn="BigCompany">
  <deny from="10.10.100.0/24" to="10.10.200.1/32" />
</policy>
```

General Considerations

Even if not explicitly stated, a configuration expressed with this language determines the role played by each datapath (P router, PE router, NAT), which in turn determines what flow entries the controller will install on that datapath. Actually, all the datapaths must at least be able to forward traffic between PE routers, namely must behave as P routers. The other roles are determined by the configuration context: for example, a datapath mentioned in a `connection` statement is to be considered a PE router. Another relevant aspect is that every packet coming from a customer site and entering the ISP's network must be assigned to the correct VPN. This is not difficult in general because, even if the customer site participates in multiple VPNs, the packet's destination IP address may belong to only one of these VPNs. An ambiguity arises if the source customer site and the destination IP address belong to more than one common VPN. We solve this ambiguity by assuming that the packet is sent to the IP address in the first VPN specified in the configuration.

From Configurations to Flow Entries

We now describe how the SDN controller translates VPN configurations into flow entries to be installed on datapaths. Unless differently specified, all flow entries are installed proactively. We divide flow entries into classes according to their role and structure. Also, we call *ingress PE* the router through which packets from a customer site enter the ISP's network and *egress PE* the router through which these packets leave the ISP's network and reach their destination. Table 3.1 can be used as a reference to follow the description of the flow entries.

We first consider flow entries installed on P routers: these flow entries support delivery of MPLS-encapsulated packets between PE routers, therefore we call them *tunnel flow entries*. Each entry matches only on the outer MPLS label of a packet, which determines a path towards a PE router, and sends the packet out of the port that leads to that PE router along that path. If the P router under consideration is the penultimate hop along the path to the PE, the flow entry also pops the outer label before forwarding the packet. In principle, on a P router there is one tunnel flow entry for each PE router. However, since the same MPLS label is used along all the path and the union of all the paths to a PE router forms a tree, in most practical cases the number of installed flow entries is lower. On the other hand, the controller may also compute multiple paths (for example, one for each VPN) towards the same PE router to support traffic engineering: correspondingly, there can be multiple tunnel flow entries for the same PE router.

We now describe flow entries installed on PE routers, which belong to 3 different classes. These entries depend on VPN configurations and on a

Table 3.1: An overview of the flow entries installed on datapaths in the ISP's backbone to support VPN services.

Entry class	**Datapaths**	**Match conditions**	**Actions**	**Expected number of entries**
Tunnel	P	MPLS label	If penultimate hop towards a PE, pop the label. Send packet out of the port along the path determined by the label.	One for each PE (or more, for traffic engineering)
Local delivery	PE	Input port, source and destination IP subnet	Send packet to the target customer site.	One for every pair of IP subnets in directly connected customer sites in the same VPN
Remote delivery	Ingress PE	Input port, source and destination IP subnet	Push one label for the VPN and one for the path to the egress PE. Send packet out of the port along the path to the egress PE.	One for every pair of IP subnets in remotely connected customer sites in the same VPN
Remote delivery	Egress PE	MPLS label, destination IP subnet	Pop label. Send packet to the target customer site.	One for every IP subnet of every VPN of directly connected customer sites
Default-out	Ingress PE	Input port, source IP subnet	Push one label for the path to the default PE. Send packet out of the port along the path to the default PE.	One for every IP subnet of every customer site that has a default PE
Default-out	Egress PE	—	Send packet out of the configured Internet-connected port.	One
Default-in	Ingress PE	Input port, destination IP subnet	Push one label for the VPN and one for the path to the default PE. Send packet out of the port along the path to the egress PE.	One for every IP subnet of every customer site that uses this specific PE as default PE

proper identification of the destination VPN and customer site of each packet, as specified in Section 3.4. The first class of flow entries supports connectivity between customer sites attached to the same PE router: we call them *local delivery flow entries*. Such a flow entry matches on the input port as well as on the source and destination IP subnet, and sends the packet out of the port connected to the target customer site. In the absence of configured policies, the controller only installs entries that support communication between sites (and prefixes) in the same VPN. On a PE router there is at most one local delivery flow entry for every pair of IP subnets in directly connected customer sites belonging to the same VPN. In practice, there can be fewer flow entries because two customer sites may share more than one VPN (a single entry for each pair of IP subnets is then enough to support communication for all these VPNs).

The second class of flow entries at PE routers supports delivery of packets between sites connected to different PE routers: we call them *remote delivery flow entries* and describe them separately for ingress and egress PE routers. A remote delivery flow entry at an ingress PE router matches on the input port as well as on the source and destination IP subnets. Upon successful match, it pushes two MPLS labels on the packet (one identifying the VPN and the other identifying the path to the egress PE) and sends the packet out of the port that is along the path to the egress PE. The SDN controller assigns a distinct label to each VPN and to each routing path: when installing remote delivery flow entries, the controller selects the VPN label to be applied as specified in Section 3.4, and selects the path label according to the computed routing paths (which, in turn, depend on the configured routing policies). For the count of remote delivery flow entries, the same considerations made for local delivery flow entries apply. A remote delivery flow entry at an egress PE router matches on the VPN label of a packet and on its destination IP subnet, it pops the label, and it sends the packet out of the port to which the target customer site is connected, which is determined based on the destination IP and the VPN. At a given PE router there will be one such flow entry for each IP subnet of each VPN of the customer sites that are connected to that router.

The third class of flow entries supports routing of network traffic from a customer site of a VPN to the Internet through the PE router that has been configured as default for that site. We call such entries *default-out flow entries*, and those supporting response traffic from the Internet *default-in flow entries*. We first describe the basic structure of these flow entries in the absence of NAT address translation rules. A default-out flow entry at an ingress PE matches on the input port (required because every customer site can use a distinct PE router as an egress to the Internet) and on the source IP subnet, it pushes one MPLS label indicating the path to the egress PE router (one label suffices because packets are not directed to a specific VPN but to the Internet), and sends the packet out of the port that is along the path to the egress PE router. Flow entries of this type are installed with a lower priority value so that they are matched only after any other flow entries, thus implementing a “default route”. There is one such flow entry for every subnet of every customer site that has been configured for Internet access. A default-out flow entry at an egress PE just sends the packet out of the Internet-connected port, specified in the configuration. On a specific PE router there is just one such flow entry, and it is also installed with a low priority value. A default-in flow entry at an ingress PE router matches on the input port (the Internet-connected port) and on the destination IP subnet, pushes two labels on the packet (one for the VPN and one for the path to the egress PE), and sends the packet out of the port that leads towards the PE router that is connected to the destination customer site. If the destination IP address belongs to more than one VPN, the actual target VPN can be determined as assumed in Section 3.4, or proper NAT translation rules need to be set up to prevent any ambiguity. There is one default-in flow entry at an ingress PE router for every IP subnet of every customer site that uses that PE router as an exit point to the Internet. There is no need for specific default-in flow entries at an egress PE router, because their function is accomplished by remote delivery entries. If a customer site uses the PE router it is connected to as the default PE router, ingress and egress flow entries are condensed into a single entry: in this scenario a default-out flow entry matches on the input port and on the source IP subnet and sends the packet out of the Internet-connected port; a default-in flow entry matches on the Internet-connected port and on the destination IP address, and sends the packet out of the port connected to the relevant customer site.

Depending on their structure, the SDN controller can install flow entries at different moments: entries that are independent of the ISP network’s topology, including local delivery flow entries and entries that are deployed on egress PE routers, can be installed immediately after the controller has fetched VPN configurations, and are never altered unless configuration changes occur. Any other entries, including tunnel flow entries and entries deployed on ingress PE routers, involve an output port in their actions and therefore depend on the network topology: as such, they can only be installed once the controller has computed paths between PE routers. If the controller recomputes these paths, for example because better ones become available, existing ones are disrupted (e.g., due to link failures), or because of traffic engineering policies, such entries need to be updated accordingly.

Our approach requires the installation of a number of flow entries that is comparable with the number of forwarding table entries of routers implementing traditional MPLS VPNs, and in general can be lower. Tunnel flow entries are at most as numerous as forwarding entries because, since we abandon label switching, there is no need for a distinct entry for each input port. Default-out flow entries at the egress PE are less numerous because one entry suffices for all VPNs. On the other hand, for some classes (local delivery and entries installed at ingress PE routers except default-in entries) we use a higher number of flow entries, because they also match on the source IP subnet. We made this choice to prevent forwarding traffic from unintended VPNs, thus ruling out the integrity and isolation problems documented in [41]. If this is not a concern, the number of flow entries can be largely reduced.

We now describe NAT-related flow entries, which are implemented as a variation of other classes of entries at PE routers. The specification of NAT rules in the configuration is independent of whether NAT is implemented on ingress or egress routers (see [72] for a discussion of the two alternatives), but different flow entries are installed in the two cases. Implementing NAT at the egress requires the following changes. Default-out flow entries at any ingress PE router must also push the VPN label, and default-out flow entries at any egress PE router must also match on this label and on the source IP subnet: this is required to recognize the source customer site and apply the correct address translation rules. Before forwarding a packet, default-out flow entries at the egress PE must also replace its source IP address with a public address according to the configured rules. Similarly, when a default-in flow entry at an ingress PE router matches a response packet from the Internet directed to a mapped IP address, it must replace its destination IP with the original private address before forwarding the packet. On the other hand, NAT can be equivalently implemented at the ingress in the following way: each default-out flow entry at the ingress PE router of a customer site that is configured for NAT must also replace the source IP address of each packet with a public address before forwarding it. Similarly, default-in flow entries at the same router, which are implemented by remote delivery flow entries, must replace the destination IP address of a packet directed to a mapped IP address with the original private address. Regardless of where NAT is implemented, if a customer site uses the PE router it is attached to as the default PE router, then the entire address mapping takes place on that router.

In order to implement NAT, existing default-out and default-in flow entries must be replicated for each IP address that is subject to translation. In accordance with [72], implementing NAT on ingress routers can improve scalability in this scenario, because flow entries can be distributed across several devices. These considerations apply as long as flow entries are installed proactively. An alternative approach (the most viable one for full NAT) is to install NAT-related flow entries on-demand when traffic from new source IP addresses is observed, and to set them up with a timeout so that public IP addresses can be reused when this traffic is interrupted. With this solution, the number of NAT-related flow entries equals the number of entries in a traditional NAT translation table.

The last class of flow entries implements forwarding policies specified with the `<policy>` primitive defined in Section 3.4. We call such entries *policy flow entries* and assume they are deployed at ingress PE routers to which customer sites hosting the specified subnets are attached. For each `<deny>` policy, a flow entry is installed that drops any packets within the specified VPN (determined by the input port) that match the specified source and destination IP subnets. These entries are installed with a high priority, so that they override any other matching flow entries. For each `<allow>` policy, additional local or remote delivery flow entries are installed to support communication between the specified subnets.

In order to translate VPN configurations into flow entries, our controller reconstructs an internal representation of the network topology, as well as of the VPNs and the participating customer sites, and sends to the datapaths the appropriate flow entries for each class, as described above and summarized in Table 3.1. Although conceived with OpenFlow primitives in mind, our language is independent of the specific technologies used to deploy VPN configurations: in principle, a specification expressed in our language could be implemented on network devices using other protocols like, e.g., NETCONF.

3.5 Experimental Evaluation

In order to validate the effectiveness of our approach, we implemented a prototype SDN controller (available at [73]) and performed several experiments.

Scenario and Technologies

We developed the controller using Ryu[†] [74], a Python-based framework which extensively supports OpenFlow specifications 1.0 through 1.4, enabling us to

[†]Ryu 3.8, latest Git snapshot as of April 2nd, 2014.

handle MPLS labels. We tested our controller with Open vSwitch[‡] [58], one of the most feature-rich implementations of an OpenFlow-compliant software switch that is also used on a range of commercial hardware devices (e.g., Corsa Technology switches). We ran our tests inside Mininet[§] [75], a state-of-the-art emulator that is commonly adopted for experimenting with SDN.

Our prototype implements all the components described in Sections 3.3 and 3.4. At present configuration parsing, topology discovery (by means of an API offered by Ryu), routing path computation between PE routers, and all NAT types (static, dynamic, full) are supported. In addition, the controller reacts to topology changes: when a new link is detected, the Topology Reconstructor is notified (see Figure 3.1) and the Routing component recomputes paths between each pair of PE routers for which a path is not yet available. If a new path is found, the Rule Installer deploys flow entries on the relevant datapaths. When an existing link disappears, the Routing component determines whether there are currently used paths in which the disappeared link occurs. For every such path, the Rule Installer removes from the relevant datapaths all the flow entries that support that path and that are not used by any other paths (we use a reference counter to determine when an entry can be removed). Other features are unsupported or only partially implemented: our prototype controller understands a slightly simplified version of our configuration language, it does not detect on-the-fly configuration changes, it selects a random shortest path between PE routers, and it only allows the specification of a single IP subnet for each VPN of each customer site. We argue that none of these features affects the significance of our experiments.

In implementing the controller we had to face some technical issues. Due to the lack of a feature called "recirculation", in the official implementation of Open vSwitch it is currently impossible to look past MPLS labels in a match condition. We overcame this problem by using VLAN tags as VPN identifiers in place of MPLS labels. Accordingly, we split each flow entry matching both the VPN identifier and the IP subnet into two entries placed in separate flow tables: one entry that matches the VLAN tag, stores its value in a standard *Metadata* registry, removes the VLAN header, and points to the other flow table; another entry that matches the Metadata and the IP subnet and performs the required actions. We also experienced problems with the MAC address learning module of Ryu, which we worked around by populating the ARP caches of host machines in advance. Finally, due to a Mininet bug [76], for

[‡]Open vSwitch version 2.3.90, latest Git snapshot as of May 19th, 2014.
[§]Mininet version 2.1.0+, latest Git snapshot as of May 7th, 2014.

Table 3.2: Results of our test runs with different topologies (from [77]) and VPN configurations.

Topology	**Nodes**	**Links**	**Max flow entries**	**Convergence time (s)**	**Recovery time (s)**
Att	25	56	98	0,8	0,03
Bell Canada	48	64	47	1,2	0,03
BT Europe	24	37	106	0,5	0,05
Evolink	37	45	63	0,7	0,1
NTT	47	63	67	0,8	0,04
Oxford	20	26	100	0,6	0,24
Renater2006	33	43	72	0,6	0,11
York	23	24	91	0,7	0,17

each test we had to update the association between Open vSwitch port numbers and virtual network interfaces. None of these problems is due to flaws in our approach or even OpenFlow itself: they are implementation-specific issues that do not impair the usage of our controller with hardware datapaths.

Experiments

We exploited our prototype to verify the functionality, scalability, robustness, and performance of our approach. For these purposes we used 3 groups of topologies and VPN configurations (the realism of these configurations was confirmed in a discussion with a medium-size ISP):

- A simple topology with 2 hosts in the same VPN, each attached to a PE router; the two PE routers were connected by 3 independent paths made up of 10 P routers each. We used this topology for functional tests.

- A more complex topology consisting of 12 P routers and 7 PE routers, each with one host attached and 2 of which performing NAT, totaling 25 links. We used variations of this topology to consider increasingly complex scenarios: at first just 2 customer sites in the same VPN, then more VPNs with overlapping IP address space, and finally VPNs with default

PE routers and NAT translation rules (we modeled "the Internet" as a host connected to each default PE router).

- A range of real-world topologies from the Topology Zoo [77] project. For each of them we considered 100 VPNs, each consisting of 2 hosts in 2 different IP subnets. The hosts in each pair were attached to 2 distinct randomly picked network nodes which therefore acted as PE routers (at most 33 VPNs were set up on every PE); all the other routers were just plain P routers. We used these topologies for scalability tests.

Functionality

For the functionality tests, we successfully verified connectivity between hosts in the same VPN and isolation between hosts in different VPNs, using a standard ping. We also verified that packets were encapsulated as expected.

Scalability

For the scalability tests, we assessed the number of installed flow entries and checked that it did not exceed the expected count stated in Section 3.4. In the most complex topology of group b) that we considered (9 VPNs, 2 customer sites for each VPN with one IP subnet each, and 9 static NAT rules) we had at most 36 flow entries on PE routers and at most 3 on P routers. Table 3.2 shows the maximum number of flow entries on any datapath for the topologies of group c). We believe these numbers show that our approach is pretty conservative in terms of consumed flow table space.

Robustness

To verify robustness to topology changes, we simulated several link failures, carefully selected to disrupt a high number of paths taken by network traffic. In the presence of such faults, packet loss may occur for at least two reasons. One is that packets may get trapped in a disconnected network region. The other reason is related with the behavior of datapaths: if an incoming packet does not match any flow entries, a datapath sends a request to the controller and only buffers the packet until a response is received. We ascertained that, if a matching flow entry is installed independently of this request-response transaction (like our controller does), the packet is lost. We verified the impact of link failures during a flood ping (i.e., a ping at the maximum rate made possible by the network) and during a TCP file transfer between two hosts:

no more than 4 ICMP packets were lost in all our tests, and the transfer was never disrupted nor significantly slowed down. Our implementation can be further improved because it needlessly clears and reinstalls some flow entries involved in the changed paths.

Performance

To evaluate performance, we started the controller in advance and estimated the time since the network was brought up to the moment in which all flow entries were installed: it never took more than 0.8s to install flow entries on all datapaths in group b) topologies; results for group c) topologies are in column "Convergence time" of Table 3.2. We then estimated the time required to recompute paths and install new flow entries after link failures, which was always below 0.3s both for group b) topologies and for group c) topologies (column "Recovery time" in Table 3.2). Considering the limits imposed by the emulation, we deem these figures rather promising for a production deployment.

3.6 Open Problems

Although our language and approach are readily applicable, our research directions envision several improvements. Support for traffic engineering, albeit envisioned, is currently very limited: more elaborate routing policies could be introduced, and traffic could be dynamically re-routed depending on target QoS levels. Moreover, an incremental handling of installed flow entries is desirable, to support configuration changes as well as addition, removal, and migration of customer sites with minimal disruptions. A mechanism to learn customer IP prefixes, possibly by integration with existing protocols like BGP, should also be introduced. In addition, the language could be extended to support services such as Virtual Private LAN Service (VPLS). Finally, improvements to our prototype implementation and experiments on hardware testbeds or operational networks as well as performing further experiments aimed at better evaluating scalability are likewise part of our work plan.

Chapter 4

Taking Care of ARP in a Software Defined Network [*]

The Address Resolution Protocol (ARP) enables communication between IP network nodes by reconstructing the association between the IP address of an interface and its MAC address. In a Software Defined Network (SDN) this mechanism is not needed, because the controller knows the topology and each datapath forwards packets based on their contents, without requiring additional addressing information.

We tackle the interoperability problem that arises between legacy network devices, that rely on ARP (end systems, as well as non-SDN routers), and SDN datapaths, where network protocols are replaced by the controller’s logic. In particular, we propose the design of a controller that supports ARP traffic exchange with legacy devices, thus enabling end-to-end communication between IP network portions separated by a SDN. Our controller installs a minimal set of flow entries in datapaths and confines ARP traffic to the edge of the SDN, thus reducing its overhead. We also discuss a possible way to support the neighbor discovery functions of ICMPv6 and argument about the applicability of our approach, which is confirmed by experiments performed on SDN switches from a range of different vendors.

[*]Part of the material presented in this chapter is based on the following publication: R. di Lallo, G. Lospoto, M. Rimondini, G. Di Battista. How to Handle ARP in a Software-Defined Network. In *Proc. NetSoft*, IEEE, 2016

4.1 Introduction

Software Defined Networking (SDN) is the new standard approach to the design and implementation of network functionalities. At an intermediate deployment stage, interoperability between SDN and legacy network devices is essential (it is a concern for about 35% of the companies surveyed in [78]). This especially applies to basic mechanisms such as associating the IP address of a host with its MAC address: the Address Resolution Protocol (ARP), which implements this function and is fundamental for the operation of IP networks, becomes useless in a pure SDN because the controller knows the complete topology, and datapaths forward packets based on their contents, without the need for additional addressing information.

We consider the problem of supporting ARP in a network where legacy and SDN-enabled devices coexist. Arguably, this is a very practical problem: in fact, the vast majority of end systems will continue to use IP for a reasonably long time, and their reciprocal connectivity must be ensured even if their provider network is fully migrated to SDN. We propose the design of a controller that addresses this problem and has the following features: it requires very few message exchanges with the datapaths and it installs a minimal number of flow entries; it confines ARP traffic to the edge of the SDN, avoiding the overhead of transporting it through a potentially large infrastructure; and it supports ARP resolution in the presence of multiple IP subnets. We rigorously specify the controller's logic using pseudocode and illustrate an extension of our approach to support ICMPv6 neighbor discovery functions.

The rest of the paper is organized as follows. In Section 4.2 we review the relevant state of the art. In Section 4.3 we describe how ARP is usually handled by existing SDN controller frameworks. In Section 4.4 we introduce our proposed design for an ARP-aware SDN controller. The practical applicability of our approach is discussed in Section 4.5, and confirmed by experimental tests described in Section 4.6. In Section 4.7 we draw conclusions and propose lines for future work.

4.2 Related Work

The problem of handling ARP traffic in a SDN is of course well recognized and widely discussed. A recently published patent [79] claims a method for implementing an OpenFlow controller that can handle ARP requests. However, this method has limited scalability, because the controller floods ARP requests throughout the SDN when the target MAC address is not known; it only supports a single IP subnet; and it offloads any other tasks (e.g., neighbor unreachability detection) to the hosts, without taking full advantage of the potential of SDN. A more detailed and technical description of how to deal with ARP traffic is in [80]. However, since the paper focuses on an Internet Exchange Point network, it still considers a single IP subnet. Moreover, some choices in the controller's logic are tailored to the context-specific problem of preventing storms of ARP broadcasts towards dead routers. Last, ICMPv6 neighbor discovery mechanisms are intentionally left out. In [81] the authors advocate the organization of SDN controllers into composable functional modules, and briefly discuss possible ways to handle ARP traffic. While we share the architectural vision, we improve the proposed ARP handling mechanisms by reducing the impact of flooding and by limiting the processing of ARP packets to the edge of the SDN. A related paper [82] describes a data flow that comprises exchange of ARP packets: compared to this, our approach is more general and adopts a more structured splitting of the controller's functional modules. ARP traffic handling is also discussed by [83] in the context of multi-tenant data centers. However, since the proposed solution aims at attaining isolation among tenants, it focuses on techniques (e.g., MAC addresses rewriting) which are not useful for our goals.

A working group within the Open Networking Foundation (ONF) is proposing guidelines to migrate existing network services to SDN. However, the technical documents produced so far (see, e.g., [69, 84]) describe general methods and omit most technological details.

From a technological point of view, several frameworks for developing SDN controllers (e.g., Ryu [74], Floodlight [85], POX [86]) include basic ARP traffic management capabilities, which we analyze in Section 4.3.

We are not aware of any contributions discussing the operation of ICMPv6 neighbor discovery functions in the presence of an SDN.

4.3 ARP Traffic Handling in Current Controllers

Several possible approaches can be pursued to handle ARP traffic in a SDN. A simple solution consists in proactively installing a flow entry on all the datapaths that instructs them to process ARP packets using the standard networking stack. However, this choice has several drawbacks: it requires support for the NORMAL [8] reserved output port, a feature that all versions of the OpenFlow specification declare as optional; it increases the overhead of ARP traffic, which needs to be broadcast over a potentially large SDN; and it introduces a dependence on legacy technologies, which pure OpenFlow datapaths cannot satisfy.

In order to illustrate more advanced approaches, we refer to the scenarios in Figure 4.1. In this figure squares represent switches (S1, S2, ...) and routers (R1, R2), circles represent end systems (hosts H1, H2, ...), lines represent physical connections (possibly involving standard 802.1D switches), and labels net_i represent IP subnets. Nodes enclosed in gray clouds are SDN-enabled datapaths, while the other nodes represent IP devices.

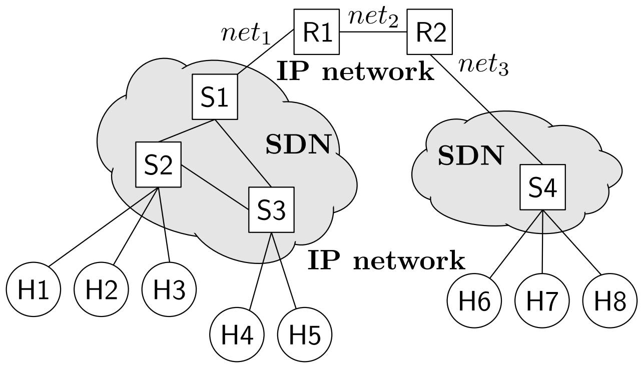
Most SDN-based approaches and technologies to support ARP traffic consider scenario (a), where every SDN independently handles ARP traffic for a single IP subnet. In particular, this involves the following tasks.

A backward learning mechanism in the controller implements the standard IEEE 802.3 learning process: it looks at the source MAC address `src_mac` of every packet received by a datapath *dpid* and keeps a correspondence between `src_mac` and the port from which *dpid* received the packet. In this way, the controller learns the position of `src_mac` and can instruct each datapath to forward packets directed to that host out of the most appropriate port. Usually, this mechanism is only triggered for the first packet of a flow, for which datapaths have no matching flow entries. Most existing SDN controller frameworks [74, 85, 86] include a backward learning mechanism as a readily usable function.

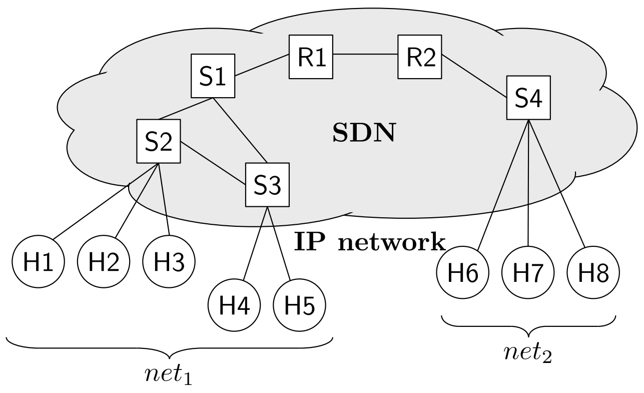
An additional ARP forwarding mechanism, often integrated with backward learning, is specifically responsible for handling ARP packets. We describe the implementation of this mechanism that is natively available in POX [86] (other controller frameworks adopt a similar approach). Consider Figure 4.1a, assume that the flow tables of all datapaths are empty, and suppose datapath S2 receives an ARP request from host H1 asking for the MAC address of target host H5. S2 then sends the packet to the controller, which:

- learns the association between the MAC address of H1 and the port of S2 to which it is attached;
- instructs S2 to flood the packet (i.e., send it out of all the ports except the ingress one), without installing flow entries.

The packet reaches the neighboring datapaths S1 and S3, which again send it to the controller. Similarly, the controller instructs S1 and S3 to flood the packet. The process continues until the packet reaches H5, which sends an



(a) Scenario where each SDN handles ARP traffic for a single IP subnet.



(b) Scenario where a SDN handles all ARP traffic regardless of the IP addressing plan.

Figure 4.1: Our reference network scenarios, consisting of switches (S1, S2, ...), routers (R1, R2), and hosts (H1, H2, ...). Devices enclosed in gray clouds are SDN-enabled.

ARP reply to S3. Since S3 has no flow entries that match the packet, it sends it to the controller, which accomplishes the following tasks:

- it learns the association between the MAC address of H5 and the port of S3 to which it is attached;
- by reconstructing the sequence of datapaths traversed by the ARP re-

quest, it determines a forward and reverse path between H1 and H5;

- it installs flow entries on each datapath along these two paths, which specify how to deliver the ARP reply sent by H5 to H1 and how to forward any future unicast ARP packets (e.g. gratuitous ARP packets) without flooding them. The match condition of these flow entries checks the Ethernet type (it must be `0x806` – ARP) as well as the destination MAC address and IP address in the ARP payload, while the action forwards packets using a port consistent with the path computed by the controller.

This mechanism has several shortcomings. First of all, ARP packets are transported all the way to the target host: this limits the extent of the SDN to a single LAN, because the source and the target of ARP packets must be on the same subnet. Figure 4.1a depicts exactly this scenario, where each SDN only handles ARP traffic within a specific subnet net_i. A data packet directed to a different LAN must either go through an IP router and then enter another SDN (like in Figure 4.1a), or reach an SDN datapath that accomplishes the tasks of a router (e.g., it must reply to an ARP request for the default gateway), whose implementation involves a number of different functions (see also [81]).

On the other hand, ARP requests need to be broadcast over a potentially large topology, introducing a forwarding overhead. To mitigate this, some controller frameworks [86] natively offer a layer 3 learning mechanism that acts as an ARP cache: it maintains associations between the IP address and the MAC address of each host. When a datapath S1 receives an ARP request for a target host H2, the controller checks whether the MAC address of H2 is already known and, if so, instructs S1 to immediately send an ARP reply without further propagating the request. However, this mechanism is still able to operate on a single subnet only.

Moreover, the above illustrated ARP forwarding approach does not work in the presence of loops, because broadcast ARP requests would cycle forever. Some controller frameworks [86] address this problem by running a Spanning Tree Algorithm in advance, but this may prevent blocked ports from sending any types of traffic. Our solution is completely free from this issue because it keeps ARP traffic confined to the edge of the SDN.

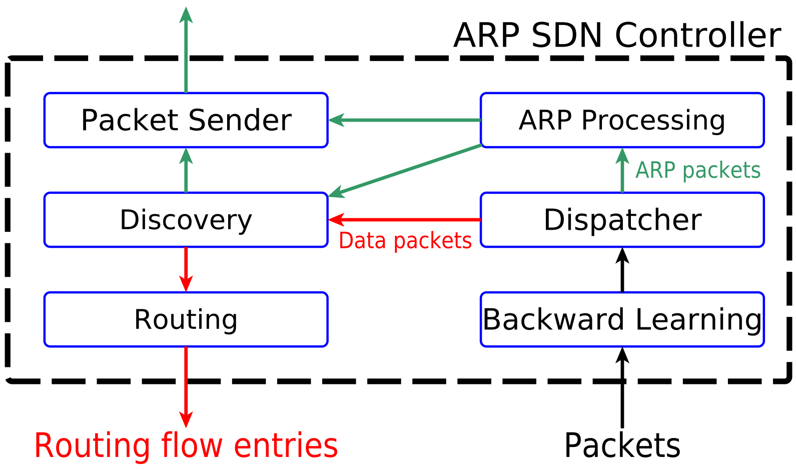


Figure 4.2: Architecture of our SDN controller to handle ARP.

4.4 A Novel Design of an ARP-aware Controller

We now illustrate the architecture and operation of an SDN controller that is able to support ARP traffic exchange with legacy end systems (or routers) in a scalable and flexible way. The description in Section 4.3 demonstrates that even simple approaches can involve several little tasks. Therefore, also inspired by [81], we organize our controller into functional modules, as shown in Figure 4.2.

Those modules cooperate as follows. When a packet reach the controller inside a PacketIn, it is processed by the Backward Learning module; after that, this module invokes the Dispatcher module which triggers ARP Processing module or Discovery module based on the content of the processed packet's header. If PacketIn carries an ARP packet, then the ARP Processing module, which holds an arp cache containing all associations between IP and MAC addresses, is triggered (as depicted by green arrows in Figure 4.2); otherwise in presence of a data packet (e.g. IP packet) the Discovery module is invoked (red arrows in Figure 4.2). In the former case, if no entries are found in the arp cache, ARP Processing module cooperates with the Discovery module in order to correctly locate the destination, whose IP address will be retreived exploting the information contained in the data packet's header; this task is accomplished by invoking the SendPacket module. After determining the location of the destination, the Discovery module cooperates with Routing module in order to compute paths, and send to each datapath in the network a set of flow entries, allowing data packets to reach the desired destination. Note that just the information contained in the header of the packets are used by modules, in order to correctly handle specific task (e.g. handling ARP packets sent by hosts).

In order to clearly illustrate our controller, we will refer to several algorithms, which specify its ARP-specific operational logic using pseudocode. Procedure PACKETRECEIVED (Alg. 2) is the entry point, and is executed when datapath *dpid* sends to the controller a PacketIn message containing a packet *pkt* that *dpid* has received through port *inport*.

Algorithm 2 Algorithm to identify the type of packet.

```
1: procedure PacketReceived(dpid, inport, pkt)
2:     BackwardLearn(dpid, inport, pkt.src_mac)
3:     if pkt.eth_type==0x806 then ProcessARP(dpid, pkt)
4:     else if pkt.eth_type==0x800 then                          ▷ IP packet
5:         destdpid ← DiscoverHost(pkt.ip_header.dst_ip)
6:         FindPath(pkt.ip_header.src_ip, pkt.ip_header.dst_ip, dpid, destdpid)
7:     end if
8: end procedure
```

Our controller consists of different modules, each having in charge a specific task. In the following, we report a detailed description of each module, jointly to the algorithms used by those components.

Backward Learning Module

The backward learning module is described in procedure BACKWARDLEARN (Alg. 3). As also said in Section 4.3, it is triggered for every packet *pkt* received by a datapath *dpid* from a host with MAC address *macaddr*. For each datapath, it maintains an association between the MAC address of each host and the datapath port it is connected to (map *outport* at line 2). For convenience, we also maintain an association between each MAC address and the datapath it is attached to (map *datapath* at line 3).

Algorithm 3 Backward learning algorithm.

```
1: procedure BackwardLearn($dpid$, $port$, $macaddr$)
2:     $outport[dpid][macaddr] \leftarrow port$
3:     $datapath[macaddr] \leftarrow dpid$
4: end procedure
```

ARP Processing Module

ARP processing module is described in procedure PROCESSARP (Alg. 4). It handles every ARP packet *pkt* received by a datapath *dpid* at the edge of the SDN, namely a datapath connected to IP-speaking node (e.g. a host). To support this module, a flow entry must be installed on edge datapaths, instructing them to forward all ARP packets to the controller. Exploiting the information in these packets it keeps an association between every IP address and the corresponding MAC address (map *arpcache* at lines 2 and 4). In addition, it immediately replies to ARP requests coming from hosts by sending a forged ARP reply with a fixed fake MAC address M (lines 6-13). This is enough to make a host start sending data packets. This choice is motivated by the following considerations: i) we want to quickly reply to the host that performed the ARP request, without waiting for delay introduced by further communications (e.g. the time spent to the controller in retrieving the actual MAC address of the destination); ii) we want to obtain the IP address of the destination as soon as possible, in order to correctly compute a routing. This is only possible looking at data packets.

It is important to observe that address M can be chosen arbitrarily, and will never be seen by any hosts but the one that sent the ARP request. Possible conflict-free choices for M are the controller's MAC address or a reserved MAC address such as `ff:ff:ff:ff:ff:fe`. By applying this mechanism, ARP traffic is confined to the edge of the SDN and never reaches any internal datapaths.

IP-speakers Discovery Module

It tracks the location of IP addresses, namely the datapath and port to which each IP-speaking node is connected. Such information is partially gathered by the ARP processing module, and partially derived from: 1) passive traffic monitoring (e.g., by looking at the first packet of each flow, which is always delivered to the controller), 2) static configuration, indicating which IP

Algorithm 4 Algoritm to process ARP packet at the controller.

```
1: procedure ProcessARP(dpid, pkt)
2:     arpcache[pkt.data.src_ip] ← pkt.data.src_mac
3:     if pkt.data.arp_opcode==2 then                          ▷ ARP reply
4:         arpcache[pkt.data.dst_ip] ← pkt.data.dst_mac
5:     else if pkt.data.arp_opcode==1 then                     ▷ ARP request
6:         arpreply ← new ARP reply packet
7:         arpreply.data.src_ip ← pkt.data.dst_ip
8:         arpreply.data.src_mac ← $M$
9:         arpreply.data.dst_ip ← pkt.data.src_ip
10:        arpreply.data.dst_mac ← pkt.data.src_mac
11:        arpreply.eth_header.src_mac ← arpreply.data.src_mac
12:        arpreply.eth_header.dst_mac ← arpreply.data.dst_mac
13:        SendPacket(dpid, arpreply)
14:    end if
15: end procedure
```

subnets are connected to each datapath, and 3) probing mechanisms such as ICMPv6 Neighbor Unreachability Detection (NUD), which refresh the visibility status of each host. If this piece of information is available from maps *arpcache* and *datapath*, it is immediately returned (line 14). This module also takes care of reconstructing the MAC address of a destination IP node (required to make the node accept the received packets), by issuing ARP requests either from a single datapath, if the recipient's location is known, or from the datapaths that are attached to the recipient's IP subnet (known by configuration). Note that also this ARP exchange is confined to edge datapaths. ARP requests can use M as the source MAC address and an arbitrary IP (e.g., the controller's IP to avoid collisions) as the source IP address. We verified that, despite these manipulations, hosts successfully reply to such ARP requests. Although not explicitly described, this module may also be responsible for reconstructing the network topology using LLDP packets, in the usual way envisioned with OpenFlow (see also [87]).

Routing Module

Routing module is described in procedure FindPath (Alg. 6). Once the ARP request-ARP reply exchange has been completed and a host with IP address

Algorithm 5 Algoritm to discovery hosts in the network.

```
1: function DiscoverHost(ipaddr)
2:   if arpcache does not have key ipaddr then
3:     for each datapath dpid that is attached to a subnet containing ipaddr do    ▷ Known by configuration
4:       arpreq ← new ARP request packet
5:       arpreq.data.src_ip ← controller's IP address
6:       arpreq.data.src_mac ← M
7:       arpreq.data.dst_ip ← ipaddr
8:       arpreq.eth_header.src_mac ← arpreq.data.src_mac
9:       arpreq.eth_header.dst_mac ← ff:ff:ff:ff:ff:ff
10:      SendPacket(dpid, arpreq)
11:    end for
12:    Wait until the association arpcache[ipaddr] is learned
13:  end if
14:  return datapath[arpcache[ipaddr]]
15: end function
```

src_ip starts sending data packets directed to another host *dst_ip*, this module computes a path P from datapath *origin_dpid*, (entry point to the SDN) to datapath *dest_dpid* (to which host *dst_ip* is attached), based on an arbitrary routing policy and exploiting the knowledge of the network topology acquired from the discovery module. The routing module then installs flow entries on all datapaths along P, instructing them about how to route packets directed to *dst_ip* (any data packets that have been buffered at *origin_dpid* are automatically forwarded at this point). For convenience, the module also installs flow entries to route packets from *dst_ip* to *src_ip* along the reverse path. In addition, it also installs on *dest_dpid*, the last datapath along P, a flow entry that overwrites the MAC address of every packet directed to *dst_ip* with the actual MAC address corresponding to *dst_ip* (line 6). This is required because packets travel in the SDN with the fake MAC M as the destination address, and end systems would therefore not accept them. The same rewriting rule is installed at *origin_dpid* for packets going to *src_ip*.

SendPacket Module

Send packet module is described in procedure SendPacket (Alg. 7). It is a utility procedure to ask datapath *dpid* to send a packet *pkt* to its intended

Algorithm 6 Algoritm to compute a path in the network.

```
1: procedure FindPath(src_ip, dst_ip, origin_dpid, dest_dpid)
2:     Compute a path $P$ from origin_dpid to dest_dpid
3:     for each datapath dpid along $P$ do
4:         Use a FlowMod to tell dpid how to handle traffic from src_ip to dst_ip and from dst_ip to src_ip
5:     end for
6:     Use a FlowMod to tell dest_dpid to write arpcache[dst_ip] in the destination MAC of packets directed to dst_ip. Do the same with dest_dpid, dst_ip replaced by origin_dpid, src_ip
7: end procedure
```

destination, choosing the most appropriate output port.

Algorithm 7 Algorithm to produce a PacketOut sending to the datapath.

```
1: procedure SendPacket(dpid, pkt)
2:     if pkt.eth_header.dst_mac== ff:ff:ff:ff:ff:ff or outport[dpid] does not have key pkt.eth_header.dst_mac then
3:         Use a PacketOut to tell dpid to send pkt out of all ports
4:     else
5:         Use a PacketOut to tell dpid to send pkt out of port outport[dpid][pkt.eth_header.dst_mac]
6:     end if
7: end procedure
```

ARP Processing Example

We now run through a complete example of how ARP packets would be handled by the above described modules, using Figure 4.1b as a reference. In the figure, R1 and R2 are SDN switches: they are still identified as routers only because we assume that they may still run an IP stack in order to support in-band communication between switches and controller, but they do not accomplish any particular function in SDN apart from forwarding packets. Suppose datapath flow tables are empty and host H1 in subnet net_1 wants to send traffic to host H7 in subnet net_2. At first H1 broadcasts an ARP request asking for the MAC address of its default gateway, for example R1. S2 immediately sends back an ARP reply specifying the fake MAC M, so that H1 can start sending traffic. Once the first IP packet from H1 is received by S2, the controller can determine that the ultimate destination of this packet is H7. The packet is therefore held in a buffer at S2 while a broadcast ARP request asking for H7's MAC address is sent out of S4 (which, by configuration, is known to be connected to the target subnet net_2). After H7 sends an ARP reply to S4, the controller computes a path from S2 to S4 (for example, $\langle \mathsf{S2}, \mathsf{S1}, \mathsf{R1}, \mathsf{R2}, \mathsf{R4} \rangle$) and installs on all the datapaths along this path flow entries that instruct them about how to forward IP traffic from H1 to H7 and back. Other flow entries instruct S4 to write H7's MAC address in packets directed to H7 and S2 to write H1's MAC address in packets directed to H1. At this point the packet(s) buffered at S2 are sent and the communication between H1 and H7 can proceed.

With respect to the example just described, we want to remark that ARP traffic is bounded at the edge of the network. In fact, using our method, datapaths S1 and S3 will never be reached by any ARP traffic unlike other SDN controller implementations discussed in Section 4.3 or traditional ARP implementations.

4.5 Applicability Considerations

In this section we discuss the scalability and practical applicability of our approach, and present possible alternatives for the realization of specific modules.

Scalability

We argue that our approach is very scalable, for several reasons. First of all, our controller installs a minimal set of flow entries: all datapaths only have a pair of rules for handling forward and reverse traffic between every pair of end systems. Datapaths at the edge of the SDN have just one extra flow entry for each attached IP speaking interface, used to rewrite the fake MAC address with the actual one of that interface. Capacity constraints of flow tables should therefore not be an issue. For the case of very large networks, flow entries can be suitably distributed in order to meet such constraints: for example, MAC rewriting rules could be moved from edge datapaths to any other datapaths along the path between two hosts, and different traffic flows that share a common subpath could be aggregated to reduce the number of flow entries used to route them.

In addition, controller and datapaths only communicate during the ARP request-ARP reply exchanges and when the first packet of a flow is sent by an IP host: any subsequent packets are handled by the flow entries installed on datapaths, without further communication. Indeed, the controller never receives PacketIn messages from internal datapaths (i.e., those that are not connected to any IP nodes), because they already know how to forward traffic thanks to the flow entries that are proactively installed by the routing module.

The discovery module floods ARP requests from every datapath that is connected to the target subnet, and this could have an impact on network performance and utilization. As a simple alternative solution, such packets could be selectively sent out of the datapath ports that are known to be attached to hosts (in principle, these ports could be detected because they do not receive any LLDP packets during the topology discovery process). The impact of broadcast ARP traffic can also be reduced with more advanced approaches like SEATTLE [88], which takes advantage of a DHT-based directory service, or Enhanced Lookup (ELK) [89], a technique which involves reassigning MAC addresses to organize them hierarchically, for which an OpenFlow-based implementation has already been proposed in [90].

Traffic Engineering and Network Dynamics

Our approach can be improved in order to accommodate basic traffic engineering requirements. As a preliminary consideration, it is possible to differentiate the forward and reverse paths computed by the routing module between a pair of hosts.

Furthermore, our controller always replies to ARP requests with a fixed fake MAC address M. As a consequence, this address appears as the destination MAC in all traffic packets sent by the hosts, and it could be exploited as a label to drive traffic engineering decisions. For example, packets that a datapath receives from a host through a specific input port could be assigned a different fake MAC M^* to reflect the fact that they must be routed along a different path in the SDN. Collisions with existing MAC addresses must of course be avoided, and this can be achieved by using reserved MAC ranges or by querying for the availability of a MAC address using, e.g., Reverse ARP packets.

In order to support network dynamics (e.g., host connections or disconnections), learned MAC addresses as well as the association between IP addresses and MAC addresses must be kept up to date. Our controller is already designed to refresh the applicable data structures when new information are received. However, the discovery module can be improved to trigger such refresh by periodically sending ARP requests to each known host, a mechanism that is similar to the ICMPv6 NUD function. A change in the reachability of a host may require the routing module to compute a new path (if available) and to update the installed flow entries accordingly. A change in the MAC address of a host requires an update of the sole flow entry that rewrites the fake MAC with the actual MAC address of that host.

Processing of ICMPv6 Neighbor Discovery Packets

With the introduction of IPv6, many network diagnostic and support functions have been delegated to ICMPv6. This includes the association of an IPv6 address with a MAC address, which is realized by the ICMPv6 neighbor discovery mechanism. This mechanism uses ICMPv6 neighbor solicitation and neighbor advertisement packets in a much similar fashion as ARP requests and replies. There are only two main differences with respect to ARP (besides the packet format, of course), which require adaptations in our discovery module. The first is that neighbor solicitations use multicast destination addresses, which the source host (in our case, the controller) must suitably compute. The second one is that ICMPv6 has Neighbor Unreachability Detection (NUD) mechanisms, which involve periodically sending neighbor solicitations to the host of interest and monitoring traffic coming from that host. In Section 4.5 we discuss how our controller can support similar mechanisms even for IPv4, for tracking the reachability of a host (including, e.g., the case in which its position has changed) or changes in its MAC address.

4.6 Experiments

In order to verify the applicability of our approach on real-world devices, we used the Ryu framework [74] to implement a prototype SDN controller according to the design described in Section 4.4. We then ran experiments using OpenFlow 1.3 compliant switches from 3 different vendors, which in the following we call *A*, *B*, and *C* because of NDA constraints.

Testbed Description

Our experiments were mainly targeted at assessing the compatibility of our controller design with existing devices and at performing interoperability tests. We therefore considered the following simple linear topology with two hosts and two real OpenFlow-enabled switches: $\boxed{\text{host1 — dp1 — dp2 — host2}}$. We instantiated dp1 and dp2 with all the possible combinations of the available switches, namely ($\text{dp1} = A, \text{dp2} = B$), ($\text{dp1} = A, \text{dp2} = C$), and ($\text{dp1} = B, \text{dp2} = C$). host1 and host2 were Ubuntu 14.10 machines with IP addresses in different subnets (respectively, `10.0.0.0/24` and `20.0.0.0/24`). Each host had a default gateway set, pointing to an arbitrary IP address. Controller-switch communication was realized out of band. We used `ff:ff:ff:ff:fc:ac` as the fake MAC address M.

We verified the correct handling of ARP traffic by the controller using a simple ping. In particular, we checked that the following steps were successfully accomplished:

- ARP caches of the hosts were populated with the fake MAC;
- ARP packets were kept on the edge of the SDN; and
- hosts exchanged ICMP packets.

Experimental Results

All the pings in our tests successfully worked. We put special attention in verifying the interoperability among different vendors and the transparency of our approach for end systems, by verifying that:

1. ARP requests generated by the source host were correctly sent to the controller via PacketIn messages;
2. datapaths correctly delivered forged ARP replies containing the fake MAC address M to the requesting host, when instructed to do so via PacketOut messages;
3. hosts accepted and cached the contents of the forged ARP replies;
4. datapaths correctly installed flow entries for the forwarding of ICMP traffic, and these entries were correctly matched;
5. datapaths correctly installed MAC address rewriting flow entries, and the MAC address of ICMP packets was actually rewritten as expected.

Based on the above described observations we conclude that, even in the presence of devices from multiple vendors and in a scenario involving different IP subnets, our approach for handling ARP traffic is effective.

4.7 Open Problems

Consolidating our proposed design requires further investigation on several aspects. As discussed in Section 4.5, fake MAC addresses could be exploited to distinguish traffic classes and route them distinctly throughout the SDN. Improving support for IPv6 and, in particular, ICMPv6 mechanisms (e.g., router solicitation, router renumbering, authentication and encryption) poses interesting challenges: for example, path MTU discovery is difficult to implement with plain OpenFlow. Some information about the network configuration (e.g., IP subnets attached to each datapath) could be derived by legacy routing protocols (e.g., OSPF) that may be running to support in-band communication between the switches and the controller. Our prototype controller implementation can be improved and further tests on more complex topologies could be performed.

Chapter 5

Experimental Evaluation of SDN Devices [*]

Software-Defined Networking (SDN) is a de-facto established approach that separates the packet switching functions of a device from its operational logic, which is controlled by a piece of software. Due to its potential for realizing new network architectures and services, a whole thread of scientific literature is devoted to SDN and its most adopted incarnation, OpenFlow. However, limited attention has been put in verifying the viability of the proposed approaches on currently available hardware.

We address this deficiency through the following contributions: i) a critical review of the literature about SDN in terms of applicability issues stemming from publicly documented limitations of OpenFlow implementations; ii) a methodology for verifying the support of SDN-related functionalities in a network device, comprising an OpenFlow compliance test as well as custom targeted tests; iii) an application of the methodology to devices from 7 different vendors, unveiling extensive anomalous behaviors affecting even the most basic features; iv) a discussion of this outcome in terms of relevance of the discovered anomalies and of their implications on the applicability of state-of-the-art contributions on SDN. Besides taking a snapshot of the viability of research results, with this chapter we intend to highlight aspects that operators should consider when picking SDN devices.

[*]Part of the material presented in this chapter is based on the following publication: R. di Lallo, M. Gradillo, G. Lospoto, C. Pisa, M. Rimondini. On the Practical Applicability of SDN Research. In *Proc. NOMS*, IFIP/IEEE, 2016.

5.1 Introduction

Software-Defined Networking (SDN) has been the future of network architectures for a few years, feeding multiple research areas with unprecedented challenges and being increasingly adopted by device vendors. Even now that it is the present, activities on the scientific as well as the technological side of SDN are still fervent, due to the unmatched infrastructure scalability, improved flexibility of management, and vendor independence it brought about.

Despite being a hot topic in the research community, most research contributions on SDN validate the novel network architectures and services they propose on ad-hoc testbeds, with little attention to their practical viability using currently available devices. On the other hand, even if OpenFlow is now somewhat mature, vendors seem to lag behind in terms of supported functionalities. Preserving this dual (scientific and technological) perspective, the main goal of this paper is to take a snapshot of the applicability of state-of-the-art contributions that leverage SDN on network devices that are currently available on the market. We first accomplish this based on publicly documented vendor-specific limitations of OpenFlow implementations. After that, we introduce a testing methodology to assess the level of support of OpenFlow functionalities on a device, comprising custom targeted tests besides a standard compliance test. We use this methodology to assess the OpenFlow implementation progress on a range of commercially available devices[†] manufactured by 7 major vendors, revealing many unexpected anomalies that affect even basic functionalities. Finally, we discuss the outcome of our tests in terms of relevance of these anomalies (e.g., whether they are due to hardware or software flaws) and of their impact on the applicability of SDN research contributions. In addition, we compile a catalog with some of the issues we experienced during our tests, which can help network administrators in making more informed decisions on what aspects to care about when choosing the devices for a network infrastructure. To our knowledge, this is the first work that relates results from the research community with OpenFlow functionality tests, and that analyzes in detail the outcome of such tests.

The rest of the chapter is organized as follows. In Section 5.2 we review a selection of the state of the art on SDN and assess how its applicability is impacted by vendor-declared limitations. In Section 5.3 we introduce our methodology to check the support of SDN-related functionalities in a device. In Section 5.4 we document the outcome of applying the methodology on

[†]None of the devices we analyzed appears in the lists of certified devices available at [91, 92].

a range of commercially available devices, while in Section 5.5 we illustrate some of the observed abnormal behaviors. In Section 5.6 we discuss the implications of our test results on the literature on SDN. In Section 5.7 we draw conclusions and directions for future work.

5.2 A Review of SDN Literature

SDN has been attracting the interest of the research community for at least a decade, resulting in a very rich body of contributions. In this section we focus on a selection of the most recent ones that we consider most directly impacted by the accuracy of OpenFlow implementations.

First of all, there is a class of papers that address the scalability issues induced by the limited capacity of TCAM memories, where flow entries are typically stored [93, 94, 95, 96, 97, 98]. These papers aim at saving space in the flow tables by replacing existing flow entries with more compact, equivalent entries. In particular, [93] proposes an algorithm for implementing forwarding policies by distributing the associated flow entries on multiple datapaths along selected routing paths. In [98] a slightly less strict approach is adopted, since constraints on traffic paths are relaxed when flow table capacities are exceeded. In [95, 96, 97, 99] the authors propose methods for massively compressing flow tables using wildcards, whereas the flow table decomposition approach in [94] pursues the opposite strategy, because match conditions are manipulated to reduce the number of wildcard bits.

Another relevant class of papers [100, 101, 102, 103] addresses the practical problems involved in deploying SDN in specific application scenarios. In [100] the authors propose a solution for deploying SDN inside an Internet eXchange Point (IXP). They define a fully SDN-based architecture where each network connected to the IXP can independently specify high-level routing policies, which are combined together and translated to flow entries to be installed on the switches. In [101] the authors propose a pure SDN approach for realizing Virtual Private Networks (VPNs) based on Multi-Protocol Label Switching (MPLS). They describe how VPN configurations, expressed in a centralized high-level specification, are translated into flow entries which make use of MPLS-specific operations for traffic forwarding. SDN is also used in [102] to realize an inbound traffic engineering solution, which is based on altering the source address of outbound IP packets. Last, in [103] the authors propose an OpenFlow-based architecture for a wireless mesh network capable of reacting to failures.

Moreover, there is a thread of contributions aimed at introducing languages for an abstract specification of packet forwarding policies (e.g., [104, 105, 106]), which also constitute a building block for novel network architectures (e.g., [100]): these contributions include run-time systems that translate policy specifications to flow entries installed on datapaths. A plenty of additional research results on SDN-related topics appeared in the last years, but it is out of the scope of this chapter to review all of them (for a survey on SDN, please see [107]). Moreover, they mostly focus on fault tolerance, live migration, and network architectures: as such, they usually consider a higher level of abstraction and do not directly deal with the installation of flow entries on the switches.

At the time of writing, all the major vendors have been including SDN-enabled switches in their device offer for quite a long time and have adopted the OpenFlow specification (possibly side-by-side with a proprietary SDN implementation), thus enabling the deployment of many approaches proposed in the literature. However, despite the fact that OpenFlow has been around for at least 6 years, limitations still exist in vendor implementations that restrict the applicability of these approaches. We identified a selection of top OpenFlow switch vendors for which these limitations are documented in publicly available user manuals. In random order, they are: HP [108], Dell [109], Brocade [110], Arista Networks [111], and Extreme Networks [112]. Please note that these vendors are not necessarily related with those considered in the tests described in Sections 5.4 and 5.5. We isolated the most relevant and frequently occurring limitations and associated them with the vendors declaring them: the results are in Table 5.1.

The capacity of flow tables obviously depends on hardware constraints, but may be further restricted due to fixed partitioning schemes of the internal memory used to implement them (see, e.g., [110]). Match conditions are often restricted to comply with predefined patterns (see, e.g., [112]): for example, matching on MAC addresses may not be permitted when considering ICMP packets. Moreover, their structure influences memory consumption, affecting flow table capacities (see, e.g., [109]): for example, efficiently representing wildcards in match conditions requires dedicate TCAM memory slots [113], which are often available in very limited quantities, and some datapaths (see, e.g., [110]) enforce limits on the number of flow entries that do not match the input port. Some vendors support a single flow table, thus limiting scalability and making it difficult to implement certain use cases [114]. Support for MPLS is often minimal or absent, limiting the possibility to implement related network services. The coexistence of OpenFlow and traditional layer-2/layer-3 protocols on the same ports (*hybrid-port mode*) may be prohibited, and the

Table 5.1: Common limitations in OpenFlow implementations and vendors declaring them.

Limitation	HP	Dell	Brocade	Arista Networks	Extreme Networks
Flow table size constraints		●	●		
Restricted structure of match conditions		●	●		●
Single flow table				●	●
Lack of MPLS support	●		●	●	
Hybrid-port mode unsupported			●	●	
NORMAL reserved port unsupported			●		●

NORMAL reserved port number (used to process a packet using the traditional non-OpenFlow networking stack) may be unsupported: the lack of the latter two features poses significant limits on realizing in-band communication between the datapaths and the controller.

To the extent of our knowledge, the level of awareness of such limitations in the literature about SDN is still modest, thus potentially affecting the applicability of some promising approaches on real devices. In particular, papers aimed at reducing flow table sizes [93, 94, 95, 96, 97, 98] ignore the restrictions imposed on match conditions, thus generating potentially unusable flow entries. From another point of view, honoring vendor-imposed restrictions on match conditions may invalidate flow table compression strategies. Usage of wildcards in flow entries can reduce the capacity of flow tables by one or two orders of magnitude (see, e.g., [109]), a problem that affects [95, 96, 97] but not [94] (which still is not immune from potential violations of other vendor restrictions). The memory requirements of using wildcards are also discussed in [115] where, unlike the aforementioned works, the authors evaluate their approach using hardware switches: not surprisingly, they suggest usage of exact match conditions whenever possible. Application of SDN to IXPs is also impacted: according to [100], flow entries that support control traffic should match layer-4 packet header fields, implying a drastic reduction in the number of installable flow entries. The proposed architecture may still be considered deployable under the realistic assumption that an average-sized IXP does not have more than ~ 500 participants, in terms of Autonomous Systems. The VPN architecture in [101] requires support for handling MPLS labels, making the approach inapplicable with devices from certain manufacturers (see, e.g., [108]). Both [100] and [101] do not address how in-band communication between the controller and the datapaths can take place. At least, this requires support for the NORMAL reserved port and for hybrid-port mode, two features that are assumed to be available even in the wireless mesh network scenario in [103] and that influence the applicability of these approaches. Finally, limits on the number of flow entries that match IP addresses (see, e.g., [109]) may affect the traffic engineering approach in [102].

5.3 Device Testing Methodology

Vendor-declared limitations in OpenFlow implementations cause evident issues in deploying certain approaches proposed in the literature about SDN. However, there may be further undocumented constraints that also impair the viability of these approaches, which even vendors may not be aware of. With the goal of revealing such constraints, in this section we define a methodology to assess the level of support of SDN-related functionalities in a datapath. The methodology consists of two main phases: a deep test of compliance with the OpenFlow specification, and a verification of the availability and correct operation of several additional features that enable the deployment of SDN-based network architectures.

OpenFlow Compliance Test

OpenFlow is a continuously evolving specification: since its introduction in 2009 up to the time of this paper, at least 13 different revisions have been published by the Open Networking Foundation. OpenFlow 1.1 is a milestone in this landscape, considering that it improved handling of VLAN tags and it introduced support for multiple flow tables, MPLS labels, and bitmasks on MAC and IP addresses (used, e.g., to match IP subnets). Despite the importance of these features for most practical applications, many devices and controllers are declared to comply with the earliest OpenFlow 1.0 specification, while more recent versions are being incrementally adopted. Since revisions of the specification are released at a rather rapid pace, most implementations

Table 5.2: Features of OpenFlow compliance test suites.

Test suite	**OpenFlow version(s)**	**Test cases**
Ryu [117]	1.3, 1.4	991 (OF 1.3)
oftest [118]	1.2, 1.3, 1.4 in the works	~ 200
ONF OpenFlow conformance tests [119, 120]	1.0.1, 1.3.4	208 (OF 1.0.1) 334 (OF 1.3.4)
Open SDN network virtualization test [121]	N/A	<100
Ixia IxANVL OpenFlow test suite [122]	1.0.1, 1.3.2	>194 (OF 1.0) 528 (OF 1.3)
OFLOPS [123]	1.0	<50
Veryx ATTEST OpenFlow conformance test [124]	1.0.0, 1.3.1, 1.3.2, 1.3.3	400
Spirent TestCenter OpenFlow compliance test [125]	1.0.1, 1.3, 1.4	>950 (OF 1.3) >100 (OF 1.4)

skipped from OpenFlow 1.0 directly to OpenFlow 1.3, whereas the most recent OpenFlow 1.5 [8] is supported in a negligible number of cases. Therefore, we refer our compliance test to OpenFlow 1.3 [116].

Testing the level of compliance with OpenFlow is a rather cumbersome task, given the extension of the specification. Fortunately, there exist software tools that automate this job. Table 5.2 describes the main OpenFlow compliance test suites in terms of the OpenFlow version they can test and of the count of applied test cases. For our tests we chose the OpenFlow switch test tool included in the Ryu controller framework [117] (we used version 3.18), which is publicly available, offers a very rich set of test cases, is maintained by an active community of researchers and developers, and is also used by some vendors for compliance tests (as confirmed during internal communications). Although this test suite is readily usable, we argue that drawing conclusions from its reports is not immediate, and we give a contribution also in this direction.

Running the Ryu test suite required 2 additional devices besides the target datapath, as specified in Figure 5.1:

A tester datapath, used to submit different kinds of packets to the target

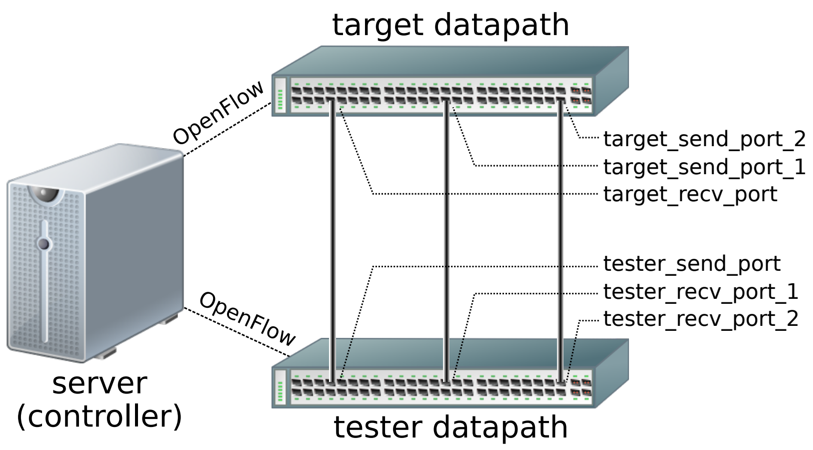


Figure 5.1: Network topology required in order to run the Ryu test suite.

and to collect response packets from the latter, and a server, acting as a controller for both the target (where different flow entries are installed for each test case) and the tester (which is instructed to emit packets and send the received responses back to the controller). Different test cases may require different sets of ports, therefore the target datapath and the tester datapath are connected as in the figure. Since different datapaths assign OpenFlow port numbers in different ways, before running the Ryu test suite we had to slightly adapt its test cases to consider parametric port numbers.

The Ryu test suite considers four classes of test cases:

Action Verify packet forwarding and manipulation functions (e.g., TTL alteration, push/pop of VLAN/MPLS headers).

Group Verify support for *group actions*, namely actions that are performed on multiple copies of the same packet (e.g., forwarding on multiple ports).

Match Verify an extensive assortment of match conditions, considering various packet header fields, applying bitmasks on those fields, exploiting multiple flow tables, and using reserved port numbers (e.g., CONTROLLER to send a packet to the controller, ALL to send it out of all the ports).

Meter Verify support for the *meter table*, which is used for rate-based traffic classification. For these tests, the tester datapath is instructed to emit packets at configured rates.

In order to check the ability of a datapath to hit or miss certain match conditions, every test case is repeated multiple times (typically between 3 and 4) with test packets that have a different set of headers (for example, MPLS or VLAN) or transport different protocols (for example, ARP, IPv4, or IPv6).

Custom Tests

We also defined a set of additional custom tests, which we used to cover aspects not considered by the Ryu test suite and to investigate deeper in the cause of failed Ryu test cases. Some of these tests are listed in Table 5.3.

For example, we exploited test `ct_normal` to determine the ability of a datapath to communicate in-band with the controller (we recall that support for the NORMAL port is declared as optional in the OpenFlow specification). Test `ct_multi_ctrl` is meant to assess robustness and manageability: it verifies whether a datapath sends a copy of the received packets to all the configured controllers (implying consistence problems in the controller design) or just to a single one, considering the others as backups. Finally, test `ct_hidden` verifies whether a datapath maintains default flow entries that are not normally visible in any flow tables and yet influence its behavior.

We also defined targeted versions of selected Ryu test cases, which we used to investigate the cause of reported failures. These tests consist in repeating the applicable test case in a controlled environment, namely using a simplified flow entry on the target datapath that is enough to trigger the problem and using a network sniffer for monitoring the test packets entering and exiting that datapath. In this way we discovered that certain Ryu test cases failed because the corresponding flow entries mix unsupported features with features that would be supported if used alone, or because the value of a header field in a manipulated packet was off by one unit, which we do not consider to be detrimental for deploying an SDN-based architecture.

Finally, we carried out performance tests, to assess the presence of bottlenecks in OpenFlow implementations. For the `ct_perf_switch` test we connected 10GbE ports on the target datapath to create two loop topologies, as shown in Figure 5.2. We also installed simple flow entries matching on the

[‡]Metadata are user-defined registers, mainly used to pass information between flow tables.

Table 5.3: A sample of the custom tests that we defined, with the functionalities they verify.

Functional tests	
`ct_normal`	Support for the NORMAL reserved port
`ct_multi_ctrl`	Behavior in the presence of multiple configured controllers
`ct_hidden`	Existence of hidden flow tables with default entries

Targeted versions of selected Ryu test cases	
`ct_group`	Operation of group actions
`ct_mask`	Operation of bitmasks applied on matched header fields
`ct_vlan`	Support for pushing/popping single or multiple VLAN tags
`ct_mpls`	Support for pushing/popping single or multiple MPLS labels; assessment of label stack size limits
`ct_metadata`	Support for metadata‡in match conditions and actions

Performance tests	
`ct_perf_switch`	Switching performance as a function of flow table size
`ct_cpu`	CPU usage for flow entry matching and packet switching
`ct_flow_insert`	Time required to install entries in the flow table

input port and forwarding packets along the loops. Then, we first saturated the bandwidth of the involved ports by injecting enough test packets in every loop (we sent PacketOut messages to the datapath for this purpose). In order to increase the frequency of lookups in the flow table, we used small IPv4 packets with a payload of 46 bytes as test packets. Once 100% port usage was steadily reached, we started adding a progressively increasing number of en-

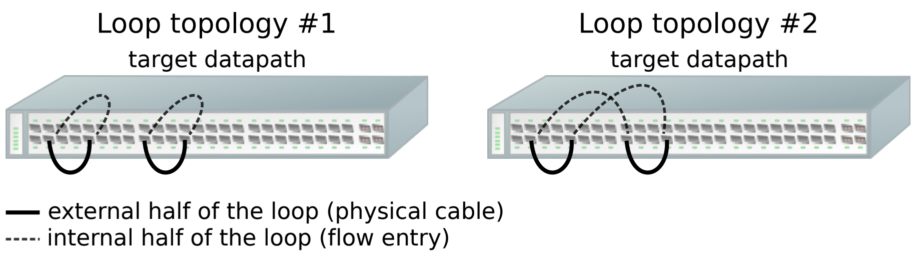

Loop topology #1
target datapath
Loop topology #2
target datapath
external half of the loop (physical cable)
internal half of the loop (flow entry)

Figure 5.2: Network topologies used during the performance tests.

tries to the flow table: each entry simply matched a different non-existent IP address, and its action was to forward packets out of a random port. Finally, as entries were installed, we checked whether switching performance was affected, which we consider an evidence of the overhead of matching flow table entries. We are aware that one or two loops are extremely far from saturating the capacity of a switch, but even with this simple experiment we could experience performance drops beyond a certain flow table size.

Once the datapath reached 100% port usage we also monitored the amount of consumed CPU, to determine whether packet forwarding was hardware accelerated (test `ct_cpu`). Moreover, for the `ct_flow_insert` test we measured the time required to install an increasingly high number of flow entries starting from an empty flow table. Each flow entry matched IP packets with randomly chosen IP addresses (duplicates were avoided) and had a single action, CONTROLLER. After installing each flow entry using a FlowMod message, we sent to the datapath an `OFPT_BARRIER_REQUEST` message, asking it to acknowledge the installation with an `OFPT_BARRIER_REPLY` message. We considered the time elapsed between adding the first flow entry and receiving the last `OFPT_BARRIER_REPLY` as the overall insertion time. We recall that the controller can issue an `OFPT_BARRIER_REQUEST` message to wait for completion of certain operations (e.g., the installation of flow entries), which is notified by the datapath in the form of an `OFPT_BARRIER_REPLY` message.

5.4 Outcome of Device Tests

In this section we describe the results of extensive tests we performed using the methodology in Section 5.3 on a range of hardware datapaths, which largely confirm the limitations discussed in Section 5.2 and unveil additional

Table 5.4: Main features of the tested datapaths.

ID	10GbE Ports	Switching Fabric Capacity	CAM	OpenFlow version	OVS based
S1	8	about 2 Tbps	CAM	1.3	No
S2	128	about 2 Tbps	TCAM	1.3	No
S3	64	about 1 Tbps	n/a	1.3, 1.4	Yes
S4	4	about 500 Gbps	TCAM	1.3	No
S5	4	about 500 Gbps	TCAM	1.3	No
S6	72	about 1 Tbps	n/a	1.3	No
S7	40	about 500 Gbps	n/a	1.3	Yes
OVS	n/a	n/a (software switch)	No	1.x	Yes

ones.

Test Setup

We analyzed 7 hardware datapaths manufactured by 7 major vendors, considering it a valuable opportunity to have so many devices simultaneously available for examination. Due to NDA constraints, we are unable to declare the model of the tested switches and the name of the involved vendors, therefore in the following we address them as S1, S2, ..., S7. All the datapaths were equipped with a forwarding ASIC. Some of them ran a customized version of Open vSwitch (OVS) [58] under the hood, associated with drivers for hardware-accelerated forwarding, whereas others had a proprietary OpenFlow implementation. The main features of the datapaths we considered are in Table 5.4. As a term of comparison, we also executed our tests (except performance tests) on OVS, which is known to have a very good level of compliance with the OpenFlow specification (see also [91]).

We used OVS version 2.3.90 as tester datapath, running on top of a server equipped with an Intel i7 3.50GHz CPU, 32GB of RAM, and 4 10GbE SFP+ network interfaces (we only used 3 of them). Since OVS is one of the most stable, feature-rich, and standards-compliant software datapath implementations, this choice ensured that our tests were not biased by implementation bugs in the tester datapath. We ran the Ryu test on a virtual machine with

2 Intel Xeon Sandy Bridge processors allocated from the hosting server and 2GB of RAM. Both servers executed Ubuntu Server Linux 14.04.2 LTS as operating system. The control channel between Ryu, the tester datapath, and the target datapath was realized by a dedicated 100Mbps management network. We performed the OpenFlow compliance tests as described in Section 5.3. As the sole exception, we could not connect the `target_send_port_2` of S2 (see Figure 5.1), which prevented us from running the *group* and *meter* test cases on this datapath. Moreover, we could not execute any performance tests on it either.

Before executing the Ryu test suite on each datapath, we had to reconstruct the OpenFlow port numbers associated with its physical interfaces, an important task considering that OpenFlow port numbers are assigned based on arbitrary conventions (see, e.g., [112]). We then passed these port numbers to Ryu to assign them the roles in Figure 5.1. We executed every run of the Ryu test multiple times, in order to make sure that the obtained results were reproducible (due to a firmware bug, we experienced non-deterministic outcomes on at least one of the datapaths). For those cases in which a suspiciously low count of passed tests was reported, we performed the following actions: we launched a reduced set of test cases, to verify the operation of at least basic functionalities, we installed simple flow entries in the target datapath to support a simple ping test, and we executed the custom tests in Section 5.3 to delve further into the problem. In this way we could at least rule out fundamental flaws in the various OpenFlow implementations.

Test Results

In Figure 5.3 we show the count of test cases that Ryu reported as passed for each considered datapath, distinguishing between tests that verify mandatory features in the OpenFlow specification (e.g., support for matched packet header fields, actions, etc.) and those that verify optional features. The dashed horizontal line is a threshold that corresponds to the total count of test cases (276) for mandatory features comprised in the Ryu test suite. The plot evidently shows that, besides OVS, only datapaths S3 and S7 passed more than 300 tests. Interestingly, these two datapaths are OVS-based. Other proprietary OpenFlow implementations typically barely reached 100 passed tests. A bit surprisingly, the count of passed tests for mandatory features never approached the threshold, not even for OVS, which however passed the highest number of tests. The gap for S6 in this plot as well as the following ones is due to a subtle bug in the OpenFlow implementation: in fact, this datapath

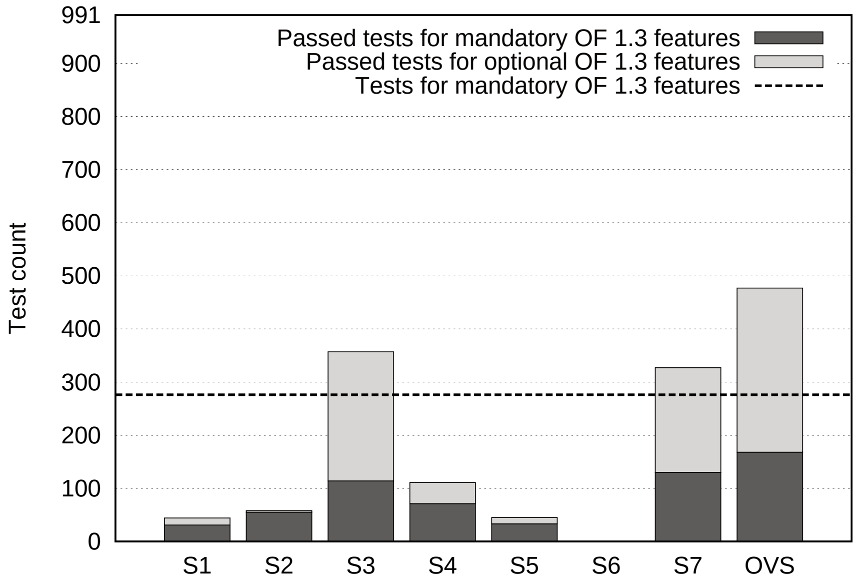


Figure 5.3: Count of passed Ryu tests.

assigned flow table IDs starting from 1 (instead of 0 as per the specification), causing mismatches in the outcome of all the applied test cases.

In order to have a more clear view of the functionalities supported by each datapath, we broke down the total number of passed tests according to the test classes defined in Section 5.3. Figure 5.4 shows, for each class, the percentage of tests passed by each datapath with respect to the total number of tests in that class. In accordance with the format of Ryu test reports, we moved to a separate class *set-field* those test cases that apply modifications to existing packet headers (e.g., rewrite L2/L3 addresses). It can be immediately noticed that test cases on the meter table were not passed by any datapaths: in all cases, the request to add meters was simply rejected by the datapath. Test cases concerning the group table and group actions were only passed by 3 datapaths (excluding OVS), despite the fact that this is a mandatory feature in the specification. Moreover, restricting to the *action* and *group* classes, datapath S3 performed better than any other datapaths, including OVS, even though it ranked second in terms of the total count of passed test cases (see

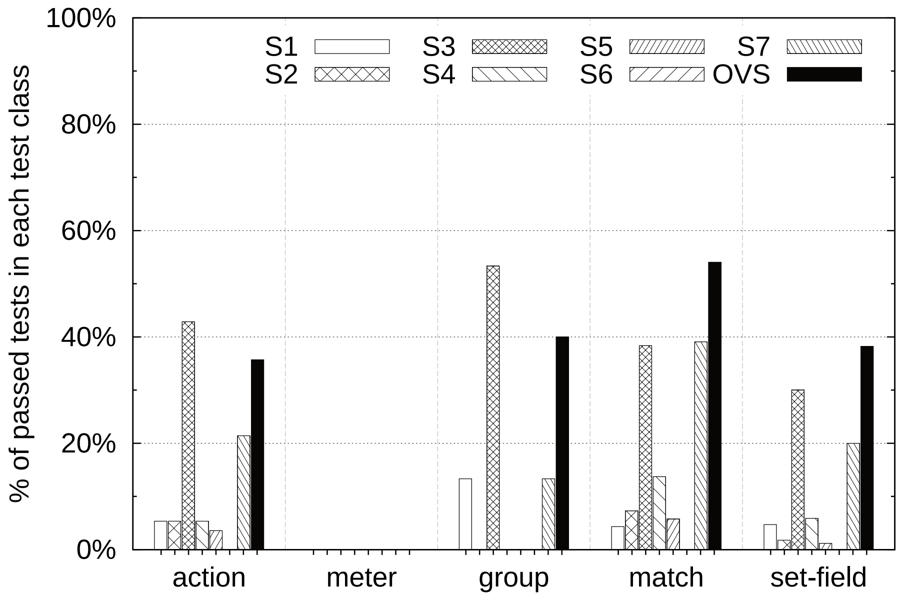


Figure 5.4: Percentage of passed OF 1.3 Ryu tests per test class.

Figure 5.3). Overall, no datapaths reached 60% of passed test cases per class.

To gain an even better understanding of the supported functionalities, we also classified the test cases according to the protocol headers used in the test packets sent to the target datapath, generating the plot in Figure 5.5. In particular, Ryu repeats each test case multiple times: tests focusing on functionalities that deal with layer 2, meters, and action groups are repeated with test packets carrying an IPv4, IPv6, or ARP header, whereas tests focusing on functionalities that concern layers 3 and 4 as well as ARP are repeated with test packets carrying a plain Ethernet (Eth), VLAN, MPLS, or PBB header[§]. For example, the MPLS class in Figure 5.5 represents test cases that verify the ability of the target datapath to process layer 3/4 headers in test packets that carry an MPLS header (this is different from saying that MPLS is the class of tests that verify support for MPLS-related functionalities: such tests are scat-

[§]PBB (Provider Backbone Bridge) is a standard acronym for indicating IEEE 802.1ah, a variation of QinQ.

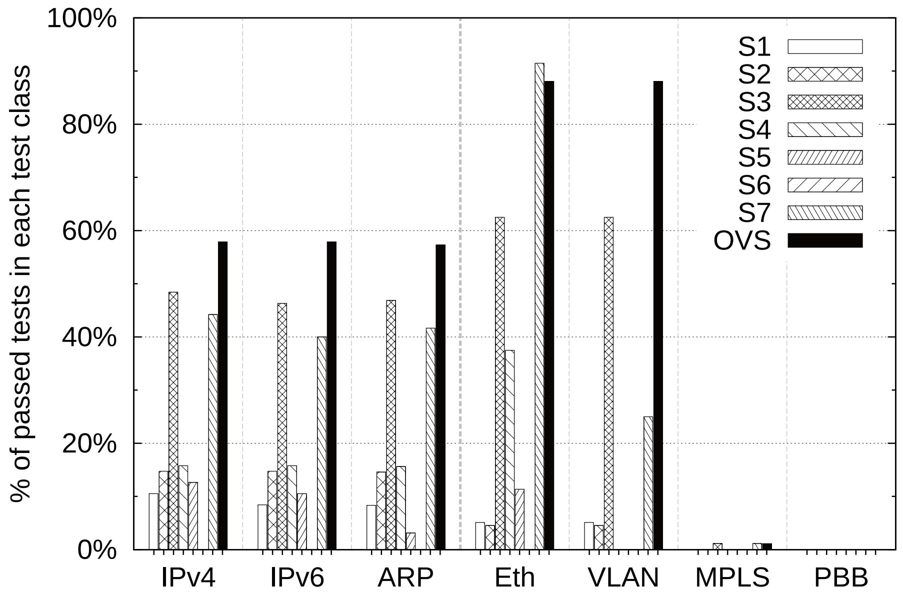


Figure 5.5: Percentage of passed OF 1.3 Ryu tests for test packets carrying a specific protocol.

tered among the IPv4, IPv6, and ARP classes). From the figure it is pretty evident that basically all datapaths were unable to accomplish any operations on packets carrying MPLS or PBB headers, thus impairing their usage in the related fields of application. As a confirmation of the results in Figure 5.4, datapath S3 always performed better than any others (excluding OVS), with the sole exception of the Eth class, where S7 remarkably exceeded 90% of passed tests.

Referring to the Figure 5.6, we now describe the results obtained during the performance tests, excluding S2 as already specified in Section 5.4. Datapaths S3 and S5 are those that exhibited the best performance during the `ct_perf_switch` test: even after saturating link capacities on the loops, the datapaths continued forwarding packets at a steady rate even with 200 additional entries installed in the flow table. For S1 and S4 the test could not be completed because installing additional flow entries unexpectedly caused the looping packets to be dropped (see Section 5.5 for the details). S6 was so flawed that we could not reliably exploit the installed flow entries to realize the loops in Figure 5.2. Finally, S7 exhibited a very low throughput (around 18Mbps) since the beginning of the test, due to the fact that firmware limitations forced us to disable hardware accelerated packet forwarding. For the `ct_cpu` test we used the CLI of each target datapath to collect statistics about CPU usage while the `ct_perf_switch` was running. For the aforementioned reasons we do not have CPU load results for S2 and S6, but for all the other datapaths the CPU was not involved in the packet switching (therefore, the flow table matching) process at all. Figure 5.6 shows the outcome of test `ct_flow_insert`, namely the time it took to install increasingly large sets of flow entries on the target datapath starting from an empty flow table. This time highly depends on the internal processing operations accomplished by the datapath to store flow entries into high-performance memory areas: to limit this bias, we used flow entries with a very simple regular structure (see Section 5.3). The plot, whose axes are in logarithmic scale, shows outstanding differences between the datapaths, especially beyond 500 installed flow entries. S1, S3, and S7 handled the installation process very efficiently (even though they may be subject to the `OFPT_BARRIER_REPLY` reliability problems discussed in [126]). S4 was slightly slower, but its insertion time was still proportional to the number of flow entries. On the other hand, S5 started to saturate beyond a certain flow table size, and did not even install all the requested flow entries due to hardware limitations. Even worse, S6 just dropped the connection with the controller beyond 50 installed flow entries.

As final remark, we want to point out that since the support for counters is declared as optional, as reported in Section 1.2, the outcome of the tests we executed on the datapaths never depended on their values. Instead, we often exploited the counters (if available) during troubleshooting sessions to inspect whether the expected flow entries were being matched.

5.5 A Catalog of Experienced Issues

While running the tests described in Section 5.4, we observed unexpectedly failed test cases as well as a number of anomalous behaviors. We debugged these issues with the help of the targeted tests described in Section 5.3, which we executed using a set of custom minimal controllers that performed basic operations (e.g., clearing the flow table, installing a single flow entry, sending an PacketOut message). In this section we describe the main anomalies we

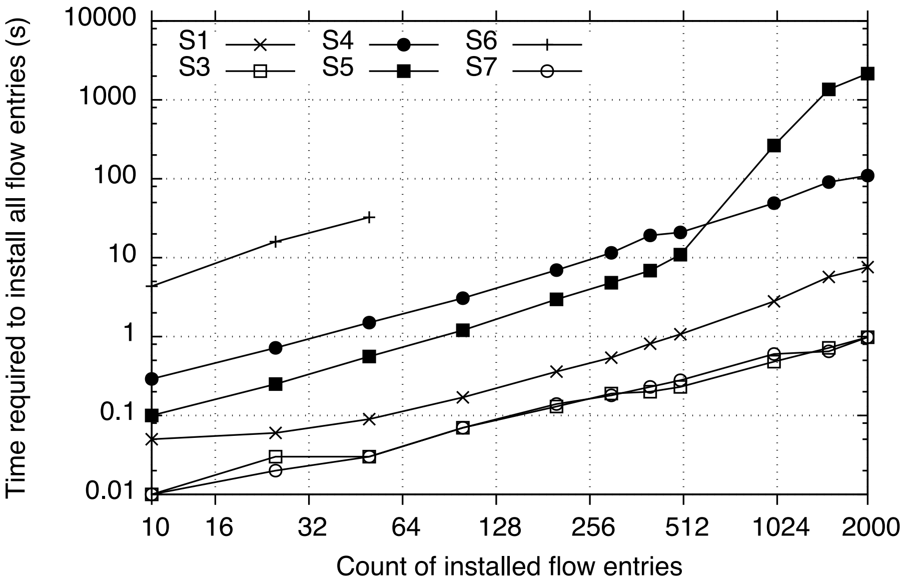


Figure 5.6: Flow entry insertion performance (axes are in logarithmic scale).

experienced, hoping that they can highlight aspects that network operators should care about when selecting devices for production use.

Functional Bugs

Some of the unexpected behaviors were evidently due to improperly implemented features. For example, on one datapath every flow entry was removed immediately after being installed, and on another datapath setting the table-miss flow entry to send packets to the controller had no effect (whereas an entry in a standard flow table did). Firmware upgrades (or even downgrades) sometimes solved such issues, but in other cases we were just stuck with unusable features.

By watching per-flow counters of matched packets, which most vendors implement, we observed several inconsistent conditions: for example, some installed entries did not match any packets unless their match conditions included an explicit Ethernet type; other entries matching on the VLAN ID only worked if their action was set to CONTROLLER; in one case, matched flow entries did not push a VLAN tag as requested; in another case, a flow entry whose match conditions contained additional fields besides the input port matched the expected packets but did not forward them unless the additional fields were removed from the match condition. While these behaviors might be due to improperly handled TCAM entries, we argue that such confusing conditions should be prevented by prohibiting flow entry installation altogether. In extreme cases, we saw the same set of flow entries nondeterministically matching or missing the same test packets (causing the Ryu tests to succeed only on a fraction of the runs), and we witnessed a packet matched by multiple flow entries (instead of a single one) pre-installed in a hidden flow table and missed by a custom flow entry that was supposed to match.

By monitoring the packets exchanged between datapaths and controller during our custom targeted tests, we could reveal a couple of very subtle bugs: on S1, PacketIn messages[¶] generated by a flow entry with an explicit action CONTROLLER always had the `reason` field incorrectly set to `OFPR_NO_MATCH`, meaning that all the entries in the flow table were missed. On S6, the `length` value of the `match` field carried by PacketIn messages (which contains the datapath port through which a data packet was received) was always incorrectly computed, causing PacketIn messages to be malformed. As a side note, on one datapath the counters of packets and bytes matched by each flow entry were available but only correctly updated for those rules that had at least one match condition (i.e., different from all-wildcard).

Violations of the Specification

Some Ryu test cases failed because of other misbehaviors caused by mistakes in the implementation of the OpenFlow specification. However, their severity was indeed much more moderate, because they did not impair the datapath operation.

For a couple of datapaths we observed that flow table IDs were not assigned starting from 0: for S2 this was due to the fact that lower IDs were reserved for flow entries with a specific structure, while S6 just adopted the convention of assigning IDs starting from 1. While affected datapaths can still

[¶]PacketIn is the OpenFlow message used by a datapath to forward a received data packet to the controller.

function correctly, this is likely to cause mismatches for example when watching counters associated with flow entries.

According to the OpenFlow specification, when pushing a new header (e.g., VLAN, MPLS) on a packet, the values of some fields in the pushed header should be inherited from existing headers, or initialized to 0 if no matching headers exist. For one datapath we observed that the value of the IP TTL was unexpectedly decremented by 1 unit when copying it to a freshly pushed MPLS header. Even more, an MPLS header pushed on top of an ARP packet (which misses the TTL) had its TTL set to 64. Interestingly, these issues did not occur when pushing an MPLS header on top of an existing MPLS header. Last, an MPLS header pushed on top of an IPv6 packet had its label erroneously initialized to 0x02, and popping the MPLS header off an IPv6 packet improperly changed the Ethernet type to 0x0800 (IPv4) instead of 0x86dd (IPv6). While the latter anomalies may lead to malformed packets, the others are less harmful and can be easily worked around.

In the presence of VLAN tags, the OpenFlow specification states that a match condition on the Ethernet type should consider the type of the first non-VLAN (i.e., innermost) header. All the datapaths we considered followed this rule except S4, which always picked the type carried in the Ethernet header.

Issues Revealed Under Stress

There is a variety of additional anomalies that we could discover only while carrying out performance tests. PacketOut messages were processed at a surprisingly low rate by S6 and, even though test packets were matched by the flow entries supporting the loop, no incoming packets were observed on any interfaces, invalidating the experiment. For other datapaths, after reaching 100% usage of network interfaces during `ct_perf_switch`, we saw that installing new flow entries that were not supposed to match any packets caused the test packets to be unexpectedly drained from the loop. On S4 this happened even without reaching 100% interface usage. On S1, we observed a consistently reproducible behavior: installing even a single additional flow entry while packets were looping caused any existing flow entries with higher priority to have their match counters reset and to temporarily stop matching any packets (the table-miss flow entry was applied instead). We believe such a condition is at least as dangerous as the inconsistent forwarding states observed in [126].

During test `ct_flow_insert`, not all the datapaths could keep the pace with FlowMod messages: one of them only installed a randomly selected subset of the requested flow entries, and from time to time it lost connection with the controller, causing new random subsets of flow entries to be cyclically installed every time a connection retry succeeded.

5.6 Takeaway

Combining publicly documented OpenFlow implementation limitations with the outcome of our tests results in a rather disappointing scenario, considering the maturity of the OpenFlow specification. In this section we summarize the implications of our study on the main research results on SDN.

General Considerations

First of all, some devices (e.g., S6) are still affected by hardware design issues (causing, e.g., flow entries to match packets non-deterministically) and software bugs (causing, e.g., malformed OpenFlow messages) that make them completely unusable for any practical applications of SDN. While software problems can be overcome with upgrades, hardware problems require a revision of the involved components and a replacement of any affected devices that are already deployed, making them harder to fix in the short term. In terms of functionalities and compliance with the OpenFlow specification, OVS-based datapaths (e.g., S3 and S7) proved to be the best choice and, since packet matching and forwarding tasks are offloaded to hardware, they also have very good performance. This makes them best suited for SDN application scenarios such as [100, 102, 103]. Open vSwitch can also be easily adapted to the underlying hardware: for example, it can explode flow entries with wildcards into exact match flow entries when there are no TCAMs available. However, due to the consequent memory load, this mode of operation is usually more suited for software-based switching, making some flow table compression strategies [95, 96, 97, 99] ineffective in such a scenario. Interestingly, our results on flow insertion times confirm those in [127] and show the absence of any correlation with switching fabric capacity.

Some functionalities are still widely unsupported. Only 3 out of the 7 datapaths we tested support match conditions and actions on IPv6 headers, impairing SDN deployment in modern networks. Rate-limiting traffic, a useful operation in scenarios such as [100, 101], can be very difficult with OpenFlow because the meter table is not supported by any devices. Load sharing may be impaired as well due to the limited support for group actions (see also [8]).

The missing support for processing PBB packets can make SDN-based interconnections between provider networks quite difficult but, admittedly, research in this direction is still preliminary (see, e.g., [128]). Moreover, the lack of functions for handling MPLS packets affects a class of approaches such as [101]. At least, all the tested datapaths are able to correctly process packets with standard Ethernet headers (see Figure 5.5), obviously except datapath S6 that fails due to hardware design issues, as mentioned at the beginning of this section.

Memory Constraints

Several vendors enforce partitions on the amount of available TCAM memory: this requires a careful planning of the estimated amount and type of installed flow entries. Choosing the optimal setup may be very difficult, considering that partitioning schemes are often constrained to predefined profiles, the count of TCAM entries consumed by a flow entry is rarely documented, TCAM memory blocks may be allocated to specific types of flow entries (e.g., matching on layer-2 or layer-3 fields), and some network modules may not support mixing these types. Moreover, the number of flow entries may be difficult to estimate in scenarios such as [102].

It is worth mentioning that the flow table compaction problem has also been tackled with general approaches that are independent from the structure of match conditions: in [113] this is accomplished by replacing match conditions with flow identifiers that are looked up in a separate table, while [129] proposes a method to optimize flow entry timeouts in order to minimize the number of stale entries. CacheFlow [130] introduces a clever caching mechanism to support the installation of more flow entries than allowed by flow table capacities. The applicability of all these approaches is unaffected by any of the discussed vendor limitations, especially considering that [129] and [130] do not manipulate existing flow entries in any ways.

As a side note, the presence of switch memory constraints makes finding a schedule for consistently updating flow entries on a set of datapaths an NP-complete problem [127]. It would be interesting, although out of the scope of this chapter, to analyze whether the computational complexity of other NP-hard theoretical problems analyzed in the literature about SDN is reduced by the introduction of vendor constraints.

Robustness Issues

Once a datapath with a suitable set of features has been singled out, several adjustments may need to be applied to ensure its robust operation. The default action undertaken by the table-miss flow entry (drop or send to controller) varies across vendors, therefore it is advised to explicitly set it in advance. Robustness of the datapath-controller communication channel is crucial, especially because losing connection with the controller may result in the flow table being cleared and no "emergency" flow entries being installed. Notably, only 2 or 3 controllers can often be configured for a single datapath, and a secure communication channel (e.g., SSL-based) may only be supported for a subset of them.

The behavior of a datapath should be carefully verified before putting it in production: as generic as this recommendation may seem, flow entries that do not produce the desired effect and interruptions of the matching process after entries with certain priorities have been installed (see Section 5.5) are very difficult issues to debug if not known in advance.

Policy Specification Languages

Several contributions rely on abstract policy specification languages such as [105, 106] to implement complex network architectures and services (e.g., [100, 131, 132]). While this constructive approach is very effective, high-level policies must ultimately be translated into flow entries: depending on the policy compiler logic, the resulting flow entries may violate vendor-imposed constraints, affecting the applicability of a whole class of scientific contributions that depend on the chosen policy language. Fortunately, some compilers [104] are designed considering that certain features (such as wildcards) should not be used on certain datapaths, and their behavior can be tuned accordingly (see the "reactive specialization" in [104]). Even P4 [133], a recently introduced language that supports a more flexible specification of the packet processing logic than in OpenFlow, risks to be affected by similar issues.

We are aware that some of the anomalies we experienced could be due to faulty devices. However, we consider this situation unlikely, because for some switches we had multiple instances exhibiting consistent behaviors, and because certain issues were common to different switches. All the discussed anomalies are exacerbated in a multi-vendor environment: besides the interoperability problems that may arise, an SDN controller may be restricted to consider only the intersection of the functionalities available on the controlled devices.

5.7 Open Problems

As an obvious continuation of this work, the testing methodology can be applied to additional devices. However, we also plan to extend our set of custom tests to comprise additional functionalities. Also, considering that the test specification format we used internally is very similar to the one adopted in [119], we would like to publicly release this specification. On the methodological side, we will be considering how existing SDN-based architectures such as [101] can be fit to comply with vendor-imposed restrictions.

Concluion and Future Works

The coexistence of many protocols in order to provide services leads some problems in the networks, in terms of flexibility, provisioning and manageability. Interacting with configuration files does not give to network administrators the same flexibility levels of other alternatives, in particular those relying on SDN. In this thesis, we study the problem of improving flexibility, provisioning, and manageability, showing that is possible and it can be done exploiting both distributed and centralized approaches. Of course, centralized approach is very promising in terms of simplifying many aspects in the networks, as shown in Chapter 3. We also provide an efficient mechanism for handling interoperability when SDN devices have to cooperate with legacy ones.

In Chapther 2 we introduce a control plane for internal routing inside an ISP's network with several desirable properties, including fine-grained control of routing paths, scalability, robustness, and QoS support, thus improving flexibility with an effort in simplifying the configuration effort and the number of protocols involved, since our control plane also provides QoS mechanisms. We provide a formal description about messages and algorithms involved in our control plane. We validate our approach through extensive experimentation, revealing promising scalability and competitive convergence times compared with OSPF.

In Chapter 3, we demonstrate how a complex and widely used service can be simplified, from the configuration point of view and the number of involved protocols in providing that service. Taking advantage of the programmability of SDN, we propose a novel approach to realize MPLS VPNs, which supports easy provisioning and setup based on a simple and flexible configuration language, facilitates management and troubleshooting by dropping unneeded technologies, and improves controllability and predictability by enforcing a centralized coordination of the behaviors of network devices, also enhancing the manageability of the network itself.

In Chapter 4, addressing interoperability issues, we describe the design of an SDN controller that handles exchange of ARP packets between SDN switches and legacy IP devices, also in the presence of multiple IP subnets, with low forwarding overhead for the SDN and low communication overhead for the controller. Functional tests on a range of SDN switches from different vendors confirm the viability of our approach, that is very important with respect to what described in Chapter 5.

Finally, in Chapter 5 we point out how research in the field of SDN is largely unaware of several restrictions encountered when it comes to deploying a proposed architecture on real devices. In this chapter we contrast a selection of recent research contributions with publicly documented limitations of OpenFlow implementations, and we exploit a custom testing methodology to carry out extensive tests on a range of commercial switches, unveiling several additional limitations and anomalies. We consider this work as a milestone assessing the gap between research results and their practical applicability on currently available devices, which turned out to be more appreciable than expected.

Unfortunately, Chapter 5 also reveals as the adoption of SDN, using OpenFlow, is still braked by some vendor implementations. This is not the only consideration that can be drawn. Indeed, another relevant consideration is referred to the applicability of some proposals at the state of the art. In particular, many papers addressing the problem of the flow tables' compression cannot be applied. For example in [93] authors do not take care of constraints that many vendor impose in the format of the flow entries. Other papers, like [95, 96], propose a massively use of wildcard: unfortunately this choice is not profitable, because wildcards drastically reduce (up to two orders of magnitude) the size of the flow tables, as reported in [109]. Another proposal affected by results observed in Chapter 5 is [100]. In fact, as also discussed in Section 5.2, using rules that match on layer-4 packet header fields reduce the number of installable flow entries. This chapter could also undermine the approach about VPN services shown in Chapter 3, becuase we chose to use MPLS labels for traffic forwarding and Figure 5.5 shows how current OpenFlow implementations do not support MPLS protocol at all. Actually, that lack does not affect our proposal, because we use MPLS labels just as a way to forward traffic: in absence of MPLS support, another way to forward traffic based on label is surely valid: an alternative is represented by VLAN header, that are extensively supported in current OpenFlow implementations by vendors.

With respect to the future works, an interesting direction is to define some objective criteria for establishing if SDN is better than traditional approaches. Considering several metrics (e.g., number of exchanged control plane messages), different implementations of a given service can be compared, in order to try to establish if an implementation performs better to another one, or if there is some relationship with external factors, like topology. This is useful both to help operators to understand whether, given a service, an implementation works better of another one, and to compare novel research proposals with existing ones.

Appendices

Appendix A: Pathlet Intra-Domain Routing Algorithms

In this chapter we extensively discuss algorithms used in our control plane. In Appendix A.1 we report algorithms used to handle network dynamics, such as a topological change whereas in Appendix A.2 we report algorithms describing the actions undertaken by each vertex in conseguence of the reception of a message. Finally, in Appendix A.3 we report the algorithm which our topology generator, discussed in Section 2.7, relies on.

A.1 Algorithms to Handle Network Dynamics

We now describe the operations undertaken by a vertex u when, in consequence of a topological change or an administrative reconfiguration, its set Π_u of known pathlets is changed to a new set Π_{new}. These operations are formalized as procedure UPDATEKNOWNPATHLETS(u, Π_{new}) in Algorithm 8. First of all, u disseminates pathlets that are newly appeared in Π_{new} (with respect to Π_u) and withdraws those that are no longer in this set. Note that, as anticipated in Section 2.5, at line 22 the forwarding state for removed pathlets is only cleared after a timeout T_u^f. Vertex u also disseminates pathlets that had their scope stack or set of destinations updated, and withdraws them from those neighbors that (according to propagation conditions and routing policies) cannot receive the updated instance. Then, u updates its set Π_u and removes pathlets that can no longer be used in any composition because their start vertex is not reachable. Last, u updates its crossing and final pathlets.

Algorithm 8 Algorithm to update the set Π_u of known pathlets at a vertex u. Procedure UPDATEKNOWNPATHLETS only realizes the update of Π_u, while UPDATEKNOWNPATHLETSANDSTACK considers the case when the label stack of u is contextually changed from S_{old} to S_{new}.

```
1:  procedure UPDATEKNOWNPATHLETS($u$, $\Pi_{new}$)
2:    UPDATEKNOWNPATHLETSANDSTACK($u$, $S(u)$, $S(u)$, $\Pi_{new}$)
3:  end procedure

4:  procedure UPDATEKNOWNPATHLETSANDSTACK($u$, $S_{old}$, $S_{new}$, $\Pi_{new}$)
5:    for each $\pi = \langle FID, v, w, \sigma, \delta \rangle \in \Pi_{new} \setminus \Pi_u$ do
6:      $\pi$ is a new pathlet or an updated instance of a pathlet that is both in $\Pi_u$ and in $\Pi_{new}$
7:      if $u = v$ then
8:        Update $u$'s forwarding state according to the composition of $\pi$
9:        $M \leftarrow$ new Pathlet message; $M.\mathtt{p} \leftarrow \pi$
10:       Send $M$ to each neighbor $x \in N(u, S_{new}, \pi)$
11:     end if
12:   end for
13:   for each $\pi_{old} = \langle FID, v, w, \sigma_{old}, \delta_{old} \rangle \in \Pi_u \setminus \Pi_{new}$ do
14:     if $u = v$ then
15:       $M \leftarrow$ new Withdrawlet message; $M.\mathtt{f} \leftarrow FID$; $M.\mathtt{s} \leftarrow \sigma_{old}$
16:       if $\exists \pi_{new} = \langle FID, v, w, \sigma_{new}, \delta_{new} \rangle \in \Pi_{new}$ then
17:         $\pi_{old}$ is in $\Pi_u$ and has an updated instance $\pi_{new}$ in $\Pi_{new}$
18:         Send $M$ to each neighbor $x \in N(u, S_{old}, \pi_{old}) \setminus N(u, S_{new}, \pi_{new})$
19:       else
20:         $\pi_{old}$ is in $\Pi_u$ but has been removed in $\Pi_{new}$
21:         Send $M$ to each neighbor $x \in N(u, S_{old}, \pi_{old})$
22:         Clear $fids_u(FID)$ and $nh_u(FID)$ after a timeout $T_u^f$
23:       end if
24:     end if
25:   end for
26:   $\Pi_u \leftarrow \Pi_{new}$
27:   for each $\pi = \langle FID, v, w, \sigma, \delta \rangle \in \Pi_u$ do
28:     if $chains(\Pi_u, u, v, S_{new}) = \emptyset$ or any concatenation of a pathlet in
        $chains(\Pi_u, u, v, S_{new})$ with pathlet $\pi$ has a cycle then
29:       Initialize $T_u^p(\pi)$ to a configured pathlet timeout
30:     else
31:       $T_u^p(\pi) \leftarrow \oslash$
32:     end if
33:   end for
34:   UPDATECOMPOSEDPATHLETSANDSTACK($u$, $S_{old}$, $S_{new}$, $\Pi_u$)
35: end procedure
```

We now illustrate the operations undertaken by u to update its sets of crossing and final pathlets C_u and F_u, for each area for which u is a border vertex. These operations are formalized as procedure UPDATECOMPOSEDPATHLETS(u, Π_{new}) in Algorithm 10. First of all, u composes any newly available crossing pathlets and removes those that can no longer be composed. The latter operation uses function ISPATHLETCOMPOSABLE(u, π, Π, V_{end}) in Algorithm 9, which determines whether π can (still) be composed by u based on known pathlets in Π and admissible end vertices in V_{end}. Then, u updates its forwarding state for newly composed pathlets and disseminates them. Similarly to [19], before removing a pathlet, u checks whether it can be transparently replaced with an alternative one and, if so, u just updates its forwarding state without sending any messages. Otherwise, u sends withdrawal messages to its neighbors, and clears the forwarding state after a timeout T_u^f which is used to support correct forwarding of network traffic that still uses the old pathlets.

Algorithm 9 Algorithm to check whether a pathlet π can (still) be composed by a vertex u given a set Π of known pathlets and a set V_{end} of admissible end vertices for π.

```
1: function IsPathletComposable($u$, $\pi$, $\Pi$, $V_{\text{end}}$)
2:     Let $\pi = \langle FID, u, v, \sigma, \delta \rangle$
3:     if $v \in V_{\text{end}}$ and $\exists(\pi_1\ \pi_2\ \dots\ \pi_n) \in chains(\Pi, u, v, \sigma)$ such that $\pi_i = \langle FID_i, u_i, v_i, \sigma_i, \delta_i \rangle$, $i = 1, \dots, n$ and $fids_u(FID) = (FID_2\ FID_3\ \dots\ FID_n)$ and $nh_u(FID) = u_2$ and the pathlet composition rules allow composition of $\pi$ then
4:         return True
5:     else
6:         return False
7:     end if
8: end function
```

We remark that in our model most topological variations and administrative reconfigurations can be represented as a change of label stacks: addition of a link (u, v) is modeled by setting $S_u(v) = S(v)$ and $S_v(u) = S(u)$; removal of a link (u, v) is modeled by setting $S_u(v) = ()$ and $S_v(u) = ()$; addition and removal of a vertex are modeled as a simultaneous addition or removal of all its incident edges; an administrative change of the label stack $S(v)$ of a vertex v is modeled as an update of stacks $S_w(v)$ of the neighbors w of v.

As a consequence, we can assume that all relevant network dynamics involve a change of the stack of some vertex. When a vertex u has its stack

Algorithm 10 Algorithm to update the sets of crossing and final pathlets composed by u. Procedure UPDATECOMPOSEDPATHLETS implements the refresh of crossing and final pathlets, while procedure UPDATECOMPOSEDPATHLETSANDSTACK also considers the case when the label stack of u is contextually changed from S_{old} to S_{new}.

```
1: procedure UPDATECOMPOSEDPATHLETS($u$, $\Pi_{new}$)
2:     UPDATECOMPOSEDPATHLETSANDSTACK($u$, $S(u)$, $S(u)$, $\Pi_{new}$)
3: end procedure

4: procedure UPDATECOMPOSEDPATHLETSANDSTACK($u$, $S_{old}$, $S_{new}$, $\Pi_{new}$)
5:     for each area $A_\sigma$ do
6:         $C_{new} \leftarrow \emptyset$; $C_{old} \leftarrow C_u(\sigma)$
7:         if $u$ is a border vertex for $A_\sigma$ then
8:             $B_u(\sigma) \leftarrow$ DISCOVERBORDERVERTICES($u$, $\sigma$, $\Pi_{new}$)
9:             if $C_u(\sigma) = \emptyset$ then        $u$ is border for $A_\sigma$ but has not composed any pathlets
10:                $C_{new} \leftarrow crossing_u(\Pi_{new}, \sigma)$ (end vertices are in $B_u(\sigma)$)
11:            else        $u$ has to refresh crossing pathlets
12:                $C_{new} \leftarrow$ new crossing pathlets for $A_\sigma$ not in $C_u(\sigma)$, that $u$ can compose towards vertices in $B_u(\sigma)$ using pathlets in $\Pi_{new}$ and according to the pathlet composition rules
13:                $C_{old} \leftarrow \{\pi | \pi \in C_u(\sigma)$ and not ISPATHLETCOMPOSABLE$(u, \pi, \Pi_{new}, B_u(\sigma))\}$
14:            end if
15:        end if
16:        Update $u$'s forwarding state for any pathlet in $C_{new}$
17:        for each $\pi_{old} = \langle FID_{old}, v, w, \sigma, \delta\rangle \in C_{old}$ do
18:            if $\exists \pi_{new} = \langle FID_{new}, v, w, \sigma, \delta\rangle \in C_{new} \setminus C_{old}$ then  Transparently replace $\pi_{old}$ by $\pi_{new}$
19:                $fids_u(FID_{old}) \leftarrow fids_u(FID_{new})$; $nh_u(FID_{old}) \leftarrow nh_u(FID_{new})$
20:                $C_{new} \leftarrow (C_{new} \setminus \{\pi_{new}\}) \cup \{\pi_{old}\}$
21:            else        Withdraw a no longer composed pathlet
22:                $M \leftarrow$ a new Withdrawlet message; $M.\mathtt{f} \leftarrow FID_{old}$; $M.\mathtt{s} \leftarrow \sigma$
23:                Send $M$ to each neighbor $x \in N(u, S_{old}, \pi_{old})$
24:                Clear $fids_u(FID_{old})$and $nh_u(FID_{old})$ after a timeout $T_u^f$
25:            end if
26:        end for
27:        for each $\pi_{new} = \langle FID_{new}, v, w, \sigma, \delta\rangle \in C_{new} \setminus C_{old}$ do
28:            Announce newly composed pathlets
29:            $M \leftarrow$ a new Pathlet message; $M.\mathtt{p} \leftarrow \pi_{new}$
30:            Send $M$ to eachneighbor $x \in N(u, S_{new}, \pi_{new})$
31:        end for
32:        $C_u(\sigma) \leftarrow (C_u(\sigma) \setminus C_{old}) \cup C_{new}$
33:    end for
34:    Repeat the same steps replacing set $C_u(\sigma)$ with $F_u(\sigma)$, set $B_u(\sigma)$ with $A_\sigma \cap D$, set $crossing(\Pi_{new}, \sigma)$ with $final(\Pi_{new}, \sigma)$, and "crossing pathlets" with "final pathlets"
35: end procedure
```

changed from S_{old} to S_{new}, it accomplishes several operations. First of all, u sends to its neighbors a **Hello** message M with $M.\mathtt{s} = S_{new}$, $M.\mathtt{d}$ set according to the network destinations available at u, and $M.\mathtt{a} = \mathtt{False}$. Then, u refreshes its knowledge of all the atomic pathlets towards its neighbors, by re-constructing them as explained in Section 2.4, and prepares a refreshed set Π_{new} accordingly. At this point, u triggers a refresh of its set Π_u containing all known pathlets by executing procedure UPDATEKNOWNPATHLETSANDSTACK(u, S_{old}, S_{new}, Π_{new}) in Algorithm 8, which realizes the same steps as UPDATEKNOWNPATHLETS while restricting the propagation of any messages in accordance with the stack change. Last, u updates its sets of crossing and final pathlets by executing procedure UPDATECOMPOSEDPATHLETSANDSTACK(u, S_{old}, S_{new}, Π_{new}) in Algorithm 10, which only adds to UPDATECOMPOSEDPATHLETS the ability to propagate messages according to the stack change.

To complete the picture of possible network dynamics, we assume that a change in the routing policies at a vertex u causes a reboot of u. Additionally, if u has its set δ of available network destinations updated, it just needs to send an updated **Hello** message to its neighbors. This message will, in turn, trigger a refresh of the pathlets towards u stored by other vertices in the network.

A.2 Message Handling

We now describe the actions undertaken by a vertex u upon reception of a message M.

Receipt of a Hello Message

When a vertex u receives a **Hello** message M from a neighbor $M.\mathtt{o}$, it performs several actions. First of all, u updates its knowledge about vertex $M.\mathtt{o}$ by setting $S_u(M.\mathtt{o}) = M.\mathtt{s}$ and $D_u(M.\mathtt{o}) = M.\mathtt{d}$. Moreover, u updates its own status of border vertex depending on whether $S(u) \not\sqsubseteq M.\mathtt{s}$: if this is the case, then u is a border vertex for any areas A_σ such that $(S(u) \rightarrowtail M.\mathtt{s}) \sqsubseteq \sigma \sqsubseteq S(u)$. Then, if $M.\mathtt{a} = \mathtt{True}$, which means that this is the first **Hello** message sent by $M.\mathtt{o}$ since its activation, u synchronizes $M.\mathtt{o}$ with the current network status by sending to $M.\mathtt{o}$ **Pathlet** and **Withdrawlet** messages according to the content of the history H_u. In particular, for every pathlet $\pi = \langle FID, v, w, \sigma, \delta \rangle$ in any of the sets Π_u, C_u, and F_u kept by u such that $M.\mathtt{o} \in N(u, S(u), \pi)$, u sends to $M.\mathtt{o}$ a **Pathlet** message M_P with $M_P.\mathtt{p} = \pi$, $M_P.\mathtt{o} = v$, and $M_P.\mathtt{t} = t$, where t is taken from entry $\langle FID, v, \sigma, t, + \rangle$ in history H_u (note that such an entry must exist for every pathlet learned or created by u). Moreover, for every entry $\langle FID, v, \sigma, t, -\rangle$ in history H_u such that any pathlet with start vertex v and scope stack σ could have been sent by u to $M.\texttt{o}$, u sends to $M.\texttt{o}$ a **Withdrawlet** message M_W with $M_W.\texttt{f} = FID$, $M_W.\texttt{s} = \sigma$, $M_W.\texttt{o} = v$, and $M_W.\texttt{t} = t$. Observe that, in sending these messages, u preserves the origin vertex and timestamp of the originally learned information, as specified in the history.

Next, u constructs or updates an atomic pathlet towards $M.\texttt{o}$, as explained in Section 2.4, and updates its set Π_u of known pathlets consequently, by executing procedure UPDATEKNOWNPATHLETS in Algorithm 8. Based on its status of border vertex, u also updates its crossing and final pathlets by executing procedure UPDATECOMPOSEDPATHLETS in Algorithm 10.

Note that, even if vertex $M.\texttt{o}$ has sent an updated label stack, for example due to a stack change, it may be the case that no pathlets are updated by u and no messages are sent by u. In fact, if $S(u) \bowtie M.\texttt{s}$ is unchanged, u will not apply any changes to atomic pathlets in consequence to the received **Hello** message, thus leaving its set Π_u unchanged: in these conditions none of the **for** cycles at lines 5 and 13 of Algorithm 8 executes any iterations, and the operations within the **for** cycle at line 27 have no effects. Likewise, sets C_{old} and C_{new} in Algorithm 10 have no elements when the execution reaches line 16, resulting in no actions being performed.

Finally, if the set of known destinations $D_u(M.\texttt{o})$ has been modified, u disseminates the new information by re-sending **Pathlet** messages to its neighbors with the updated destinations.

Receipt of a Pathlet Message

If M is a **Pathlet** message received from $M.\texttt{src}$, u first checks whether the carried information is fresh, by evaluating procedure ISPATHLETMESSAGEFRESHER(u, M) in Algorithm 11: if this is not the case, u sends back to $M.\texttt{src}$ an updated message, as shown in the algorithm, and undertakes no further actions. Otherwise, u refreshes the history H_u and executes Algorithms 8 and 10 to update sets Π_u C_u, and F_u according to the newly learned information and to disseminate any updated information to its neighbors.

Algorithm 11 Algorithm to determine whether a **Pathlet** message M carries updated information about a pathlet: the function returns `True` only in this case. It also handles message forwarding and history update.

```
 1: function IsPathletMessageFresher($u$, $M$)
 2:   $\pi_{msg} \leftarrow M.\texttt{p}$
 3:   Let $\pi_{msg} = \langle FID, v, w, \sigma_{msg}, \delta_{msg} \rangle$
 4:   if $u = v$ then
 5:     $u$ is the originator of pathlet $\pi_{msg}$ and therefore has information about this pathlet
 6:     if there is no pathlet identified by $FID$ and with start vertex $u$ in $\Pi_u$ or in any of the sets $C_u$ and $F_u$ then
 7:       $M_W \leftarrow$ new Withdrawlet message; $M_W.\texttt{f} \leftarrow FID$; $M_W.\texttt{s} \leftarrow \sigma_{msg}$
 8:       Send $M_W$ to neighbor $M.\texttt{src}$
 9:     else
10:       $\pi_{cur} \leftarrow$ a pathlet identified by $FID$ and with start vertex $u$ that is in $\Pi_u$ or in any of the sets $C_u$ and $F_u$
11:       if $\pi_{msg} \neq \pi_{cur}$ then
12:         Pathlet $\pi_{cur}$ is always fresher because it has been created by $u$
13:         $M_P \leftarrow$ new Pathlet message; $M_P.\texttt{p} \leftarrow \pi_{cur}$
14:         Send $M_P$ to neighbor $M.\texttt{src}$
15:       end if
16:     end if
17:     return False
18:   else
19:     if $\exists \langle FID, v, \sigma, t, type \rangle$ in $H_u$ then
20:       $\pi_{msg}$ was already known at $u$
21:       if $t < M.\texttt{t}$ then
22:         The received pathlet is fresher
23:         Replace $\langle FID, v, \sigma, t, type \rangle$ in $H_u$ with $\langle FID, v, \sigma_{msg}, M.\texttt{t}, + \rangle$
24:         Send $M$ to each neighbor $x \in N(u, S(u), \pi_{msg}) \backslash \{M.\texttt{src}\}$
25:         return True
26:       else
27:         $u$'s information about $\pi_{msg}$ is fresher than the information carried in $M$
28:         if $type = +$ then
29:           $\pi_{cur} \leftarrow$ pathlet in $\Pi_u$ identified by $FID$ and with start vertex $v$
30:           $M_P \leftarrow$ new Pathlet message; $M_P.\texttt{p} \leftarrow \pi_{cur}$; $M_P.\texttt{t} \leftarrow t$
31:           Send $M_P$ to neighbor $M.\texttt{src}$
32:         else
33:           $M_W \leftarrow$ new Withdrawlet message; $M_W.\texttt{f} \leftarrow FID$; $M_W.\texttt{s} \leftarrow \sigma$;
34:           $M_W.\texttt{t} \leftarrow t$
35:           Send $M_W$ to neighbor $M.\texttt{src}$
36:         end if
37:         return False
38:       end if
39:     else
40:       $\pi_{msg}$ is a newly learned pathlet
41:       Add $\langle FID, v, \sigma_{msg}, M.\texttt{t}, + \rangle$ to $H_u$
42:       Send $M$ to each neighbor $x \in N(u, S(u), \pi_{msg}) \backslash \{M.\texttt{src}\}$
43:       return True
44:     end if
45:   end if
46: end function
```

Receipt of a Withdrawlet Message

Handling of a **Withdrawlet** message M received by a vertex u is much similar to that of a **Pathlet** message. First of all, u checks the freshness of the information carried by M by evaluating function IsWithdrawletMessageFresher(u, M), which is defined in Algorithm 12: if this function returns `False`, then u sends back to $M.\mathtt{src}$ a **Pathlet** or a **Withdrawlet** message with the most up-to-date information for the corresponding pathlet, and processing of M is finished. Otherwise, u refreshes the history H_u, checks whether a pathlet with *FID* $M.\mathtt{f}$, start vertex $M.\mathtt{o}$, and scope stack $M.\mathtt{s}$ exists in Π_u and, if this is the case, removes it from Π_u and executes Algorithm 8.

Receipt of a Withdraw Message

Receiving a **Withdraw** message M has the same effect of receiving several **Withdrawlet** messages, all with the same timestamp $M.\mathtt{t}$, one for each *FID* of the pathlets in Π_u that have scope stack $M.\mathtt{s}$ and start vertex $M.\mathtt{o}$. In order to handle this type of message, function IsWithdrawletMessageFresher needs to be slightly modified as follows: if M carries fresher information for all the pathlets in Π_u with scope stack $M.\mathtt{s}$ and start vertex $M.\mathtt{o}$, then history H_u is appropriately updated for all these pathlets and only the single **Withdraw** message is further propagated by u; otherwise, if u has a more recent history entry in H_u for at least one of these pathlets, u treats the **Withdraw** message exactly as a sequence of **Withdrawlet** messages, sending back to $M.\mathtt{src}$ single **Pathlet** and **Withdrawlet** messages with updated information, and forwarding single **Withdrawlet** messages as appropriate. If M is determined to carry fresh information, pathlets are then updated by u as already explained for the **Withdrawlet** message.

Algorithm 12 Algorithm to determine whether a **Withdrawlet** message M carries updated information about a pathlet: the function returns `True` only in this case. It also handles message forwarding and history update.

```
function IsWithdrawletMessageFresher($u$, $M$)
    if $\exists \langle M.\mathtt{f}, M.\mathtt{o}, \sigma, t, \textit{type} \rangle$ in $H_u$ then
        if $t < M.\mathtt{t}$ then
            Replace $\langle M.\mathtt{f}, M.\mathtt{o}, \sigma, t, \textit{type} \rangle$ in $H_u$ with $\langle M.\mathtt{f}, M.\mathtt{o}, M.\mathtt{s}, M.\mathtt{t}, - \rangle$
            for each $n \in N(u, S(u), M.\mathtt{s}) \setminus \{M.\mathtt{src}\}$ do
                Send $M$ to neighbor $n$
            end for
            return True
        else
            if $\textit{type} = +$ then
                $\pi_{cur} \leftarrow$ pathlet in $\Pi_u$ identified by $M.\mathtt{f}$ and with start vertex $M.\mathtt{o}$
                $M_P \leftarrow$ new Pathlet message
                $M_P.\mathrm{p} \leftarrow \pi_{cur}$
                $M_P.\mathtt{t} \leftarrow t$
                Send $M_P$ to neighbor $M.\mathtt{src}$
            else
                $M_W \leftarrow$ new Withdrawlet message
                $M_W.\mathtt{f} \leftarrow M.\mathtt{f}$
                $M_W.\mathtt{s} \leftarrow M.\mathtt{s}$
                $M_W.\mathtt{t} \leftarrow t$
                Send $M_W$ to neighbor $M.\mathtt{src}$
            end if
            return False
        end if
    else
        There is no history entry for the pathlet withdrawn by $M$, therefore $u$ cannot know anything about that pathlet. However, the Withdrawlet must still be forwarded
        Add $\langle M.\mathtt{f}, M.\mathtt{o}, M.\mathtt{s}, M.\mathtt{t}, - \rangle$ to $H_u$
        for each $n \in N(u, S(u), M.\mathtt{s}) \setminus \{M.\mathtt{src}\}$ do
            Send $M$ to neighbor $n$
        end for
        return False
    end if
end function
```

A.3 Algorithm to Generate Topologies

Algorithm 13 Algorithm used in our topology generator.

```
function PopulateArea($A$, $level$, $R_{\min}$, $R_{\max}$, $N$, $A_{\min}$, $A_{\max}$, $P$, $B$)
  if $level = N$ then
    $r \leftarrow$ a random number in $[R_{\min}, R_{\max}]$
    Add $r$ vertices to $A$
    repeat
      for each pair $(u, v)$ of vertices in $A$ do
        Add an edge between $u$ and $v$ with probability $P$
      end for
    until $A$ is connected
    Randomly pick $r \times B$ routers in $A$ and mark them as border routers for $A$
  else
    $a \leftarrow$ a random number in $[A_{\min}, A_{\max}]$
    Create $a$ areas inside $A$; let $\mathcal{A}$ be the set of these areas
    $\bar{R} \leftarrow \emptyset$
    for each $\bar{A}$ in $\mathcal{A}$ do
      PopulateArea($\bar{A}$, $level + 1$, $R_{\min}$, $R_{\max}$, $N$, $A_{\min}$, $A_{\max}$, $P$, $B$)
      $\bar{R} \leftarrow \bar{R} \cup$ all border routers for $\bar{A}$
    end for
    repeat
      $E \leftarrow \emptyset$
      for each pair $(u, v)$ of routers in $\bar{R}$ do
        Flip a coin with probability $P$
        if heads then
          Add an edge between $u$ and $v$
          $E \leftarrow E \cup (u, v)$
        end if
      end for
    until the undirected graph formed by vertices in $\bar{R}$ and edges in $E$ is connected
    Randomly pick $|\bar{R}| \times B$ routers in $\bar{R}$ and mark them as border routers for $A$
  end if
  return $A$
end function
function TopologyGenerator($R_{\min}$, $R_{\max}$, $N$, $A_{\min}$, $A_{\max}$, $P$, $B$)
  Create an area $A$
  $level \leftarrow 1$
  return PopulateArea($A$, $level$, $R_{\min}$, $R_{\max}$, $N$, $A_{\min}$, $A_{\max}$, $P$, $B$)
end function
```

Appendix B: Publications

Journal Publications

1. M. Chiesa, G. Lospoto, M. Rimondini, G. Di Battista. Intra-Domain Routing with Pathlets. Computer Communications. 46:76-86. 2014

Conference Publications

1. R. di Lallo, G. Lospoto, M. Rimondini, G. Di Battista. How to Handle ARP in a Software-Defined Network. In *Conference on Network Softwarization (NetSoft 2016)*, IEEE, 2016.

2. R. di Lallo, G. Lospoto, M. Rimondini, G. Di Battista. Supporting End-to-End Connectivity in Federated Networks using SDN. In, *Melike Erol-Kantarci, Brendan Jennings, Helmut Reiser, editors, Proc. IEEE/IFIP Network Operations and Management Symposium (NOMS 2016)*, 2016.

3. R. di Lallo, M. Gradillo, G. Lospoto, C. Pisa, M. Rimondini. On the Practical Applicability of SDN Research. In, *Melike Erol-Kantarci, Brendan Jennings, Helmut Reiser, editors, Proc. IEEE/IFIP Network Operations and Management Symposium (NOMS 2016)*, 2016.

4. G. Lospoto, M. Rimondini, B. G. Vignoli, G. Di Battista. Rethinking Virtual Private Networks in the Software-Defined Era. In *Proc. IFIP/IEEE International Symposium on Integrated Network Management (IM 2015)*, 2015.

5. G. Lospoto, M. Rimondini, B. G. Vignoli, G. Di Battista. Making MPLS VPNs Manageable through the Adoption of SDN. In *Proc. IFIP/IEEE International Symposium on Integrated Network Management (IM 2015)*, 2015.

6. M. Chiesa, G. Lospoto, M. Rimondini, G. Di Battista. Intra-Domain Pathlet Routing. In *22nd International Conference on Computer Communications and Networks (IEEE ICCCN 2013)*, IEEE, pages 1-9, 2013.

Bibliography

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

[2] J. Moy, "OSPF Version 2." IETF RFC 2328, 1998.

[3] Open Networking Foundation, "Software-Defined Networking: The New Norm for Networks," Apr. `https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf`.

[4] S. Vissicchio, O. Tilmans, L. Vanbever, and J. Rexford, "Central control over distributed routing," in *SIGCOMM*, 2015.

[5] S. Vissicchio, L. Vanbever, and O. Bonaventure, "Opportunities and research challenges of hybrid software defined networks," *ACM Computer Communication Review (Editorial Zone)*, vol. 44, April 2014.

[6] S. Salsano, P. Ventre, F. Lombardo, G. Siracusano, M. Gerola, M. Santuari, E. Salvadorii, M. Campanella, and L. Prete, "Hybrid ip/sdn networking: open implementation and experiment management tools," *Network and Service Management, IEEE Transactions on*, vol. PP, no. 99, pp. 1–1, 2015.

[7] Cisco Systems, Inc., "One Platform Kit (onePK)," Sep. `http://www.cisco.com/c/en/us/products/ios-nx-os-software/onepk.html`.

[8] Open Networking Foundation, "OpenFlow Switch Specification, version 1.5.0," Jan 2015.

[9] P. B. Godfrey, I. Ganichev, S. Shenker, and I. Stoica, "Pathlet routing," *SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 4, 2009.

[10] F. K. Leonard Kleinrock, "Hierarchical routing for large networks," *Computer Networks*, no. 1, pp. 158–174, 1977.

[11] R. G. Kewei Sha, Jegnesh Gehlot, "Multipath routing techniques in wireless sensor networks: A survey," *Wireless Personal Communications*, no. 70, pp. 807–829, 2013.

[12] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin, "Resource ReSerVation Protocol (RSVP) – Version 1 Functional Specification." IETF RFC 2205, 1997.

[13] W. Xu and J. Rexford, "MIRO: multi-path interdomain routing," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 36, no. 4, 2006.

[14] M. Motiwala, M. Elmore, N. Feamster, and S. Vempala, "Path splicing," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 4, 2008.

[15] X. Yang, D. Clark, and A. W. Berger, "NIRA: A new inter-domain routing architecture," *IEEE/ACM Trans. Netw.*, vol. 15, no. 4, 2007.

[16] P. F. Tsuchiya, "The landmark hierarchy: a new hierarchy for routing in very large networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 18, no. 4, 1988.

[17] G. T. Nguyen, R. Agarwal, J. Liu, M. Caesar, P. B. Godfrey, and S. Shenker, "Slick packets," in *Proc. ACM SIGMETRICS*, 2011.

[18] V. Van den Schrieck, P. Francois, and O. Bonaventure, "BGP add-paths: the scaling/performance tradeoffs," *IEEE Journal on Sel. Areas in Commun.*, vol. 28, no. 8, 2010.

[19] I. Ganichev, B. Dai, P. B. Godfrey, and S. Shenker, "YAMR: yet another multipath routing protocol," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 5, 2010.

[20] L. Subramanian, M. Caesar, C. T. Ee, M. Handley, M. Mao, S. Shenker, and I. Stoica, "HLP: A next generation inter-domain routing protocol," *SIGCOMM Comput. Commun. Rev.*, vol. 35, no. 4, 2005.

[21] J. Behrens and J. Garcia-Luna-Aceves, “Hierarchical routing using link vectors,” in *Proc. IEEE INFOCOM*, 1998.

[22] S. Dragos and M. Collier, “Macro-routing: a new hierarchical routing protocol,” in *Proc. IEEE GLOBECOM*, 2004.

[23] M. El-Darieby, D. Petriu, and J. Rolia, “A hierarchical distributed protocol for MPLS path creation,” in *Proc. ISCC*, 2002.

[24] T. Erlebach and A. Mereu, “Path splicing with guaranteed fault tolerance,” in *Proc. IEEE GLOBECOM*, 2009.

[25] P. Jokela, A. Zahemszky, C. Esteve Rothenberg, S. Arianfar, and P. Nikander, “LIPSIN: line speed publish/subscribe inter-networking,” *SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 4, 2009.

[26] Open Networking Foundation, “OpenFlow switch specification 1.4.0,” 2013.

[27] H. Shimonishi, H. Ochiai, N. Enomoto, and A. Iwata, “Building hierarchical switch network using OpenFlow,” in *Proc. INCOS*, 2009.

[28] “OMNeT++ simulation framework,” 2016. `https://omnetpp.org/`.

[29] “INET framework for OMNeT++,” 2016. `https://inet.omnetpp.org/`.

[30] 2013. `http://www.dia.uniroma3.it/~compunet/www/view/topic.php?id=intradomainrouting`.

[31] IETF, “Locator/ID Separation Protocol working group,” 2013. `http://datatracker.ietf.org/wg/lisp/`.

[32] AT & T, “Point of View: MPLS – A strategic technology,” 2003.

[33] Frost & Sullivan, “MPLS/IP VPN services market update, 2014,” 2014.

[34] L. Cittadini, G. Di Battista, and M. Patrignani, “MPLS virtual private networks,” in *Recent Advances in Networking, Volume 1* (H. Haddadi and O. Bonaventure, eds.), ACM SIGCOMM eBook, pp. 275–304, ACM, 2013.

[35] M. Mouzuddin and M. Shaikh, “Routing issues in deploying MPLS VPNs,” in *Proc. MultiProtocol Label Switching Conference & Exhibition (MPLScon)*, 2005.

[36] Hewlett-Packard, “Intelligent Management Center – MPLS VPN Manager Software,” 2016. `http://h17007.www1.hp.com/us/en/networking/products/network-management/IMC_MPLS_VPN_Software/index.aspx#.VyoMNlWzMfI`.

[37] Cisco Systems, Inc., “Cisco Prime Provisioning,” Jan 2015.

[38] Packet Design, “Route Explorer,” 2016. `http://www.packetdesign.com/products/mpls-wan-explorer`.

[39] G. D. Battista, M. Rimondini, and G. Sadolfo, “Monitoring the status of MPLS VPN and VPLS based on BGP signaling information,” in *Proc. IEEE NOMS*, pp. 237–244, 2012.

[40] W. Enck, T. Moyer, P. McDaniel, S. Sen, P. Sebos, S. Spoerel, A. Greenberg, Y.-W. E. Sung, S. Rao, and W. Aiello, “Configuration management at massive scale: system design and experience,” *IEEE J. Sel. Areas Commun.*, vol. 27, pp. 323–335, April 2009.

[41] R. Bush and T. Griffin, “Integrity for virtual private routed networks,” in *Proc. INFOCOM*, 2003.

[42] ONRC Research, “An SDN approach to MPLS traffic engineering and virtual private networks,” 2016. `http://onrc.berkeley.edu/research_sdn_approach_to_mpls_traffic_engineering.html`.

[43] S. Das, A. Sharafat, G. Parulkar, and N. McKeown, “MPLS with a simple OPEN control plane,” in *Proc. Optical Fiber Communication Conference and Exposition and the National Fiber Optic Engineers Conference (OFC/NFOEC)*, 2011.

[44] A. R. Sharafat, S. Das, G. Parulkar, and N. McKeown, “MPLS-TE and MPLS VPNS with OpenFlow,” *SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 4, pp. 452–453, 2011.

[45] S. Das, *PAC.C: A Unified Control Architecture for Packet and Circuit Network Convergence*. PhD thesis, Stanford University, 2012.

[46] Y. Rekhter, T. Li, and S. Hares, "A Border Gateway Protocol 4 (BGP-4)." RFC 4271, Jan. 2006.

[47] T. Bates, R. Chandra, D. Katz, and Y. Rekhter, "Multiprotocol Extensions for BGP-4." RFC 4760, Jan. 2007.

[48] L. Andersson, I. Minei, and B. Thomas, "LDP Specification." RFC 5036, Oct. 2007.

[49] IEEE, "IEEE standard 802.1AB – station and media access control connectivity discovery," 2009.

[50] Corsa Technology Inc., "SDN done right," Jan 2015.

[51] Brocade, "Software-Defined Networking – OpenFlow," 2016. `http://www.brocade.com/solutions-technology/technology/software-defined-networking/openflow.page`.

[52] European Advanced Networking Test Center, "Huawei Technologies SDN showcase at SDN and OpenFlow world congress 2013," 2013. `http://www.eantc.de/fileadmin/eantc/downloads/events/2011-2015/SDNOF2013/EANTC-Huawei_SDN_Showcase-White_Paper_Final_Secure.pdf`.

[53] Arista Networks, "Arista 7280E series," 2016. `https://www.arista.com/en/products/7280e-series`.

[54] NEC Corporation, "ProgrammableFlow PF5240 Switch," Jan 2015. `http://www.necam.com/sdn/doc.cfm?t=PFlowPF5240Switch`.

[55] Broadcom Corporation, "OpenFlow – Data Plane Abstraction Networking Software," 2016. `http://www.broadcom.com/products/Switching/Software-Defined-Networking-Solutions/OF-DPA-Software`.

[56] Marvell, "Prestera DX packet processors," 2016. `http://www.marvell.com/switching/prestera-dx/`.

[57] Cisco Systems, Inc, "Cisco plug-in for OpenFlow," 2016. `http://www.cisco.com/c/en/us/td/docs/switches/datacenter/sdn/configuration/openflow-agent-nxos/cg-nxos-openflow.html`.

[58] "Open vswitch," 2016. `http://www.openvswitch.org`.

[59] "Lagopus switch," 2016. `http://lagopus.github.io/`.

[60] Big Switch Networks, "Switch Light," 2016. `http://www.bigswitch.com/products/switch-light`.

[61] Intel, "DPDK: Data Plane Development Kit," 2016. `http://dpdk.org/`.

[62] Brocade, "NetIron SDN Configuration Guide," 2016. `http://www.brocade.com/content/html/en/configuration-guide/netiron-05900a-sdnguide/index.html`.

[63] Centec Networks, "CTC5162/CTC5163 product brief," 2016. `http://www.centecnetworks.com/en/ProductList.asp?ID=189`.

[64] Y. Chiba, Y. Shinohara, and H. Shimonishi, "Source Flow: handling millions of flows on flow-based nodes," *SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 4, 2010.

[65] NEC Corporation, "NEC ProgrammableFlow Univerge PF5240 product brief," 2016. `http://www.necam.com/docs/?id=5ce9b8d9-e3f3-41de-a5c2-6bd7c9b37246`.

[66] Huawei Technologies Co., Ltd., "SDN: The best answer to campus network challenges," 2016. `http://www1.huawei.com/enapp/2679/hw-311134.htm`.

[67] M. Carbone, G. Catalli, and L. Rizzo, "Improving the performance of Open vSwitch," in *Proc. EuroBSDCon*, 2011.

[68] CPqD - R&D Center for Telecommunications, "RouteFlow," 2016. `https://sites.google.com/site/routeflow/home`.

[69] Open Networking Foundation, "Migration use cases and methods," 2016. `https://www.opennetworking.org/images/stories/downloads/sdn-resources/use-cases/Migration-WG-Use-Cases.pdf`.

[70] A. Tootoonchian and Y. Ganjali, "HyperFlow: A distributed control plane for OpenFlow," in *Proc. INM/WREN*, 2010.

[71] M. Bjorklund, “YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF).” RFC 6020, Oct. 2010.

[72] Cisco, “Cisco IOS NAT - Integration with MPLS VPN,” Jan 2015. `http://www.cisco.com/c/en/us/support/docs/ip/network-address-translation-nat/112084-ios-nat-mpls-vpn-00.html`.

[73] Roma Tre University – Computer Networks Research Group, “Software Defined Networking,” Jan 2015. `http://www.dia.uniroma3.it/~compunet/www/view/topic.php?id=sdn`.

[74] “Ryu controller,” Jan 2015.

[75] “Mininet,” 2015. `http://mininet.org`.

[76] “Mininet port numbering bug – GitHub bug report #312,” Jan 2015. `https://github.com/mininet/mininet/issues/312`.

[77] University of Adelaide, “The Internet topology zoo,” Jan 2015.

[78] Juniper Networks, Inc. and Wakefield Research, “SDN progress report,” July 2014.

[79] K. M. Tankala and S. K. Sing, “Address resolution in software-defined networks.” Patent no. WO2014115157 A1, 2014.

[80] V. Boteanu and H. Bagheri, “Minimizing ARP traffic in the AMS-IX switching platform using OpenFlow,” tech. rep., University of Amsterdam – Graduate School of Informatics, 2013.

[81] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, “Composing software-defined networks,” 2013.

[82] G. Khetrapal and S. K. Sharma, “Demistifying routing services in software-defined networking,” 2013. Aricent, Inc. White Paper.

[83] H. M. B. Moraes, R. Nunes, and D. Guedes, “DCPortalsNg: Efficient isolation of tenant networks in virtualized datacenters,” in *Proc. International Conference on Advances in Cognitive Radio (COCORA)*, 2014.

[84] Open Networking Foundation, "SDN migration considerations & use cases," Nov 2014. `https://www.opennetworking.org/images/stories/downloads/sdn-resources/solution-briefs/sb-sdn-migration-use-cases.pdf`.

[85] "Project Floodlight," 2016. `http://www.projectfloodlight.org/floodlight/`.

[86] M. M. et al., "POX SDN platform," 2016. `http://www.noxrepo.org/pox/about-pox/`.

[87] Hewlett-Packard, "HP SDN network services modules." Technical documentation, 2013.

[88] C. Kim, M. Caesar, and J. Rexford, "SEATTLE: A scalable ethernet architecture for large enterprises," *ACM Trans. Comput. Syst.*, vol. 29, no. 1, pp. 1:1–1:35, 2011.

[89] M. Scott, D. Wagner-Hall, and J. Crowcroft, "Addressing the scalability of Ethernet with MOOSE." IETF Draft draft-malc-armd-moose-00, Oct 2010.

[90] I. Aggarwal, "Implementation and evaluation of ELK, an ARP scalability enhancement." Computer Science Tripos, 2011.

[91] Ryu SDN Framework Community, "OpenFlow switch certification," 2016. `http://osrg.github.io/ryu/certification.html`.

[92] Open Networking Foundation, "ONF OpenFlow conformant: Certified product list," 2016. `https://www.opennetworking.org/openflow-conformance-certified-products`.

[93] N. Kang, Z. Liu, J. Rexford, and D. Walker, "Optimizing the "One Big Switch" abstraction in software-defined networks," in *Proc. CoNEXT*, 2013.

[94] Y. Kanizo, D. Hay, and I. Keslassy, "Palette: Distributing tables in software-defined networks," in *Proc. INFOCOM*, 2013.

[95] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, "Scalable flow-based networking with DIFANE," *SIGCOMM Comput. Commun. Rev.*, vol. 41, Aug. 2010.

[96] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "DevoFlow: Scaling flow management for high-performance networks," *SIGCOMM Comput. Commun. Rev.*, vol. 41, pp. 254–265, Aug. 2011.

[97] A. Iyer, V. Mann, and N. Samineni, "SwitchReduce: Reducing switch state and controller involvement in openflow networks," in *Proc. IFIP Networking Conference*, 2013.

[98] X.-N. Nguyen, D. Saucez, C. Barakat, and T. Turletti, "Optimizing rules placement in OpenFlow networks: Trading routing for better efficiency," in *Proc. HotSDN*, 2014.

[99] H. Zhu, M. Xu, Q. Li, J. Li, Y. Yang, and S. Li, "MDTC: An efficient approach to TCAM-based multidimensional table compression," in *Proc. IFIP Networking*, 2015.

[100] A. Gupta, L. Vanbever, M. Shahbaz, S. P. Donovan, B. Schlinker, N. Feamster, J. Rexford, S. Shenker, R. Clark, and E. Katz-Bassett, "SDX: A software defined Internet exchange," *SIGCOMM Comput. Commun. Rev.*, vol. 44, pp. 551–562, Aug. 2014.

[101] G. Lospoto, M. Rimondini, B. G. Vignoli, and G. D. Battista, "Rethinking Virtual Private Networks in the software-defined era," in *Proc. IM*, 2015.

[102] P. Sun, L. Vanbever, and J. Rexford, "Scalable programmable inbound traffic engineering," in *Proc. ACM SIGCOMM Symposium on SDN Research (SOSR)*, 2015.

[103] A. Detti, C. Pisa, S. Salsano, and N. Blefari Melazzi, "Wireless mesh software defined networks (wmSDN)," in *Proc. WiMob*, 2013.

[104] C. Monsanto, N. Foster, R. Harrison, and D. Walker, "A compiler and run-time system for network programming languages," *SIGPLAN Not.*, vol. 47, pp. 217–230, Jan. 2012.

[105] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," in *Proc. ACM International Conference on Functional Programming*, 2013.

[106] J. Reich, C. Monsanto, N. Foster, J. Rexford, and D. Walker, "Modular SDN programming with Pyretic," *USENIX ;login:*, vol. 38, no. 5, pp. 128–134, 2013.

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

[108] Hewlett-Packard Development Company, "HP OpenFlow 1.3 administrator guide (wired switches K/KA/WB 15.16)," 2016. `http://h10032.www1.hp.com/ctg/Manual/c04495114`.

[109] Dell, "OpenFlow deployment and user guide 3.0," 2016. `http://topics-cdn.dell.com/pdf/force10-sw-defined-ntw_Deployment%20Guide3_en-us.pdf`.

[110] Brocade, "Brocade NetIron Software Defined Networking (SDN) configuration guide," 2016. `http://www.brocade.com/downloads/documents/product_manuals/B_NetIron/NetIron_05800b_SDNGuide.pdf`.

[111] Arista Networks, Inc., "Arista EOS 4.14.6M user manual," 2016. `http://www.arista.com/docs/Manuals/ConfigGuide.pdf`.

[112] Extreme Networks, Inc., "ExtremeXOS 15.7 user guide," 2016. `http://extrcdn.extremenetworks.com/wp-content/uploads/2015/06/EXOS_User_Guide_15_7.pdf`.

[113] K. Kannan and S. Banerjee, "Compact TCAM: Flow entry compaction in TCAM for power aware SDN," in *Distributed Computing and Networking* (D. Frey, M. Raynal, S. Sarkar, R. Shyamasundar, and P. Sinha, eds.), vol. 7730 of *Lecture Notes in Computer Science*, pp. 439–444, Springer Berlin Heidelberg, 2013.

[114] Open Networking Foundation, "The benefits of multiple flow tables and TTPs," Feb 2015.

[115] Y. Nakagawa, K. Hyoudou, C. Lee, S. Kobayashi, O. Shiraki, and T. Shimizu, "DomainFlow: Practical flow management method using multiple flow tables in commodity switches," in *Proc. CoNEXT*, 2013.

[116] Open Networking Foundation, "OpenFlow switch specification, version 1.3.3," Sep 2013.

[117] Ryu project team, "Ryu SDN framework," 2016. `http://osrg.github.io/ryu-book/en/Ryubook.pdf`.

[118] Project Floodlight, "OFTest," 2016. `http://www.projectfloodlight.org/oftest/`.

[119] Open Networking Foundation, "Conformance test specification for OpenFlow switch specification 1.0.1," Jun 2013.

[120] Open Networking Foundation, "Basic single table conformance test profile for OpenFlow 1.3.4," Apr 2015.

[121] SDN Hub, Open Networking User Group, and Lippis Enterprises, Inc., "Open software-defined networking test for virtualized networking." Lippis Report Fall 2013.

[122] Ixia, "IxANVL OpenFlow test suite," 2016. `https://www.ixiacom.com/sites/default/files/resources/datasheet/ixanvl-openflow.pdf`.

[123] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore, "OFLOPS: An open framework for Openflow switch evaluation," in *Proc. PAM*, 2012.

[124] Veryx Technologies, "ATTEST OpenFlow switch conformance test suite," Dec 2014. `http://www.veryxtech.com/wp-content/uploads/2015/01/ATTEST-CTS_OFS_Datasheet.pdf`.

[125] Spirent Communications, Inc., "Spirent TestCenter – OpenFlow switch compliance test suite," 2016. `http://www.spirent.com/~/media/Datasheets/Broadband/PAB/SpirentTestCenter/Spirent_OpenFlow_Compliance_Test_Suite_datasheet.pdf`.

[126] M. Kuźniar, P. Perešíni, and D. Kostić, "What you need to know about SDN flow tables," in *Proc. PAM*, 2015.

[127] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer, "Dynamic scheduling of network updates," *SIGCOMM Comput. Commun. Rev.*, vol. 44, pp. 539–550, Aug. 2014.

[128] A. Gämperli, V. Kotronis, and X. Dimitropoulos, "Evaluating the effect of centralization on routing convergence on a hybrid BGP-SDN emulation framework," *SIGCOMM Comput. Commun. Rev.*, vol. 44, Aug. 2014.

[129] A. Vishnoi, R. Poddar, V. Mann, and S. Bhattacharya, "Effective switch memory management in OpenFlow networks," in *Proc. DEBS*, 2014.

[130] N. Katta, O. Alipourfard, J. Rexford, and D. Walker, "Infinite CacheFlow in software-defined networks," in *Proc. HotSDN*, 2014.

[131] S. Narayana, J. Rexford, and D. Walker, "Compiling path queries in software-defined networks," in *Proc. HotSDN*, 2014.

[132] X. Jin, J. Gossels, J. Rexford, and D. Walker, "CoVisor: A compositional hypervisor for software-defined networks," in *Proc. NSDI*, 2015.

[133] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming Protocol-Independent Packet Processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, 2014.