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

Internet eXchange Points: Current Challenges and New Opportunities

Roberto di Lallo

Spring 2018

Internet eXchange Points: Current Challenges and New Opportunities

A thesis presented by
Roberto di Lallo
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 2018

To my brother, my sun and stars.

Acknowledgments

I always wanted a professor to be as inspiring as Prof. John Keating. I thank my advisor, Giuseppe Di Battista, for being much more than that, for being an extraordinary teacher, and a precious guide. From the first day I entered our lab for my master thesis, he was able to transmit me his love for research by means of the passion, brilliance, and commitment he feels for this activity.

I would like to thank Maurizio Patrignani, Maurizio Pizzonia, and Massimo Rimondini for their fundamental advice during my PhD. They acted as co-advisors and even more.

I thank the past and present members of the Rome Tre Research Groups in Graph Drawing and Computer Networks: Patrizio Angelini, Marco Chiesa, Giordano Da Lozzo, Marco Di Bartolomeo, Valentino Di Donato, Fabrizio Frati, Federico Griscioli, Gabriele Lospoto, Vincenzo Roselli, Claudio Squarcella, and Habib Mostafaei. Working with them has been an honor and an incentive to give my best.

I thank Prof. Steve Uhlig and the Queen Mary University of London networking team – in particular Timm Bottger, Ignacio De Castro, and Eder Leao Fernandes – for hosting me during my PhD studies. They made my stay much more than just pleasant and contributed to my personal and professional growth with their sharpness, clarity, humility, and passion. Ignacio also helped me discover the major London pubs, but that is another story!

I thank Gianni Antichi for his precious advice and for giving me the opportunity to get to know a stimulating research reality like the one of Cambridge.

I thank the reviewers of this thesis, Alberto Dainotti, and Giorgio Ventre for their precious work.

I would like to thank my friends Marco, Vale, and Marcello that made my staying in London as living at home mostly because of their Nespresso coffe machine.

My thanks also go to my "tutors" Salvatore and Angela. Since we met, about a year ago, they decided to assume the challenging role of making me an adult. I do not know whether they will succeed, but they have surely become in a short time an anchor in my life. I also thank their beautiful daughters Maddalena and Mia. They can not read yet, but they can give affection, purity and laughs as only children are able to do.

Still about anchors: thanks to all my friends, the past ones, the present ones, and those who have always been close to me like Domenico, Priscilla, Denny, and Federica. They are my second family.

Alessio, Claudia, and Arianna, for remembering me constantly how lucky I am to have people like them in my life.

"Brillo" for being a brilliant friend and literally an adventure companion in our travels around the world.

My friend Simona: starting from the moment she entered my life she has been always close to me in bad and good times.

Finally, my most heartfelt thanks go to my family. My parents, for being always an example and a stimulus in the ongoing search for improvement. There is no one I would like to look like as much as them. My uncles Giuseppina and Massimo treating me as a son. Another entire life would not be enough to give them back the love they constantly gave to me. My brother, to which this thesis is dedicated. He will cease to be the most important person in my life when the sun will rise in the West and will set in the East.

Contents

Chapter 1

Introduction

Internet is defined by many as “The Network of Networks”. This is because we can imagine it as an interconnection of many components. Their name is Autonomous Systems (ASes) and they are networks connected to each other under the control of a unique administrative authority. Some Autonomous Systems offer Internet services directly to users or companies and they are called Internet Services Providers (ISPs).

The main feature of an Autonomous System is that it communicates explicitly to the rest of the Internet which networks are under its control and which networks are reachable via its infrastructures. In the course of time, th Internet has become “unbalanced”: Big portions of the Internet have became reachable just by very big Autonomous Systems paid by smaller ones to provide them access the rest of Internet – the “ Big Internet”.

In this context a special role is played by Internet eXchange Points, that are infrastructures that give ASes the opportunity to directly connect each other in order to exchange traffic without using other providers, thus reducing distances and costs. IXPs play a crucial role in the development of the Internet, encouraging ISPs to create a dense network of interconnections at low cost. Some of them (e.g., DE-CIX, AMS-IX, and LINX) have a throughput of many Tbit/sec. and are some of the most important building blocks of today’s Internet.

IXPs have also been studied as one of the causes of the evolution from a traditional hierarchical Internet to a more “flattened” version with AS-path getting shorter over time.

In this thesis, we deeply study Internet eXchange Points from a variety of view points:

- Recent trends say that major ISPs are canceling their peerings at IXPs (*de-peering*). In several occasions they justified such decision in terms of more efficient handling of IP traffic and improvement of the Quality of Services (QoS). In this thesis we try to claim the opposite, performing many measurements and showing how IXPs impact positively on Internet performance keys. We also study whether IXPs are effective in preserving traffic locality, by checking which countries are traversed to reach frequently visited Italian destinations from Italian sources;

- We comprehensively examine the evolution of IXPs over a long period of time, characterizing its evolution and quantifying the impact they have on end-to-end paths in particular with regards to the flattening phenomenon, i.e., the reduction in the number of the as-level hops. We believe that this analysis shed light on the impact over time of a critical Internet infrastructure and how it has shaped the current Internet, as well as estimates the impact of new IXPs in the Internet ecosystem;

- Very often, Route Server functionalities are mainly leveraged by small providers and Content Delivery Networks since these players have strong interests in connecting to many IXP members by just setting up a single BGP peering with RS. On the other hand, big Internet players, with very few exceptions, tend to not have BGP peerings with an RS. We argue that this trend is the result of exposing an IXP member to a potential violation of privacy in terms of BGP policies when peering with an RS. We present a Route Server (RS) system that improves both the privacy guarantees of confidential peering information and the security of the RS;

- Federated networks represent a collaborative operational way for Internet Service Providers (ISPs) to increase revenues by sharing resources. One of the main challenges in this architecture is finding a common physical place where members can connect to each other. In this thesis we claim and show how IXPs could play an important role in this sense, reducing costs for network operators and offering them many opportunities. In particular we present a Federated Network scenario where we use an IXP as a common point for the federation and we rely on Software Defined Network in order to address management and technological challenges in a flexible and customizable way. Furthermore, starting from the idea of Software Defined eXchange point(i.e. an IXP totally based on SDN) we study the applicability of many approaches proposed in literature.

We study the impact of peering at IXPs on common network key metrics by collaborating with three medium-size ISPs, in order to actively control their BGP announcements and force the traffic to take specific routes for useful comparison. We perform experiments in which network paths between two ASes either traverse IXPs or rely only on upstream providers. Such experiments help us determine to which degree IXPs are actually beneficial for involved peers. We perform measurements regarding the following network metrics: round-trip time, hop count, packet-loss, and jitter.

We try also to have an historical view on IXP: fuelled by an increasing demand for peering, IXPs grew in number, geographical scope and size, becoming a critical element of the Internet structure. In this thesis we study their evolution over a long period of time and quantify the disruptive impact of IXPs relying on comprehensive historical datasets covering a decade of the Internet evolution. We first study how the IXPs ecosystem has evolved. We then identify how the dependence on transit providers has changed over time by looking at the increasing reachability attainable by peering at existing IXPs. We show that even though nowadays there is more than the double of IXPs than ten years ago, the percentage of announced IPv4 addresses that can be reached through them has increased less than 10%, from approximately 70% to nearly 80% -even despite of the IPv4 exhaustion. Using this analysis as an starting point, we then quantify the specific impact of the emergence and growth of IXPs. By identifying IXPs in the historical traceroutes we quantify and characterize the specific impact of IXPs on how the flattening phenomenon is conflated with the stability in the average as-level path length.

Organizations that offer Internet-based services (Internet Service Providers, Content Delivery Networks, etc.) join the Internet eXchange Points (IXPs) in order to quickly and easily reach a number of other parties networks, and gain the level of connectivity they need. Currently, IXPs offer a very useful service, called Route Server (RS). An RS allows each member connected to an IXP to easily exchange traffic with other members by establishing a peering session with the RS, instead of having one peering with each other member he wants to be connected to. Peering sessions are handled by the Border Gateway Protocol (BGP), the standard interdomain routing protocol. Surely, this functionality significantly reduces the effort needed by the IXP members to connect to the Internet. However, such organizations are usually concerned with business-critical aspects as privacy of the peering relationships, privacy of routing policies and security of the network infrastructure (links, devices). We present a Route Server system that improves both the privacy guarantees of confidential peering information and the security of the RS. Our key idea is to prevent the RS from locally storing any BGP policies. Instead, the RS queries routing policies in on-demand manner by means of a second communication channel that we instantiate between the RS and each IXP member.

Internet eXchange Points can play an important role also in the future of Federated Network. Federated networks represent a remunerable operational way allowing federated partners to increase their incomes through a sharing resource process. They have been primarily used in the context of cloud computing; nowadays they are also used to provide connectivity services, like Virtual Private Networks. Federated networks represent a collaborative operational way for Internet Service Providers (ISPs) to increase revenues by sharing resources [GGT10]. A federated network can be defined as a network in which federated partners or members (e.g. ISPs) share their own resources with any other federated member in order to satisfy growing demands from customers or possibly issue value-added services (e.g. services that could not be provisioned without the federated network itself). A federated PoP is a physical place in which all ISPs involved in a federation connect each other. In general, establishing a federated PoP needs many steps, consisting of different activities. For instance, there is the need of establishing connectivity (e.g. by using dark fiber), as well as overcoming technical difficulties (e.g. due to different physical layer technologies). Other steps regard the need of installing and using new hardware (e.g. switches) that will be used by each ISP to connect to each other and all equipments to monitor the services issued by the federation. The network hardware in a federated PoP can be either hardware owned by the provider itself or shared hardware owned by the federation. It is easy to note that the federated PoP architecture strictly recall that of any Internet eXchange Point (IXP), where providers are interconnected in order to allow their customers to exchange traffic. Relying on Software Defined Network and the idea of IXP as PoP for a federated network we present a SDN-based framework. Our framework is completely based on SDN. We choose to rely on that architecture since it brings flexibility in providing services and it also makes the provisioning phase easier. Such a choice allows us to overcome the challenges in current federated networks architecture. Indeed, we identify one main problem in the architecture, namely the federated PoP. On one hand, such an interconnection point brings several benefits (e.g. clear identification of a place in which providers can federate and clear responsibilities assignment to each federated provider). On the other hand, federated PoPs are duplicates of IXPs, requiring further effort for federated providers in terms of expenses and configurations (e.g. buying and managing devices used in the federated PoP). We argue that being connected to an IXP is enough to create a federation and this requirement is easily satisfied by providers.

Often devil is in the details and also something that seems to be stable and working for years can suddenly break as we add new components. The Address Resolution Protocol (ARP) enables communication between IP-speaking nodes in a local network by reconstructing the hardware (MAC) address associated with the IP address of an interface. This is not needed in a Software-Defined Network (SDN), because each device can forward packets without the need to learn this association.We tackle the interoperability problem arising between standard network devices (end systems, routers), that rely on ARP, and SDN datapaths, that do not handle ARP packets natively. In particular, we propose a general approach to handle ARP in a SDN, that is applicable in several network scenarios, is transparent for existing devices, and can coexist with any packet forwarding logic implemented in the controller.

A milestone in the research that combines Software Defined Network and Internet eXchange Points is an approach proposed in [GVS+]. In order to foster the deployment of SDN in the network edge, the paper identifies Internet eXchange Points as a compelling place to start, given their central role in interconnecting many networks and their growing importance in bringing popular content closer to end users. It proposes an Internet eXchange Point totally based on SDN, namely a Software Defined eXchange point. Despite the fervent activity in the scientific community on devising novel network architectures and services that take advantage of SDN, most papers validate their proposals on ad-hoc testbeds, and little attention has been devoted to determining the practical applicability of these approaches using currently available devices. On the other hand, even if OpenFlow is now somewhat mature, vendors seem to lag behind in terms of functionalities supported on their devices. Without precise indications on which features are supported, network administrators interested in switching to SDN may have a hard time trying to find the selection of SDN-enabled devices that best fits their needs. Preserving this dual (scientific and technological) perspective, we contrast a selection of the most important contributions in the literature about SDN with publicly available documentation from device vendors, highlighting the consequent applicability issues that scientific contributions may incur. We define a methodology for testing the readiness of a device to operate in an SDN-based infrastructure, combining existing OpenFlow conformance test tools with other custom tests. In the end we draw a picture of the current status of OpenFlow implementations by applying our methodology to many devices.

The thesis is divided into five parts:

- Part I introduces the reader to key definitions and concepts that are essential to the understanding of the following chapters. We start with preliminaries notions about the Internet. Then we focus on IXPs and on their architectures. In this part we also introduce some definitions about Software Defined Networking;
- Part II is devoted to the impact of IXP on the Internet both from a qualitative and quantitative point of view;
- Part III focuses on Route Server (RS) and presents a solution for a RS that guarantees security and privacy;
- Part IV combines IXP and Software Defined Network and presents a solution that lets ISPs to construct a federated network dealing with many problems (administrative and technological) related to this concept. In this part we also present a solution that solves interoperability problems related to the Address Resolution Protocol (ARP). Being SDN a novelty in the industry we also analyze how research solutions deal with today SDN network devices by testing them on recent proposed approaches;
- Part V is composed of two chapters. The first one focuses on additional research activity that has little or no overlap with IXP but still yielded interesting results and publications. The last one concludes our work with a list of conference papers, and technical reports that were published during the past three years. Each publication is the tangible result of a research topic explored in one of the previous chapters of the thesis.

Part I

Preliminaries

Chapter 2

Internet eXchange Points

Internet eXchange providers play a crucial role in directly interconnecting networks of many Internet Services Providers, allowing them to avoid the usage of one or more third providers.

In this chapter we give a formal definition of Internet eXchange Point, we analyze some possible architectures and we list some IXPs in the world and in Italy.

2.1 Definition of an Internet eXchange Point

An Internet eXchange Point (IXP) is a physical infrastructure through which Internet Service Providers (ISPs) and Content Delivery Networks (CDNs) exchange Internet traffic between their networks (Autonomous Systems). It allows ISPs to interconnect directly their Autonomous Systems, namely their networks, that is to establish a peering between them, instead of using a third party network (upstream).

Fig. 2.1 shows a simplified model of Internet: AS5 and AS6 most likely will exchange their traffic using AS4 while AS2 and AS5 as well as AS3 and AS4 are directly interconnected (i.e. they have a peering) and so they do not need to send their traffic through another AS. As detailed in [CSFW] this is convenient in terms of costs and performance. Indeed usually traffic that pass through an IXP is not billed by any party.e On the contrary, traffic that goes through an upstream is subject to standard Commercial Agreements. Furthermore, in most cases, traffic paths that traverse IXPs are shorter than the ones that go

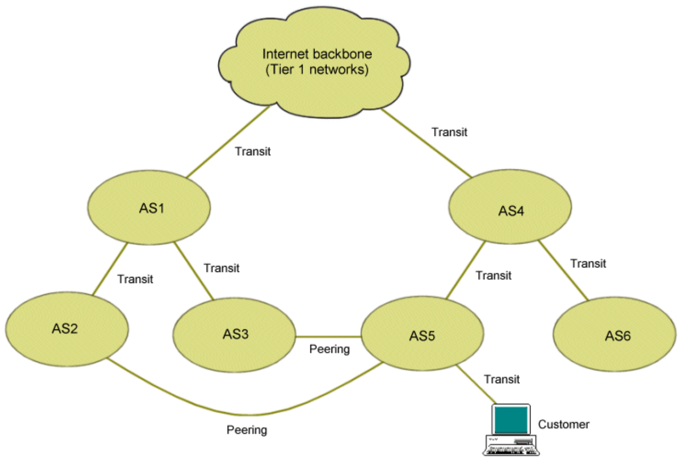


Figure 2.1: A little Internet

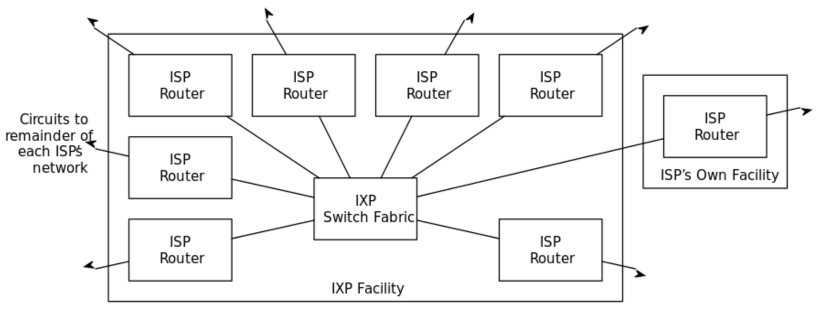
through an upstream. This means shorter distance, lower number of traversed devices and therefore benefits on the end user performance.

2.2 Architecture of an Internet eXchange Point

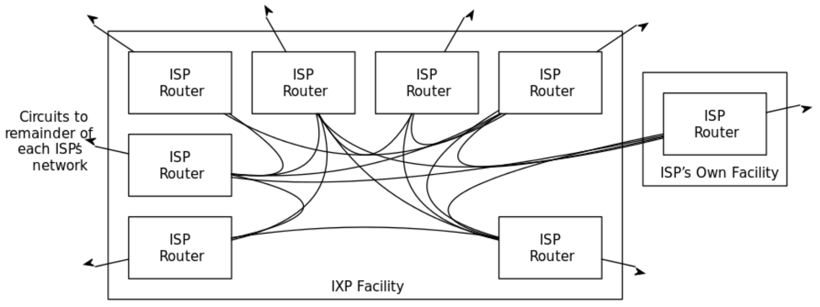
A typical IXP consists of one or more network switches, to which each of the participating ISPs connect its router. Traffic exchange between participants is facilitated by Border Gateway Protocol (BGP) routing configurations between them. They choose to announce routes via the peering relationship - either routes to their own addresses, or routes to addresses of other ISPs that they connect to, possibly via other mechanisms. The other party to the peering can then apply route filtering, where it chooses to accept those routes, and route traffic accordingly, or to ignore those routes, and use other routes to reach those addresses.

Fig. 2.2 shows a simplified topology of an IXP from two perspectives: considering the topology at the Layer 1 (Physical) and Layer 2 (Data Link) and then at the Layer 3 (Network). It is easy to see how the network level benefits this topology having available potentially connections with all the participants but using just one physical link.

Fig. 2.3 represents the topology of one of the biggest IXPs in the world,



(a) Diagram of the Layer 1 (physical) and Layer 2 (Data Link) topology of an Internet Exchange Point (IXP)



(b) Diagram of the Layer 3 (network) topology of an Internet Exchange Point (IXP).

Figure 2.2: Diagram of an IXP from different perspectives (Layers)

DE-CIX. It clearly shows the redundancy applied to the aggregations level (e.g. DE-CIX3 and DE-CIX6 are the redundancy of the core) and the distributed architecture of this IXP (indeed DE-CIX1-4 and DE-CIX7 are the five points of DE-CIX in the city of Frankfurt).

The entire infrastructure of an Internet eXchange Point could be located in a single physical place (e.g. the Cairo Internet eXchange CAIX), in more places in the same city (e.g. DE-CIX) or in the same region (e.g. the ECIX), or it could be distributed globally (e.g. the Equinix Internet eXchange Point is distributed in 19 locations spread in 17 different metropolitan areas). Moreover some IXPs are made up by multiple geographically dispersed not-interconnected

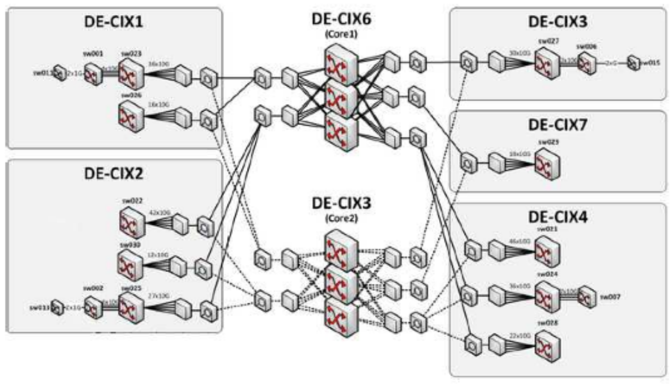


Figure 2.3: DE-CIX IXP architecture at Frankfurt in 2012

network infrastructures (i.e., an “IXP of IXPs”; e.g., Netnod in Scandinavia, or AMS-IX in Amsterdam AMS-IX and its “branch” AMSIX HK in Hong Kong).

However, irrespectively of their size and architecture, IXPs typically deploy a fully redundant switching fabric to provide an extra level of fault-tolerance and house their equipment in facilities such as data centers that are known for high levels of reliability (e.g., full UPS power backup, 99.999999% uptime), power density (e.g., heating, ventilation, AC), and security (e.g., facility access control and monitoring).

2.3 IXPs in the World

The growth of the number of Internet eXchange Points in the world has been somewhere linear as shown in Fig. 2.4 by [ana]. Starting from about 50 IXPs in 1999 they became about 400 in 2011, with the highest concentration in Europe.

The highest concentration of IXPs in Europe can be also observed in Fig. 2.5 from [pch] that shows the geolocation of all IXPs in the world. Still [pch] shows in Fig. 2.6 the distribution of IXPs in the world aggregated by nation (darker blue tones indicate a greater number of IXPs). Table 2.1 [int] contains IXPs with more than 100 members.

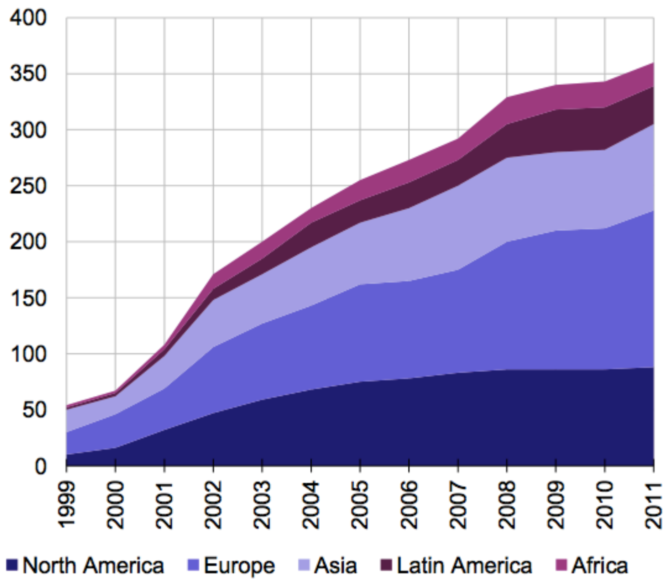


Figure 2.4: Growth of IXPs in the year by region

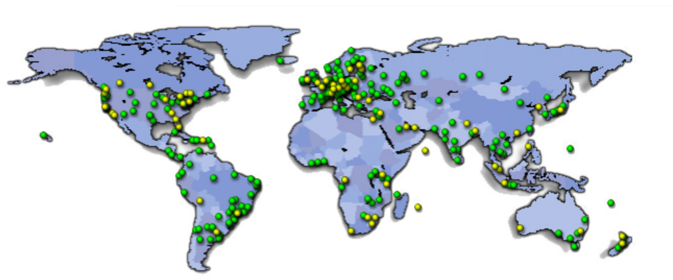


Figure 2.5: Geolocation of IXPs in the world

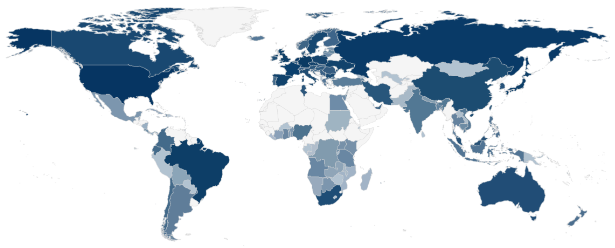


Figure 2.6: Geolocation of IXPs in the world

Continent	**Nation**	**City**	**Name**	**# of participants**
Europe	Holland	Amsterdam	Amsterdam Internet Exchange	472
Europe	England	London	London Internet Exchange	407
Europe	Russia	Moscow	Moscow Internet Exchange	344
Europe	Germany	Frankfurt	Deutscher Commercial Internet Exchange	325
Europe	Holland	Amsterdam	Netherlands Internet Exchange	294
North America	USA	Los Angeles	One Wilshire Any2 Exchange	216
Europe	Poland	Warsaw	Polish Internet Exchange	204
Latin America	Brazil	St. Paul	Ponto de Troca de Trafego Metro	167
Europe	England	London	XchangePoint London IPP	166
North America	USA	Seattle	Seattle Internet Exchange	151
Africa	South Africa	Cape Town	Cape Town Internet Exchange	150
Europe	Germany	Frankfurt	KleyReX Internet Exchange	138
North America	USA	New York	New York International Internet Exchange	137
Europe	France	Paris	Paris NAP	133
Europe	Switzerland	Zurich	SwissIX	132
Asia-Pacific	japan	Tokyo	Japan Internet Exchange	125
North America	Canada	Toronto	Toronto Internet Exchange	116
Europe	England	London	London Network Access Point	114
Europe	France	Paris	Free-IX	106
Asia-Pacific	Australia	Sydney	PIPE Networks Sydney	105
Europe	Austria	Vienna	Vienna Internet Exchange	105
Asia-Pacific	China	Hong Kong	Hong Kong Internet Exchange	104
Europe	Italy	Milan	Milan Internet Exchange	102

Table 2.1: IXPs in the world with more than 100 members (2012).

Chapter 3

Software Defined Networking

Traditional IP networks are complex and very hard to manage. It is both difficult to configure the network according to predefined policies, and to reconfigure it to respond to faults, load and changes. Software-Defined Networking (SDN) is an emerging paradigm that promises to change this state separating the network's control logic from the underlying routers and switches, promoting (logical) centralization of network control, and introducing the ability to program the network. The separation of concerns introduced between the definition of network policies, their implementation in switching hardware, and the forwarding of traffic, is key to the desired flexibility: by breaking the network control problem into tractable pieces, SDN makes it easier to create and introduce new abstractions in networking, simplifying network management and facilitating network evolution. [KRV+15]

3.1 The Idea of a Programmable Network

Software defined networking is a centralized routing paradigm. It divides physically control plane and data plane. Control functions are centralized in a server that is able to compute a path in the network for each traffic flow. It allows the administrator of a network to have a fine control on each path.

As Fig. 3.1 shows, the difference between SDN and traditional routing, is that while in standard networks the routing is computed in a distributed manner, in SDN it is computed by a software executed on a machine. We can sum up this concept as Code VS Configuration.

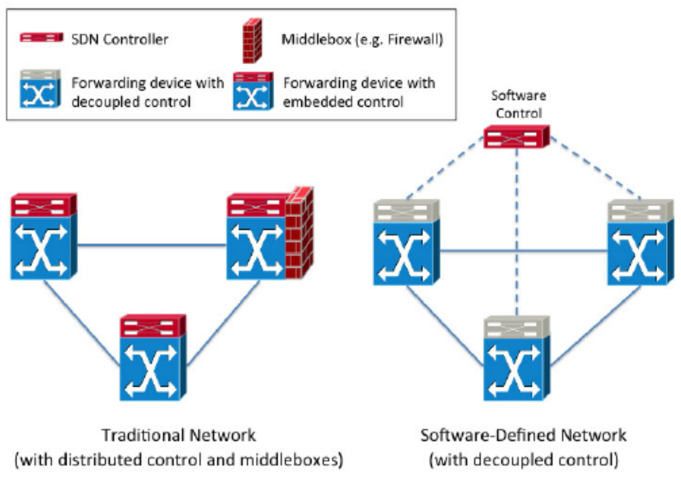

SDN Controller
Middlebox (e.g. Firewall)
Forwarding device with decoupled control
Forwarding device with embedded control
Software Control
Traditional Network (with distributed control and middleboxes)
Software-Defined Network (with decoupled control)

Figure 3.1

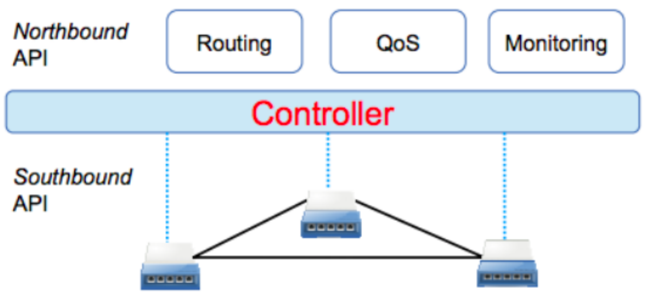


Figure 3.2: Architecture of a Software Defined Network

3.2 Architecture of a Software Defined Network

SDN divides control plane and data plane, we have two main actors:

- Controller: it runs the control plane software. It could be executed on general purpose hardware;

- Switch (or Datapath): is functionalities are limited to packets forwarding.

Controller and datapaths can communicate using specific API. The separation of the forwarding hardware from the control logic allows easier deployment of new protocols and applications, straightforward network visualization and management, and consolidation of various middleboxes into software control. Instead of enforcing policies and running protocols on a convolution of scattered devices, the network is reduced to "simple" forwarding hardware and the decision-making network controller [NMN$^+$14]. In the SDN model, the functional logic of a network device, called datapath, is realized by a piece of software called controller, while the device itself only performs packet forwarding based on a set of match-action conditions called flow entries. When a datapath does not know how to handle a packet, it submits it to the controller. The controller may either ask the datapath to emit a copy of the packet out of a specific interface, or install a new flow entry in the datapath's flow table that instructs the datapath about how to independently handle future packets belonging to the same flow: such an entry watches certain bits of the header fields of received packets while applying wildcards on other bits, and it executes an action on matching packets (e.g., forward out of a port, drop). The most widely adopted specification of SDN is OpenFlow [Ope13a] , which also defines a protocol for controller-datapath communication.

3.3 OpenFlow

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 also offer OpenFlow support as a compatibility plug-in. 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 thesis we refer to the most recent version, 1.5.1. 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

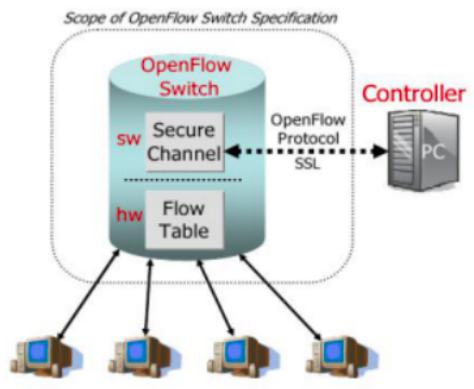


Figure 3.3: OpenFlow architecture

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 con- 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. 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. Although support for 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. 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.

Part II

IXP's Impact on the Internet

Chapter 4

Through the Lens of IXPs: A Decade of Internet Evolution

Internet eXchange Points (IXPs) have been a key element reshaping the Internet over the past decade: While previous works have provided punctual evidence of the relevance of IXPs, there is no comprehensive study and there is very limited analysis on the temporal evolution of this critical infrastructures. This work comprehensively examines the evolution of IXPs over a long period of time, characterizes its evolution and quantifies the impact they have on IPs reachability. We believe that this analysis shed light on the impact over time of a critical Internet infrastructure and how it has shaped the current Internet, as well as estimates the impact of new IXPs in the Internet ecosystem.

4.1 Introduction

With the growth of traffic and increasing demands of quality and performance, IXPs emerged as a switching facility where networks interconnect through peering links. Fuelled by an increasing demand for peering, IXPs grew in number [AKW09], geographical scope [CCGF14] and size [ACF^{+}12], becoming a critical element of the Internet structure [GILO11b,BCT^{+}16,CSFW15,CDA^{+}16, RSF^{+}14a]. While previous works provided a limited punctual analysis, in this work we provide a comprehensive study of IXPs and its evolution over a long period of time and quantify the disruptive impact of IXPs. This work, thus provides a fine-grained understanding of the impact and temporal trajectory of IXP's critical infrastructure, quantifying the consequences of its massive spread.

With IXPs being an ideal vantage point [CSB+13], we rely on comprehensive historical datasets covering a decade of the Internet evolution. We first study how the IXP ecosystem has evolved. We then identify how the dependence on transit providers has changed over time by looking at the increasing reachability attainable by peering at existing IXPs. We show that even though nowadays there is more than the double of IXPs than ten years ago, the percentage of announced IPv4 addresses that can be reached through them has increased less than 10%, from approximately 70% to nearly 80% -even despite of the IPv4 exhaustion. Note that we do not consider in this study IPv6, as its impact has become tangible only in the last few years [CAZ+14].

The main contributions of this work are:

- an analysis of the the evolution of IXPs.
- a quantification of the impact of IXPs on the Internet ecosystem.
- a forecast of the potential impact of new IXPs.

4.2 Related Work

While the academic community slowly acknowledged the relevance of IXPs in terms of the number of facilities [AKW09], its geographical scope [CCGF14], size [ACF+12], structural impact [GILO11b] and relevance [BCT+16, CSFW15, CDA+16, RSF+14a], little has been explored with regards to its temporal dynamics.

While some works have taken a look at the Internet evolution as a whole [DD11, LIJM+10], little is know about the specific role of IXPs over time. Cardona et al. [CRS12a] is probably one of the few exceptions, though the work refers just to one specific IXP and to its internal temporal dynamics and it is not possible to expand those insights to the overall IXP ecosystem.

Aligned with the rising relevance of IXPs, large networks expanded their geographical coverage [KMS+09] and increased peering, typically at IXPs, allowed networks to circumvent transit providers [LIJM+10].

While these factors have pointed to a flattening of the Internet topology [GALM08], the Autonomous System (AS)-level path length has remained stable [DD11].

In understanding peering dynamics and IXPs, PeeringDB [LLD+14] has been shown to be a reliable data source, which we also use in this work.

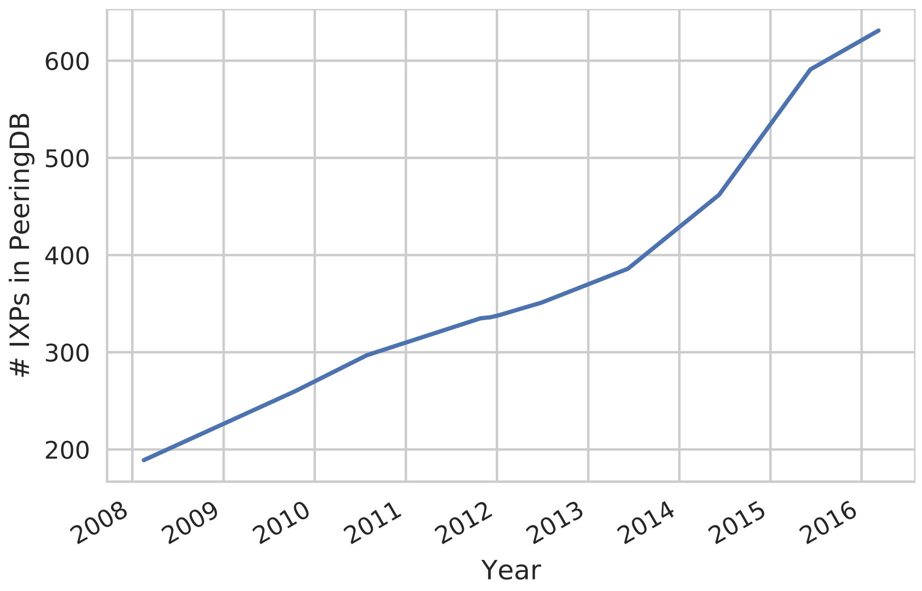


Figure 4.1: Number of IXPs per PeeringDB snapshot

Similarly, and despite of its limitations [MRWK03], traceroute repositories, such as iPLane [MIP+06], can provide a representative picture of the Internet [SW09].

4.3 Data

We rely on a large historical IXP membership dataset from PeeringDB [LLD+14] that covers from 2008 to the present. We obtained the years 2008 and 2009 using the Way Back Machine [1], from 2010 to 2016 from CAIDA [2], and we obtained 2017 directly from the PeeringDB site. In Figure 4.1 we show the number of IXPs for each PeeringDB snapshot. It is clear how their growth follows a linear trend. Starting from about 200 IXPs in 2008 we reach more than 600 in 2016. The same is shown in Figure 4.2 where results are aggregated per region. We have obviously the same trend, while is interesting to see how IXPs have been always more in Europe and only in the last few years we get

[1]`http://archive.org/web/`
[2]`http://data.caida.org/datasets/peeringdb-v1/`

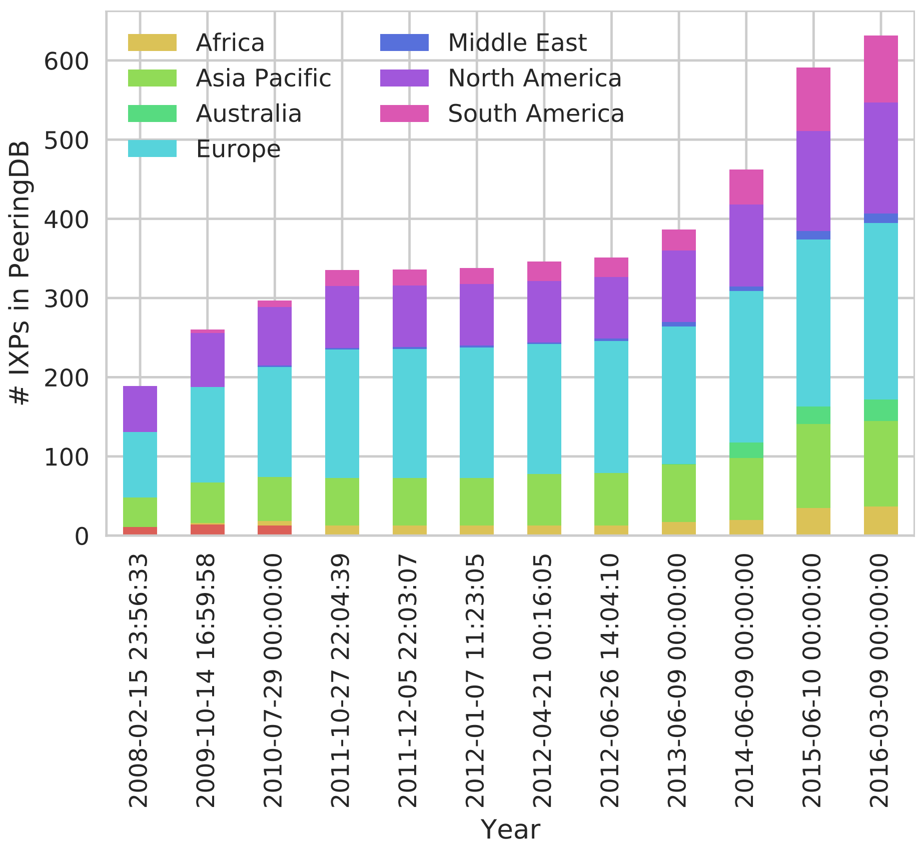


Figure 4.2: Number of IXPs per PeeringDB snapshot

a sensible growth in South America and Africa. Moreover, in the Midlle Est, and in Australia, we find a more reasonable number of them just in the last few years.

We also leverage the information of PeeringDB on the ASes willingness to peer to give a more realistic picture. ASes declare in PeeringDB whether they have a peering policy which is open (willing to peer with any network), selective (peering only takes place if some conditions are met), or restrictive (only peering with specific networks). To identify IXPs in Internet paths, we rely on the historical iPlane dataset [MIP+06] and we use Maxmind IP-geomapping datasets from the Wayback machine for each corresponding year.

4.4 Internet eXchange Points' rechability

Firstly, we define the reachability of an IXP as the set of unique IP addresses that are reachable through customer links in such IXP, i.e., the IPs announced by the IXP members and their customer cones. This metric informs of which share of the address space could be potentially reached if an AS would colocate at such IXP and peer with all its members.

We consider a global view of the Internet to deal with the disparity of perspectives that different vantage points provide. In particular, use the ASes'customer cones as computed by CAIDA [LHD+13]. We also consider the reachability of an AS as the the set of IPs reachable through the prefixes announced by such AS any of its customers. Note that while operationally wise, ASes would rather reach a given IP via the most specific prefix, we are interested in feasibility rather than in operational correctness.

4.5 Transit dependence

Because tier 1 ASes are regarded as the ultimate guarantors of universal reachability, i.e., all announced address space are supposed to be reachable through Tier 1 ASes (T1), we discard the T1 from the IXP customer cone. We regard as T1 the clique of the AS graph, i.e., the set of ASes that have only peering relationships with each other, no providers, and whose joint customer cone is the announced address space. More specifically we use the clique as inferred in [LHD+13]. As peering has been an alternative interconnection mode allowing ASes to bypass transit providers (including tier 1), we are particularly interested in how dependent are ASes on tier 1s, i.e., what is the percentage of the address space that is unreachable via IXPs.

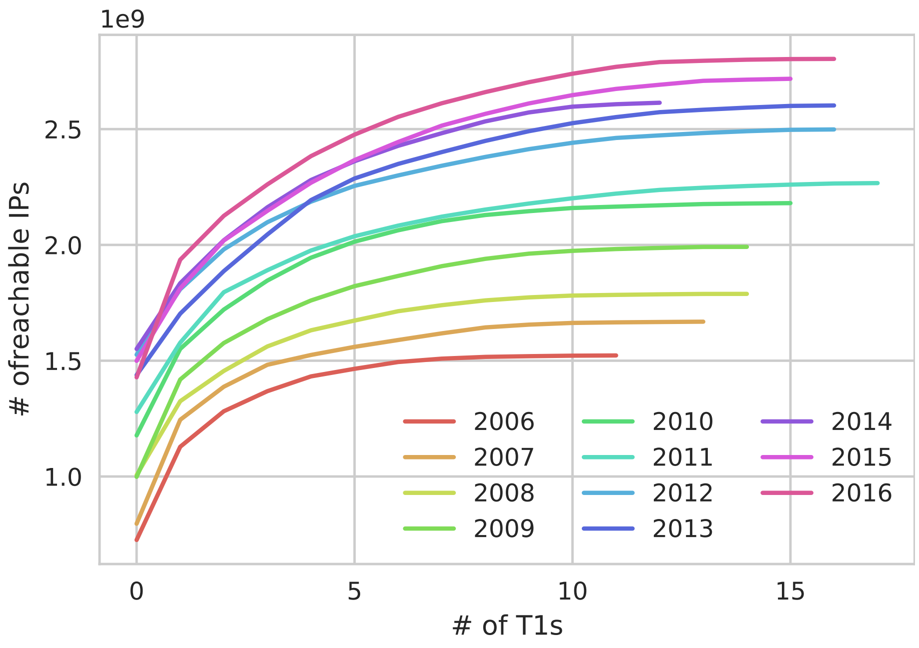


Figure 4.3: Cumulative reach of T1 networks per year

We now depict the reachability from IXPs perspective with reachable IPs as a function of IXP presence (i.e., number of IXPs in the x-axis and number of unique reachable IPs in the y-axis). IPs are sorted in terms of "accumulated reachability": the first IXP, is that one IXP with the largest reachability in terms of IP address, followed by the IXP adding most new IPs. We do this recursively until we exhaust the set of existing IXPs. As a result we can observe in Figure 4.5 both how much of the address space reachable through a tier-1 can be reached through IXPs, how many IXPs are needed to maximize such reachability, and how this has evolve over time.

Comparing Figure 4.3 with Figure 4.5 it is trivial to see how just a little portion of the whole Internet Ecosystem is still reachable just through T1s. It also easy to imagine that it consists exactly of space belonging directly to T1s Autonomous Systems.

In Figure 4.4 we have reachability through IXPs including T1s networks. Compared with Figure 4.3 it shows how the final value for reachability is the

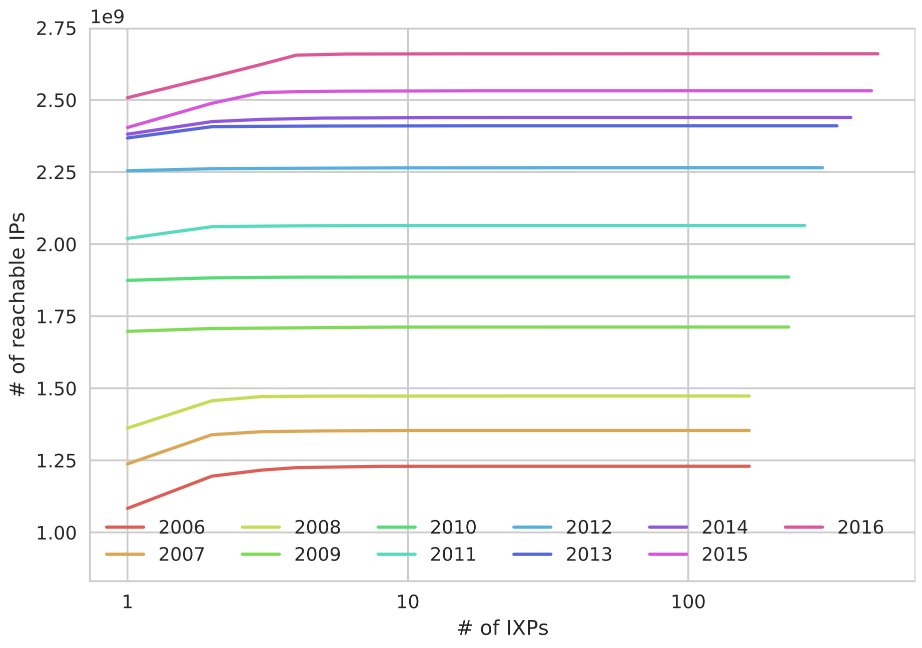


Figure 4.4: Cumulative reach of IXPs incuding T1 networks

same. This means that we do not have IPs reachable only through IXPs.

Interestingly, the percentage of destinations reachable through IXPs (relative to the total number of IPs reachable through T1s) has not steadily increased over the years, as portrayed in Figure 4.5. For instance, in 2011 a greater share of the public Internet was reachable through IXPs than in ulterior years, being 2016 the only exception. The reason is rather in the dynamics of IXPs than in abrupt changes in T1’s reachability: as Figure 4.3 demonstrates, the total number of destinations reachable through the T1 has been constantly increasing, though not steadily. For example, the earlier of years of the analysed period, witnessed a greater yearly increase in the reachability of the T1 than later years, a natural phenomenon if we take IPv4 exhaustion into account. The top level exhaustion started on early 2011, when Number Resource Organization (NRO) announced that the free pool of available IPv4 addresses is now fully depleted. Soon after the Internet Assigned Numbers Authority (IANA) allocated the

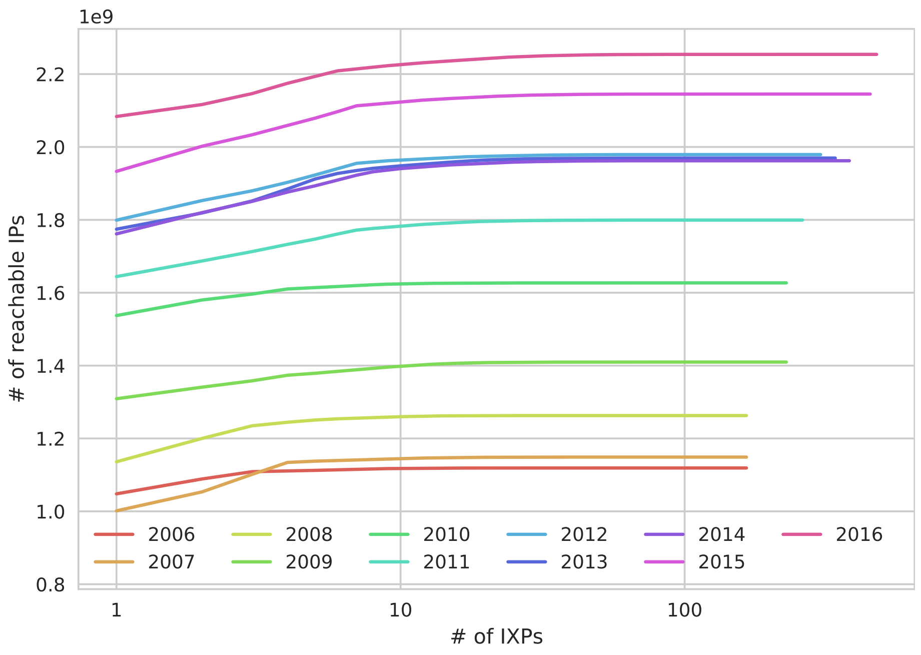


Figure 4.5: Cumulative reach of IXPs without T1 networks

remaining IPv4 blocks to the 5 Regional Internet Registry (RIR) [3]

4.6 Conclusions

In this work we showed how IXPs have been a key element reshaping the Internet in the last ten years. We comprehensively studied and analyzed their temporal evolution and their impact on the reachability of the whole Internet ecosystem. Moreover we depicted a forecast of the potential impact of new IXPs.

We believe that with this work we took a snapshot of the Internet and we shed light into the impact over time of a critical Internet infrastructure.

[3]`https://www.nro.net/ipv4-free-pool-depleted/`

Chapter 5

Is it Really Worth to Peer at IXPs?

Internet Exchange Points (IXPs) play a crucial role in the Internet ecosystem. However, existing literature fails in quantitatively assessing the advantage for an Internet Service Provider (ISP) to peer at an IXP. We give a contribution to bridge such a gap by collaborating with three medium-sized ISPs in Italy to compare key performance indicators (round-trip delay, hop count, packet loss, and jitter) as measured from several vantage points in presence and absence of IXP peerings. Our findings are that IXP-based paths exhibit better and more stable performance, whereas avoiding IXPs introduces performance deterioration and higher variability. Moreover, our measurements confirm that IXP-based paths tend to preserve the locality of traffic.

5.1 Introduction

Internet eXchange Points (IXPs) are infrastructures used by *Internet Service Providers* (ISPs) to exchange traffic between their *Autonomous Systems* (ASes). An IXP allows ISPs to interconnect their ASes directly, i.e. to establish *peerings* between them, rather than through third-party networks (*upstreams*). IXPs play a crucial role in the development of the Internet, encouraging ISPs to create a dense network of interconnections at low cost. Some of them (e.g., DE-CIX, AMS-IX, and LINX) have a throughput of many Tbit/sec. and are some of the most important building blocks of today's Internet [ACF$^+$].

Despite the importance of IXPs, scientific literature lacks extensive studies that show the impact of peering at IXPs on common network key metrics. In [CSFW], e.g., the authors claim that having peerings at an IXP is efficient in terms of performance, but their assumption is not backed by real data. The main limitation lies in the fact that thorough studies need to directly compare the performance of traffic flowing through an IXP as opposed to that through upstream providers.

In our work we make a first attempt to fill this gap by collaborating with three medium-size ISPs, in order to actively control their BGP announcements and force the traffic to take specific routes for useful comparison. More in detail, we perform experiments in which network paths between two ASes either traverse IXPs or rely only on upstream providers. Such experiments help us determine to which degree IXPs are actually beneficial for involved peers. We perform measurements regarding the following network metrics: *round-trip time* (rtt), *hop count*, *packet-loss*, and *jitter*. Our study is focused on Italy, meaning that both the source ASes and the target ASes of our measurements are located in Italy. We concentrate our attention on the two most famous Italian IXPs: MIX and NaMeX. Our study stems from recent news [dep] on major ISPs canceling their peerings at IXPs (*de-peering*). In several occasions they justified such decision in terms of more efficient handling of IP traffic and improvement of the QoS. As a second topic of interest, we also study whether IXPs are effective in preserving traffic locality, by checking which countries are traversed to reach frequently visited Italian destinations from Italian sources. This is a security issue of concern for many nations. In particular, given the recent espionage cases of network traffic [dat], a nation might be interested to see if its own citizens, to reach Internet services that are considered critical, have to cross ISPs of different countries or even of different continents [rus].

This chapter is organized as follows. Section 5.2 discusses the state of the art in the analysis of IXPs. In Section 5.3 we describe the different experiments that we conducted to assess the role of IXPs on the Italian Internet ecosystem. Section 5.4 details the results of our analysis. Finally, Section 5.5 presents our conclusions and hints for future work.

5.2 Related Work

A survey by Chatzis et al. [CSFW] lists three main reasons for an ISP to establish peerings at an IXP: saving money, improving the QoS, and meeting the requirements of big players (e.g. Google, Netflix) that incentivize other networks to connect at IXPs to obtain peerings with them. However, no experiments have been performed by the authors to support the claim on the QoS. The work also examines the diversity of IXPs around the world, highlighting the non-profit nature and the potential for innovation of European Exchange Points as opposed to standard commercial offers in North America. [CRS12b] analyses data on the peerings at Slovak IX, observing that many of them carry only little traffic and therefore their existence would not be economically viable outside of an IXP. Recent research work [ACF$^+$, CSB$^+$] proves that large IXPs host thousands of peerings between heterogeneous network that carry petabytes of traffic on a daily basis. Further, the analysis of traffic logs leads to interesting findings about trends in the Internet ecosystem, e.g. about content delivery networks and about the impact of IXP pricing models on the nature of peerings.

The discovery of IXP-based peerings is a related topic that has traditionally caught the attention of researchers for its implications in other areas, e.g. the study of the correspondence between IP addresses and ASes. Recent techniques (see, e.g., [AKW]) prove that most of such peerings cannot be found by simply mining traditional, "static" data sets like BGP routing data and Internet registry statistic files.

IXPs have also been studied [GILO11a, MHC] as one of the causes of the evolution from a traditional hierarchical Internet to a more "flattened" version with AS-path getting shorter over time.

The renewed interest in the topic is confirmed by recent research work that aims at expanding the potential of traditional IXPs. Kim et al. [KSF14] used logs of a large European IXP to study the status of IPv6 traffic during and after the World IPv6 Day in 2011 and the World IPv6 Launch in 2012. Gupta et al. recently introduced a prototypical software-defined Internet Exchange that enables expressive and highly customizable routing policies [GVS$^+$].

Despite of this large amount of investigation, a quantitative analysis on the impact of IXPs on key performance indicators like rtt, jitter, hop count, and packet loss is missing. In [AG], the authors compare the rtt of a path between two points in the Internet when the path traverses an IXP and when it does not. Differently from us, their non-IXP path is simulated, by concatenating pieces of paths coming from several traceroutes. Namely, they find a common provider C of the source AS S and the destination AS D, and then find a $S - C - D$ path in a traceroute graph computed from public traceroute datasets, such that an IXP is not traversed. Their conclusions are that the non-IXP paths have lower latency, which is in contrast with our results. In [GCF$^+$], the role of African IXPs is studied. Among other measurements, they perform traceroutes from 17 local probes towards fixed targets. They discover that many African ISPs do not have peerings at local IXPs, making local traffic paths often detour through Europe. They compute the improvement in rtt that would result by following a simulated path through an IXP. The simulation is done by replacing the detour part of a path with a direct link through an IXP or between pairs of local IXPs, for which they estimate the latency. Their conclusions are that African users would benefit from a higher connectivity between providers in African IXPs.

To our knowledge, all existing works that assess the effectiveness of IXPs by comparing key performance indicators use, at least partially, simulated data. The reason is that it is difficult to collect both a path traversing an IXP and one traversing only upstreams between a pair of hosts in a real-life network, where no direct control is possible. Supported by several ISPs, we fill this gap by comparing key performance indicators of real IXP and non-IXP paths.

5.3 Methodology

The IXPs that are central to our study are the two oldest and most popular Italian IXPs, namely, MIX and NaMeX. MIX [MIX] is the largest in terms of peers: at the time of writing, its Website reports that 151 ISPs have peerings there, of which 106 flagged as "Italian". We executed the procedure described in Section 5.4 to determine the nationality of the ISPs declared to be Italian by MIX, and it confirmed the correctness of such declaration. NaMeX [NaM] currently serves 58 ISPs, as reported on their website. They are not classified based on their nationality, therefore we executed the aforementioned procedure and conclude that 47 of them are Italian. The number of peerings at MIX and at NaMeX is not publicly available. Executives at MIX kindly provided an estimation of more than $5,000$ peerings in their IXP, while executives at NaMeX kindly disclosed an estimation of 438 peerings, i.e. 31.78% of the theoretical upper bound of 1378.

In our study we exploited the distributed infrastructure of RIPE Atlas [atl], which is a Internet measurement network based on thousands of devices, called *probes*, that are deployed in different types of networks (home, business, academic, etc) all around the world. Probes are connected to the Internet and can be remotely activated to schedule and perform standard network measurements (i.e. ping, traceroute, DNS resolution) towards arbitrary Internet targets. Each probe is labeled with its approximate location. We leverage RIPE Atlas to perform traceroute and pings from probes located in Italy and directed at a number of targets in Italy. The location of the probes used in our experiments is shown in Fig. 5.1. Several alternatives for distributed active network measurements are available nowadays. Examples include CAIDA's Archipelago [ark], Measurement Lab [mla], and SamKnows [sam]. However, the first two have very few probes in Italy, while the probes of the last are allocated to specific

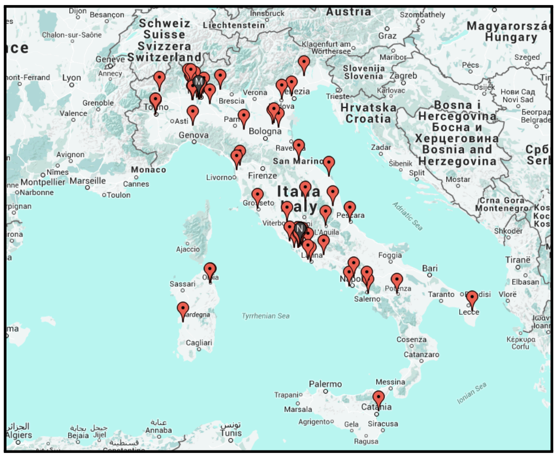


Figure 5.1: Maps showing the location of probes used in our experiments. MIX and NaMeX are marked with letters `M` and `N`, respectively.

commercial targets and are not publicly available for research purposes.

Experiment CIS: Crucial Internet Services

In our first experiment we engineered network tests aimed at producing a plausible snapshot of the QoS associated to crucial websites commonly accessed by Italian users, with the goal of classifying the results according to the usage/non-usage of IXP peerings. As a first step, we selected two sets of Internet websites that can be considered crucial for Italian users. The first contains services related to critical infrastructures and the second contains popular sites. The first set, called CRITICAL, has been selected according to the taxonomy in [Hom] and contains 46 websites belonging to the following categories: on-line banking, insurance companies, public administrations, energy companies, legal courts, travel companies, health portals, and webmail providers. The second list, called VISITED, contains the 100 Italian websites most visited by Italian users, according to the ranking published by Alexa [Ale]. We mapped each domain name belonging to our lists to an IP address as follows. We scheduled ping measurements from all Italian probes targeted at each domain, once every 10 minutes for four hours and specifically forced probes to resolve the domain before each ping by querying their local DNS. We have then filtered the obtained

IP addresses by keeping only those located in Italy according to the heuristic described in Section 5.4. The IP addresses obtained in this way are 53 for the first list and 94 for the second one. In the latter case we have fewer IP addresses than websites because some websites share the IP address. For each list, the test of Experiment CIS consists of 5 traceroutes, performed by each probe at intervals of 10 minutes, for a total duration of 40 minutes. We intentionally performed few traceroutes in order to limit the total duration of the experiment, thus reducing the probability of measuring noise due to routing changes. Based on the collected traceroute data, for each traceroute, we computed four values: rtt, hop count, and two boolean flags telling whether MIX and NaMeX were traversed. To compute such flags we preliminarily collected all IP addresses used at the two IXPs for the establishment of peerings, by directly asking representatives of the IXPs. With such information at hand, we were able to flag all traceroute paths passing through either of the two IXPs.

Experiment SBA: Selective BGP Announcements

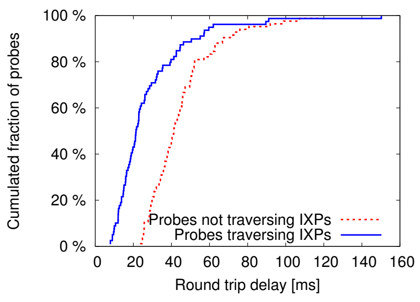
In our second experiment we focused on discovering routing alternatives involving either IXPs or upstreams used by Italian ISPs to reach Internet services, in order to directly compare their performance. To do so we partnered with Mc-link, Seeweb and Unidata, three medium-sized Italian ISPs. In preparation for our experiment, we asked each partner ISP to reserve one IP subnet and one server in its data center. Each server was assigned an IP address falling within the reserved IP subnet. Also, it was configured to correctly handle and answer ICMP echo requests sent with pings and traceroutes. For each ISP we executed, at different times, the steps described below. We asked the ISP to announce a predetermined sequence of 5 BGP updates involving its reserved IP subnet. Each update had a lifetime of 4 hours, for a total of 20 nonconsecutive hours per test. The list of BGP updates was crafted to selectively distribute routes to different subsets of the available peers, adhering to the following scheme. 1) *"UPSTREAM"*: announce only to transit ASes. 2) *"IXPS"*: announce only to MIX and NaMeX peers. 3) *"MIX"*: announce only to MIX peers. 4) *"NAMEX"*: announce only to NaMeX peers. 5) *"ALL"*: announce to all peers. During each interval we ran traceroutes and pings from all Italian probes to the IP address of the server. For each interval we started one hour before the BGP announcement and ended one hour after the end of the interval. We performed one ping per minute and one traceroute every 10 minutes. The measurements ran before the first BGP announcement was performed, in order to assess the reachability and the performance stability of each target. The great majority of the intervals (not all of them because of technical constraints of one provider) were located during the daytime hours, to avoid any time-of-day effect on the measures [GM]. Finally, at the end of each period, we filtered out all ping and traceroute measurements that took place near the 5 BGP announcements. More precisely, given the exact timestamp logs of BGP announcements, we identified 4 time intervals spanning one hour and centered at each of the timestamps, and removed each ping and traceroute falling within any of the intervals. We did so to account for potential instabilities and route flapping caused by the BGP announcements. For each probe-target-interval tuple, where an interval is one between *"UPSTREAM"*, *"IXPS"*, *"MIX"*, *"NAMEX"*, and *"ALL"*, we computed the average rtt, the average hop count, and the boolean flags. Observe that, while in EXPERIMENT CIS we took into account raw measured data, in EXPERIMENT SBA we considered average values. This was done to compare the performance of a probe-target pair measured in the different intervals. Further, from ping data, we computed both the packet loss, measured as the sum of all the ratios between the number of packets that received no answer and the total number of sent packets, and the jitter, computed as the standard deviation of all rtt values. Unidata and MC-link have one data center each, located in Rome. Seeweb has two data centers, respectively located in Milan and Frosinone, so Seeweb reserved for us two servers, one in each data center and we targeted the measurement to both.

5.4 Analysis

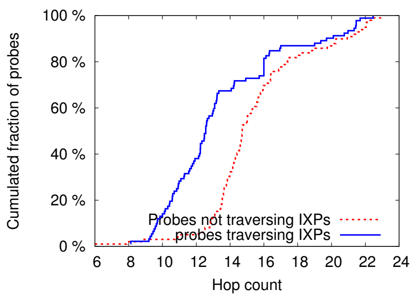
Experiment CIS

The actual number of probes that were used for EXPERIMENT CIS is 91 for the measurements towards targets in dataset CRITICAL, and 104 for the measurements towards targets in dataset VISITED.

Figure 5.2 contains graphs that give a high level overview of how performance, as measured by Atlas probes, depend on the presence of IXPs in the traceroute paths. The performance indicators used to generate the graphs are the rtt and the hop count, respectively. Each graph contains two plots, one for all the traceroutes that use paths traversing one of MIX and NaMeX (blue) and the other for traceroutes that do not see any of the two IXPs in the traceroute paths (red). Each plot is a CDF showing the distribution of measured values for the selected performance indicator. Fig. 5.2a is for dataset CRITICAL, while Fig. 5.2b is for dataset VISITED. For brevity we omit the figure on rtt of VISITED and the figure on hop count for CRITICAL which exhibit exactly the



(a) Rtt

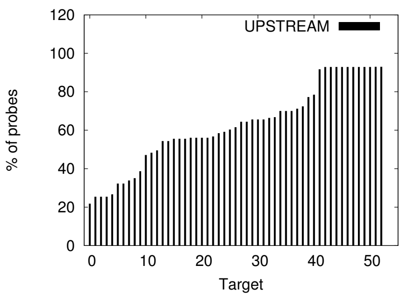


(b) Hop count

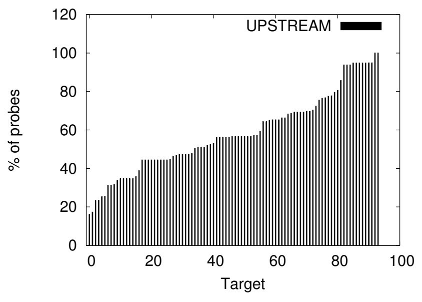
Figure 5.2

same pattern. The plots show that, in general, probes traversing IXPs have better indicators. For example, referring to Fig. 5.2a, about 70% of probes that choose IXPs have average rtt of 30ms or less, while only 20% of those that do not traverse IXPs have the same performance. However, such results must be considered very carefully, because of the following three reasons: 1. The two CDFs of Fig. 5.2a (the same holds for Fig. 5.2b) refer to disjoint sets of probes, since each probe either passes through an IXP or not. In principle, it is possible that a probe currently passing through an IXP could have a better performance when forced, in some way, to pass through an upstream (or vice-versa). 2. Rtts and hop counts refer to the last hop that replied to our measurements, which quite often is not the actual target (40 cases out of 53 for dataset CRITICAL, and 59 cases out of 94 for dataset VISITED). 3. A router in an IXP might answer to a traceroute using an interface that is not on the peering lan, leading to a misclassification of the corresponding traceroute. Also, the data on hop count could be affected by the presence of tunnels (e.g. transparent MPLS tunnels [CDP13]). On the other hand this might happen both to packets that traverse IXPs and to packets that do not. However, these graphs help us draw a first positive impression on the impact of IXPs. Observe that EXPERIMENT SBA does not suffer of the problems pointed out in items 1–3.

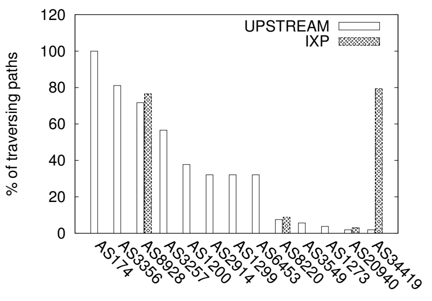
Next we study how Atlas probes distributed their traffic between paths including or excluding IXPs during EXPERIMENT CIS. Figure 5.3b shows, for each target, the percentage of probes that reached it through an upstream. These graphs show that most targets are reached preferably through upstreams. In particular, in Fig 5.3a, 40 targets over 53 were reached through an upstream



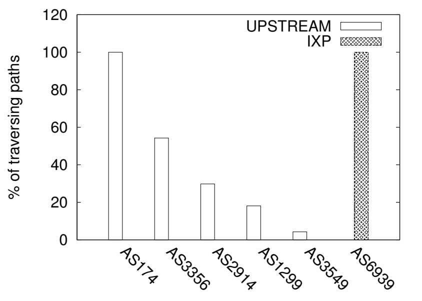
(a) CRITICAL



(b) % of probes that reach a target with an upstream in EXPERIMENT CIS.



(a) Non-Italian traversed ASes for CRITICAL



(b) Non-European traversed ASes for VISITED

Figure 5.4: % of paths traversing foreign ASes when an IXP is used or not.

by more than 50% of probes, and, in Fig 5.3b, 60 targets over 94 were reached through an upstream by more than 50% of probes.

Finally, the data collected in EXPERIMENT CIS can be analyzed in terms of security issues associated with the traffic targeted at crucial Web services. Indeed, recent cases of network traffic espionage [dat] solicited governments to check if the traffic generated by their citizens and targeted to critical Web services remains or not inside the countries [rus]. It is perfectly clear that this type of locality does not give security guarantees on such a traffic, but it is equally clear that, nowadays, governments are very sensitive to this issue. Therefore, we measure how frequently using an IXP to reach an Italian target from an Italian source permits to avoid the transit through a foreign upstream.

That is, we quantitatively check whether MIX and NaMeX are effective in keeping the local traffic local. In order to determine the country of an AS X, we query the RIPEStat [rip] service asking for the prefixes announced by X, along with the country of each of these prefixes. If the returned prefixes are all associated to the same country, then we assume that it is the country of X. Otherwise, to determine its country, we manually retrieve metadata on X looking, e.g., at the website of the corresponding provider. Fig. 5.4 illustrates our results in terms of locality. The statistics are computed as follows. First, the ASes traversed by paths in Experiment CIS are labeled as *non-Italian* and/or *non-European* if the nation detected by the aforementioned heuristic is non-Italian and/or non-European. Also, the source-target pairs of the experiment are split into two groups, based on the presence of IXPs in their paths. Finally, each detected non-Italian and non-European AS is associated with the source-target pairs that traverse it. Fig. 5.4 shows the results of the computation. As an example, abscissa 3356 of Fig. 5.4(a) shows that 80% of the source-target pairs not passing through an IXP pass through AS3356 (that is not Italian) while no one of the source-target pairs passing through an IXP traverse it. In most cases, a foreign AS is traversed only when an upstream is used, or in great prevalence when an upstream is used, which confirms that IXPs are effective in preserving the locality of traffic. A few exceptions are ASes 8928 (INTEROUTE), 8220 (COLT), 34419 (VODAFONE), 20940 (AKAMAI), and 6939 (HURRICANE). We have detected, by checking the traceroute paths, that INTEROUTE, COLT and HURRICANE have peerings at IXPs, so that is the reason for their high values in the IXP cases of Fig. 5.4. VODAFONE, instead, receives traffic from the Italian Vodafone AS, so it is a case of traversed foreign AS that cannot be avoided by using IXPs. AKAMAI is an interesting case, since in our experiments it receives traffic from BT Italia, which contains one of the targets of Experiment CIS, and then sends it back to the same AS. Our guess is that AKAMAI, as a content provider, has some network infrastructure inside the data centers of BT Italia, and it possibly hosts some targets of our measurements.

Experiment SBA

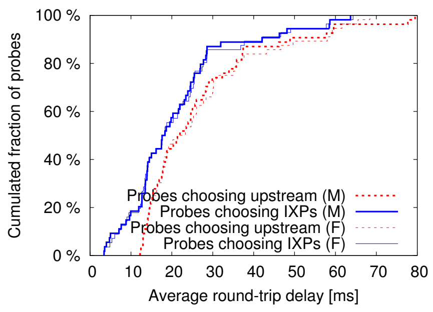
For Experiment SBA we started by selecting the subset of probes, called multi-choice, that were able to reach the target during both the *"UPSTREAM"* interval and during at least one of the *"MIX"* and the *"NAMEX"* intervals (see Section 5.3). These probes allow to measure the differences between the connectivity offered by IXPs and that offered by upstream providers. The actual number of multi-choice probes amounts to 83 for the Seeweb Milan target, 81 for the Seeweb Frosinone target, 74 for MC-link, and 80 for Unidata.

Fig. 5.5 shows the distribution of performance indicators on all multi-choice probes. Fig. 5.5a shows the rountrip delays measured during the experiment with Seeweb, Fig. 5.5c shows the hop count measured during the experiment with MC-link, and Fig. 5.5e shows the jitter measured during the experiment with Unidata. Because of space limitations we omit figures showing other combinations of performance indicators and ISPs, since they show analogous trends.

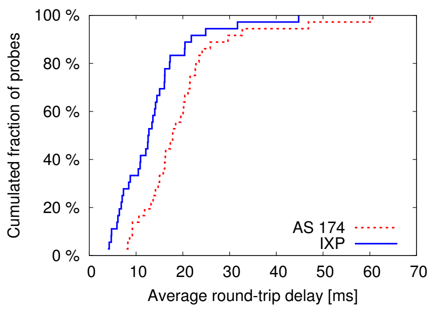
As an example, Fig. 5.5a contains 4 plots that show the distribution of the average rtt for the two Seeweb targets, in both cases when the target was reached through an IXP or through an upstream. Consider that in this and in all the following figures the plots involving paths through IXPs show the "best" option, i.e. for each performance indicator and for each probe that reached the target during both the *"MIX"* and *"NAMEX"* intervals, it is shown the smallest value between the two options. Observe that about 50% of multi-choice probes reach both targets with an average rtt of at most 15ms when using IXP connectivity, compared to only about 30% of them in case of upstream connectivity. The measurements towards the Milan target and the Frosinone target exhibit comparable performance.

Figs. 5.5a, 5.5c, and 5.5e highlight that paths traversing IXPs always give performance that are better or at least equal to those traversing upstreams, for each provider among Seeweb, MC-link, and Unidata, and for each considered performance indicator among average rtt, average hop count, jitter, and packet loss. The latter is the only metric for which IXPs and upstreams exhibit about the same performance, with negligible differences, and its graphs are omitted for brevity.

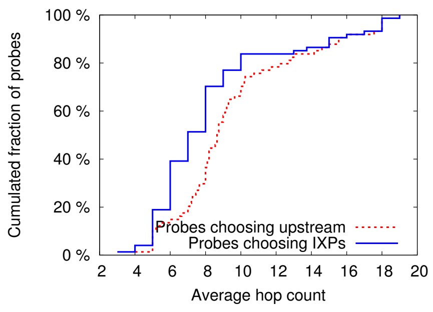
During the experiments we noticed that paths through the upstreams exhibited quite different values of performance indicators depending on the specific traversed upstream. So, looking at the traceroutes gathered during the *"UPSTREAM"* interval, we found the upstreams of each of our providers and deepened the analysis for all of them. Here is the list of upstreams that we found: 1. MC-link (AS3257, AS12874, AS174, AS3356, AS35612, AS57329); 2. Seeweb (AS3257, AS174, AS3549, AS3356); and 3. Unidata (AS3257, AS12874, AS16004, AS24796, AS20836). Figs. 5.5b, 5.5d, and 5.5f show the main outcome of the analysis. As an example, Fig. 5.5b contains the same data of Fig. 5.5a restricted to the Milan target and to the probes that reached the target through AS174 (COGENT). The IXP plot is restricted to those probes too. Observe that the plot has exactly the same trend of Fig. 5.5a. It follows that the paths traversing



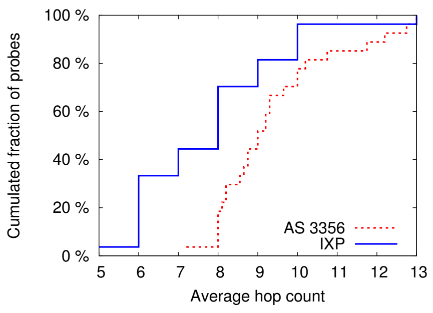
(a) Average round-trip delay, Seeweb. "M" is Milan, "F" is Frosinone.



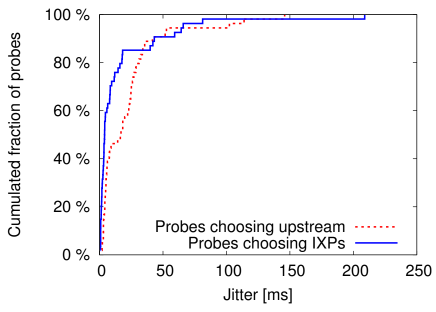
(b) Average round-trip delay, Seeweb Milan, for upstream AS174



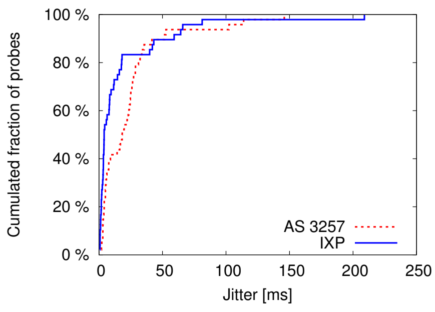
(c) Average hop count, MC-link



(d) Average hop count, MC-link, for upstream AS3356

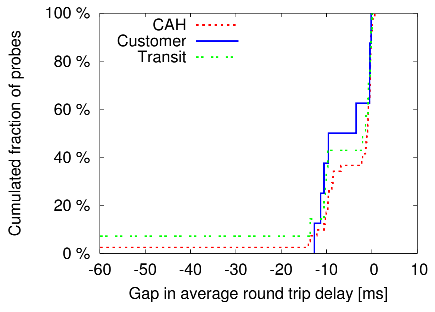


(e) Jitter, Unidata

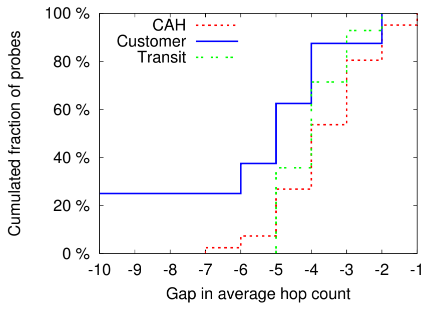


(f) Jitter, Unidata, for upstream AS3257

Figure 5.5: Comparison of performance indicators between upstream and IXP connectivity, as measured in EXPERIMENT SBA. For each of our three partner ISPs, one performance indicator is shown. The same data is also plotted restricted to a specific upstream.



(a) Average rtt



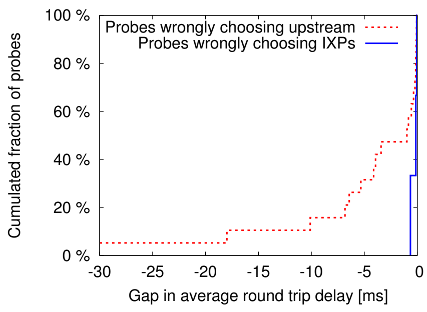
(b) Average hop count

Figure 5.6: Difference in performance based on the AS classes of the probes.

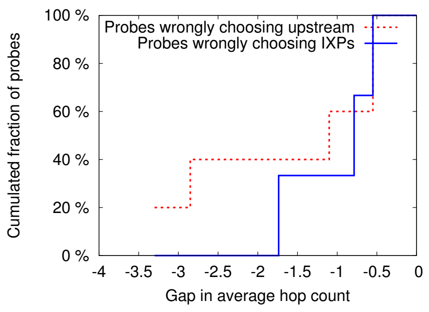
AS147 are the principal factor that makes the average upstream performance lower than the IXP performance. Similar arguments apply to Figs. 5.5d and 5.5f. In the first one, AS3356 (LEVEL3) is shown as the upstream with the largest difference from the IXP alternative among all the upstreams of MC-link. In the second one, AS3257 (TINET) exhibits a behavior similar to the plot of Fig. 5.5e.

We argue that the performance obtained from upstream connectivity heavily depend on the choice of a specific upstream.

As a next step, we consider the classification of ASes in [DD]. Our goal is to identify different trends in the improvement of performance indicators based on the type of AS that hosts a probe. ASes are assigned to 3 classes: 1. *Customers* represent various organizations, universities and companies at the network edge that are mostly users of the network. 2. *Transit* are ISPs that provide Internet access and transit services. Transit aim to maximize their customer base in their geographical area and to reduce their upstream transit costs through selective peering with ISPs. 3. *Content/Access/Hosting Providers* (CAH) are ISPs that offer Internet access and/or server hosting. Their access customers can be residential users or enterprises with no AS numbers, while their server hosting customers are content/service providers with no AS numbers. Figure 5.6 presents two graphs, respectively based on average rtt and average hop count for the Seeweb Milan target during Experiment SBA. Each graph contains 3 plots, each corresponding to one AS class. A plot is a CDF that shows the distribution of performance gaps derived from the comparison of performance indicators between paths traversing IXPs and paths avoiding them, as seen by



(a) Average rtt for Seeweb



(b) Average hop count for MC-link

Figure 5.7: Comparison of routing choices made in the *“ALL”* phase of EXPERIMENT SBA that lead lower performance than alternative routing choices. The measurement are performed towards (a) Seeweb target in Milan and (b) MC-link target.

each probe. Graphs show that Customer ASes hosting probes in Italy are the ones that benefit the most from the availability of IXPs. For example, about 50% of probes in Customer ASes see an improvement in the average rtt of at least 10ms when traversing IXPs, as opposed to about 20% and about 40% for probes in CAH and Transit ASes, respectively. Similarly, about 62% of probes in the first class has path lengths reduced by at least 5 hops, while the other two classes only reach about 25% and about 38%.

Next, we leverage data collected during the execution of EXPERIMENT SBA to understand how many probes picked the “wrong” path to the destination when all options were available (i.e. during the *“ALL”* interval), based on the comparative analysis of performance indicators. By “wrong”, or “incorrect”, we actually intend that a path is suboptimal from the point of view of one of the performance indicators we have considered, while a different path would have had better performance. Obviously, an ISP may have motivations that go beyond our metrics for choosing a path instead of another. Fig. 5.7 presents two graphs based on average rtt and average hop count, measured respectively towards the Seeweb Milan target and the MC-link target. Each graph contains two plots, one for probes that incorrectly preferred upstreams over IXPs (in red), and one for probes that incorrectly preferred IXPs over upstreams (in blue). Each plot is a CDF that on the x axis has performance gap values for a given metric, computed as $metric_a - metric_c$ where c and a are respectively the current path traversed by a probe and the alternative path that was available. One between c and a traverses an IXP, while the other does not. Only negative gaps are shown, so the graphs represents those probes that made a "wrong" choice between traversing an IXP or an upstream to reach the target. For a given performance gap g, on the y axis there is the percentage of probes with a gap less or equal to g over all probes with a "wrong" choice. More informally, the CDFs gives a quantitative overview of "wrong" choices based on how much the performance indicators would really improve by switching routing options (i.e. traversing IXPs for the red curve, and avoiding them for the blue curve). The analysis of both graphs reveals that a relevant percentage of "wrong" upstream transits actually lead to non-negligible performance degradations, whereas "wrong" transits through IXPs generally correspond to much less negative effects. For example, referring to Fig. 5.7a, about 30% of probes that choose upstreams would reduce their average rtt by at least 5ms by switching to an IXP, while no single switch in the opposite direction would give the same tangible effect.

5.5 Conclusions and Future Work

We investigated the role played by IXPs in the Italian Internet ecosystem. Our experiments highlight that peerings exploiting IXPs have a positive effect on key performance indicators such as latency, hop count, packet loss, and jitter. Also, they reduce the number of foreign ISPs traversed by the traffic between Italian citizens and critical Internet services like Banks and Public Administrations. We conclude that if an ISP motivates a de-peering through an IXP in terms of performance improvements it makes a statement that is at least questionable.

It would be interesting to extend our experiments considering more performance indicators. Unfortunately, the measurement network used in our experiments, although technically ready for measuring more parameters (e.g. for bandwidth measurement), adopts a conservative approach to limit the impact of experiments on users that host probes. Further, we plan to cooperate with ISPs and IXPs in other countries to reproduce the experiments, compare the results, and generalize the methodology. Another direction for extending our experiments would be to consider full QoE issues. However, this would imply application level measurements which are even harder to perform on common measurement networks.

Part III

Routing Policies at IXPs and their Disclosure

Chapter 6

PrIXP

Internet eXchange Points (IXPs) serve as landmarks where many network service providers meet to obtain reciprocal connectivity. Some of them, especially the largest, offer route servers as a convenient technology to simplify the setup of a high number of bi-lateral peerings. Due to their potential to support a quick and easy interconnection among the networks of multiple providers, IXPs are becoming increasingly popular and widespread, and route servers are exploited increasingly often. However, in an ever-growing level of market competition, service providers are pushed to develop concerns about many aspects that are strategic for their business, ranging from commercial agreements with other members of an IXP to the policies that are adopted in exchanging routing information with them.

Although these aspects are notoriously sensitive for network service providers, current IXP architectures offer no guarantees to enforce the privacy of such business-critical information. We re-design a traditional route server and propose an approach to enforce the privacy of peering relationships and routing policies that it manages. Our proposed architecture ensures that nobody, not even a third party, can access such information unless it is the legitimate owner (i.e., the IXP member that set up the policy), yet allowing the route server to apply the requested policies and each IXP member to verify that such policies have been correctly deployed. We implemented the route server and tested our solutions in a simulated environment, tracking and analyzing the number of exchanged control plane messages.

6.1 Introduction

Organizations that offer Internet-based services (Internet Service Providers, Content Delivery Networks, etc.) join the Internet eXchange Points (IXPs) in order to quickly and easily reach a number of other parties networks, and gain the level of connectivity they need [DDdS15]. However, such organizations are usually concerned with business-critical aspects for which IXPs do not currently provide any technical solutions. These aspects include, among the others: (i) privacy of the peering relationships, which are an evidence of the existence of commercial agreements; (ii) privacy of routing policies, which determine what kind of traffic can flow between peering partners; (iii) security of the network infrastructure (links, devices), that might be traversed by sensitive traffic.

Currently, IXPs offer a very useful service, called Route Server (RS). A RS allows each member connected to an IXP to easily exchange traffic with other members by establishing a peering session with the RS, instead of having one peering with each other member he wants to be connected to. Peering sessions are handled by the Border Gateway Protocol (BGP), the standard interdomain routing protocol. Surely, this functionality significantly reduces the effort needed by the IXP members to connect to the Internet.

Ensuring the privacy and correctness of Internet peering policies is a desired requirement for many Internet entities as this information reflects business relationships, such as commercial agreements, which must comply with stringent Service Level Agreements (SLAs). Very often, RS functionalities are mainly leveraged by small providers and Content Delivery Networks (e.g. [AMS16, LIN16, FRA16, MIX16]) since these players have strong interests in connecting to many IXP members by just setting up a single BGP peering with RS. On the other hand, big Internet players, with very few exceptions (e.g. Google [AMS16]), tend to not have BGP peerings with a RS. We argue that this trend is the result of exposing an IXP member to a potential violation of privacy in terms of BGP policies when peering with a RS. In fact, each peering policy would be stored within appropriate data structures at the RS and, potentially, these can be altered by a malicious software. As a result, most Tier-1 ISPs require their peers to sign Non Disclosure Agreements (NDAs) when peering with them [Pee12].

In this thesis, we present PrIXP, a RS system that improves both the privacy guarantees of confidential peering information and the security of the RS. Our key idea is to prevent the RS from locally storing any BGP policies. Instead, the RS queries routing policies in on-demand manner by means of a second communication channel that we instantiate between the RS and each IXP member. Namely, each time the RS performs a routing operation, it leverages this extra channel to retrieve from each member its routing policies such as the set of member neighbors that should receive certain routing information and the local preference over routes of each member. To guarantee the correct execution of the BGP routing protocol at the RS, we leverage Intel proprietary Software Guard eXtensions (SGX) technology [MAB+13], which allows external entities to attest that a remote software has not been tampered by a malicious attacker. Finally, to enable incremental deployment, we discuss a BGP compatible mechanism that can be used in place of the extra channel, thus requiring no hardware modifications at the IXP member side.

The rest of the chapter is organized as follows. In Sec. 6.2, we provide an overview of a common real-world architecture of a route server deployed inside IXPs. In Sec. 6.3, we describe our system in detail, presenting a complete example of an interaction between the route server and the IXP members connected to it. In Sec. 6.4, we address the security issues associated with peering with a traditional RS by describing our solution for allowing any IXP member to check the integrity of the RS. In Sec. 6.5, we evaluate our system by using a real-world trace of BGP updates from one of the largest IXP worldwide. In Sec. 6.6, we review the most relevant contributions related to Internet routing privacy and security. Finally, we draw conclusions and future work in Sec. 6.7.

6.2 Background: Route Server Architecture

In this section, we describe the typical architecture of a RS service offered to the members of an IXP (e.g. [RSF+14b]). To the best of our knowledge, many large IXPs such as DE-CIX, AMS-IX, and NYIIX are currently using RS implementations based on that architecture.

Before entering into the details, we introduce terminology and definitions that will be largely used throughout the chapter. We define the *RS-software* as the piece of software implementing the RS functionality and the *RS-machine* as the physical hardware that runs the RS-software. We also define the *peering LAN* as the local network managed by the IXP where its members connect and establish BGP peerings among themselves and with the RS-software.

In a standard scenario, each member of an IXP establishes a number of *bi-lateral* BGP peerings with all the members with whom it has agreed to exchange network traffic for certain IP prefix destinations. Such bi-lateral peerings usually correspond to commercial agreements between the involved parties. In contrast to this approach, many IXPs provide RS services as a convenient alternative for their members to simplify the setup of peerings while

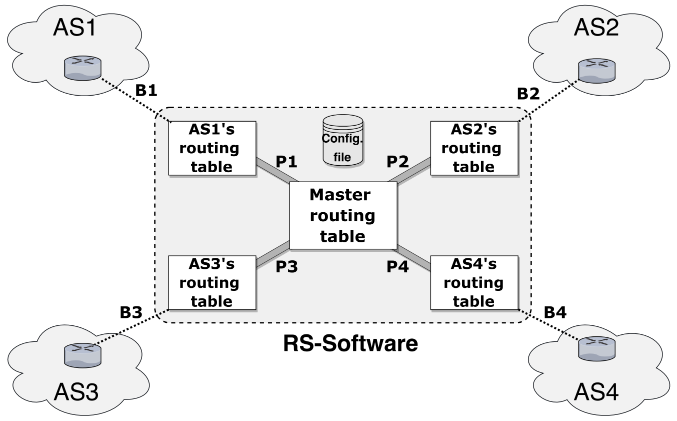


Figure 6.1: Reference architecture of route server.

optimizing the operation of the BGP control plane. Indeed, RSes reduce the configuration effort required by network operators to join and manage many *bi-lateral* BGP sessions at large IXPs, since having a single BGP peering with the RS is enough to be connected with the other members.

We now describe the set of operations that an IXP member should perform in order to make use of RS services. First, the IXP member establishes a single BGP peering towards the RS-machine with the RS-software, which is responsible for forwarding any BGP announcements according to the routing policies configured by the members. The above scenario is denoted as a *multi-lateral* peering, where the RS acts as the center of a star topology where the members are called *clients*.

The architecture of a RS-software is shown in Fig. 6.1. In this figure, AS1, AS2, AS3, and AS4 are members of the IXP, each of them connected to the IXP using a BGP-speaking router, where the dashed lines labeled B1, B2, B3, and B4 represent BGP peering sessions. Each of these routers independently keeps a routing table that stores the IP prefixes coming from its own network, as well as those received from its multi-lateral peers through the RS. The rounded dashed box labeled "RS-software" represents an instance of the RS routing software, where the contents of the box depict the most important data structures that are maintained by the software and the channels used to move data among these structures. We now describe each basic component represent in the figure.

Tables. The basic data structures maintained by a RS are *BGP tables.* A BGP table contains a set of routing entries, each of them consisting of an IP prefix and the BGP message announcing that prefix. Multiple entries for the same prefix may exist, though only one of them is marked as the best one that should be propagated to the other members. For each member, a RS-software keeps a distinct table that stores all the routes that are announced towards that client from other clients. In order to support the exchange of routing information among these tables, the RS also maintains a single *master* table, which usually aggregates all the routes received from all the client-specific tables.

Protocols. The RS software leverages different communication channels for transferring information among tables, called *protocols*. BGP routes are exchanged between a client and one of the member-specific tables inside the RS-software through a BGP session (lines B1, B2, B3, B4 in Fig. 6.1). The routes learned from these sessions can then be propagated between the different tables using a RS-specific protocol, which corresponds to the links among the BGP tables (thick lines P1, P2, P3, and P4 in Fig. 6.1).

Filters. In order to support arbitrary routing policies, it is also possible to define filters. A *filter* is typically a fragment of code, possibly written in a specific programming language, that supports evaluation of arithmetic expressions, conditional statements, etc. Filters are applied on each BGP announcement ever time they are exchanged through BGP sessions or RS-specific protocols. A filtering operation can have three possible outcomes: (1) forwarding the announcement, (2) modifying some attributes in the announcement before forwarding it, and (3) dropping the announcement. Filters can be statically configured within the RS software by the IXP operators. This practice is commonly adopted for limiting the risk of IP-prefix hijacking. The common way to perform filtering is encoding the set of members to whom a routing announcement must be sent via specific BGP attributes that are attached to the announcement itself, i.e., via BGP communities, where each BGP community simply consists of a pair (x, y) of values. We define a *whitelist export policy* as the set of members $(\mathsf{AS\_1}, \ldots, \mathsf{AS\_N})$ encoding all members that are allowed to receive a BGP announcement. A whitelist is expressed by a sequence of community values starting with a special community $(\mathsf{0} : \mathsf{IXP\_id})$, followed by a sequence of other community values $(\mathsf{IXP\_id}, \mathsf{AS\_i})$, for each $i = 1, \ldots, N$ representing all members to which forward the announcement. In the same way, we define a *blacklist export policy* as a sequence of community values encoding the set of ASes $(\mathsf{AS\_1}, \ldots, \mathsf{AS\_N})$ that should not receive a BGP announcement.

A blacklist always starts with a special community (IXP_id : IXP_id) and it is followed by a pair $(0, \mathsf{AS\_i})$, for each member that is denied to receive the announcement.

Best Route Selection and Propagation. Unless filters enforce restrictions, the adoption of a specific internal protocol, as explained before, causes all BGP routes to be copied between the tables it links, retaining all their attributes and including non-best routes. Each best route for a member is computed using the information gathered in its specific member routing table. This strategy allows IXP operators to overcome the well-know problem known as *path hiding*, which arises whenever filters are applied [JHRB16, RSF+14b]. This is a well-known problem that might affect members if the RS-software acts as a standard BGP router, where a single master route table is used to collect all the route announcements and to compute a unique best route for all the customers. For example, consider the case in which there are four members (AS1, AS2, AS3, and AS4) connected to a RS-machine through a multi-lateral peering. An IP prefix π is announced by AS1 and AS2 and the latter one defines a restricted policy that prevents AS3 to receive the announcement containing π. Also, suppose that the RS-software runs the best route process only considering the routes contained in the master table and that computation selects the route passing through AS2 as the best one. In this case, this route is only advertised to AS4, leaving AS3 without any route towards π, even though a route passing through AS1 exists. Breaking down the master table into per-member tables makes possible to run independent best route computation on each member table, preventing the above situation to happen. Although the BGP configuration language allows routes to be ranked based on the *local preferences* of each member, today's RSes do not support this mechanism and the best route is computed based on a global ranking, as defined in [RLH06].

Configuration Example

We now show an example of typical route server configuration, based on best practices that are documented in [GZ13] and that were confirmed during a discussion with the authors of [RSF+14b]. In this description we omit most technical aspects and focus solely on the filters. Consider the case, depicted in Figure 6.1, in which 4 clients connect to the RS, and assume that AS1 is a provider network and AS2, AS3, and AS4 are its customers. Also assume that ASx owns IP prefix `100.`x`.0.0/16`, $x = 2, 3, 4$, and that each prefix aggregates any other address spaces owned by lower-level customers of AS2, AS3, AS4.

First of all, the configuration of AS1's BGP routers will likely apply an inbound filter to discard all prefixes except those announced by the customers, namely `100.2.0.0/16`, `100.3.0.0/16`, and `100.4.0.0/16`. Moreover, it may also apply outbound filters to restrict announcements sent to customers to the sole default route `0.0.0.0/0`.

Configurations on the RS side have the same structure for all RS clients: essentially, they filter out BGP announcements that carry foreign prefixes and selectively propagate each announcement to other clients depending on the BGP communities it carries. In particular, the RS applies on each BGP peering B1, B2, B3, B4 a generic import filter that discards "martian" routes (that is, private, multicast, and reserved address spaces, as well as IP subnets whose netmask length is out of the standard 8-24 range). The export filter simply cleans outgoing BGP announcements by stripping any BGP communities that are used inside the RS to tag routes that must be exported to other clients. On the other hand, filters applied on `pipe`s are more restrictive. Import filters usually apply some checks to verify the consistency of each BGP announcement (e.g., whether the announcing router concides with the next hop), and accept from each client only those announcements that carry expected IP prefixes originated by expected AS numbers (the origin is determined from the last AS in the AS path): only such announcements will be inserted in the master table. Export filters first of all handle well-known communities, like `NO_EXPORT` or `NO_ADVERTISE` (see [CTL96]). In addition, they restrict exporting of entries from the master table to the table of a client x only to those routes that are tagged with x's AS number. Exportable routes may be additionally tagged with a custom community value that is used internally by the RS to prevent from re-importing them in the master table. All the above described filters are summarized in Table 6.1

To complete the example, we now briefly summarize the processing steps that are applied to a BGP announcement for `100.2.0.0/16` originated by AS2, assuming that this announcement is tagged with a community value `2:1`. First of all, AS2 applies its own export policies which, being a customer, filter outgoing announcements to prevent traffic in transit: since `100.2.0.0/16` is originated by AS2, it passes this filtering step. After that, the announcement reaches the RS, which applies the import filter of the `bgp` protocol, accepting the announcement only if it does not carry martian routes. Assuming that this is the case, an entry is then added to AS2's routing table inside the RS. Then, the RS applies an import filter for the `pipe` that connects this table to the master table: at this step, the announcement is checked for consistency and accepted only if it carries known IP prefixes (in this case `100.2.0.0/16`)

Table 6.1: Overview of the filters that are applied by the route server in commonly adopted configurations.

Protocol	**Import filters**	**Export filters**
`bgp`	No martians	Strip BGP communities used to tag exported routes
`pipe`	Check consistency + Accept well-known prefixes & origin ASes	Export routes depending on communities

originated by legitimate ASes (in this case AS2). Since both conditions apply in this case, an entry for this announcement is added to the master table. At this point, the RS considers every other client (AS1, AS3, AS4) and looks at the community value `2:1` that is carried by the announcement. This value does not match the AS numbers of AS3 and AS4, therefore the announcement is not propagated to their routing tables: this is correct, considering that AS2 is supposed to be reached by these two customers via its provider AS1. Instead, the community indicates AS1 as correct recipient of the announcement, therefore the latter is exported to AS1's routing table inside the RS. Afterwards, the RS select a best route for `100.2.0.0/16` among those in AS1's routing table (in this example there is only one available), strips from this entry any community values applied for internal use by the RS itself, and sends a BGP announcement to AS1. Finally, the latter applies its own import policies, which are supposed to accept `100.2.0.0/16` because it is a legitimate customer IP prefix.

Although a RS-software has a complex architecture, it does not provide any privacy mechanisms allowing each member to protect its policies.

6.3 Enforcing Privacy of Routing Policies

In this section, we describe how PrIXP improves the level of privacy for the members' routing policies within a RS-machine. In our system, each member can easily leverage the RS's functionalities (e.g. BGP routes dispatch based on export policies and local-preference tools) while minimizing the risk of leaking any confidential information. Our system does not propose an entirely new cryptographic protocol, but leverages well-established techniques (e.g., TLS, SGX) to secure channels and performing remote attestation. Those techniques can be replaced by any equivalent technology.

We observe that current RSes designs (described in Sec. 6.2) require IXP members to disclose their export policies. In fact, any peering relationships among the IXP members can be reconstructed by simply looking at the client-specific routing tables stored in memory or on disk. In fact, the table of a member ASx contains all the routes received by other ASes, thus revealing what are the export policies of each member towards ASx. Moreover, any enhanced RS service that allows the IXP member to rank their available routes based on their specific local preferences would raise additional privacy concerns. In fact, such services would require each member to disclose their ranking policies to the IXP.

We now describe the security assumptions and the threat model on which our system is based. First, we assume that the attacker does not have visibility of data traffic. Namely, an attacker cannot eavesdrop the packets sent through the peering LAN of the IXP in order to infer the peering relationship among the members. Second, we assume that the attacker operates on the RS-machine during a short time interval in which he tries to take a snapshot of as much information as possible from the content stored in the route server system, possibly tampering the RS-software itself.

Our system is based on the following principles. The only information that is stored within the PrIXP RS is the one needed to maintain the established BGP sessions with the IXP members and, for each announced prefix, the set of members that have a route towards it. The routing policies of the IXP members are never permanently stored by the RS-software inside the RS-machine so as to minimize the risk of privacy breaching. In contrast, these policies are asked to the members in response to the reception of a BGP announcement that has to be dispatched. We make use of an extra communication channel for retrieving this information, which can be set up using standard techniques (e.g. SSL/TLS). We observe that, in order to minimize the modification required at the member side, it would be worth to investigate how to implement this channel by tweaking the BGP protocols. The idea is to leverage *Conditional Route Advertisement*, a BGP route dissemination feature that allows to conditionally announce one or more prefixes upon the reception of some specific routes. Such a feature is currently supported by important vendors, as shown in ([Sup16, Tec16]).

The extra communication channel is used by the PrIXP RS to query each member for the following information: (i) the export policies of a routing announcement (e.g. the set of members to whom a route should be propagated) and (ii) the local preferences over routes of each BGP member that is entitled to

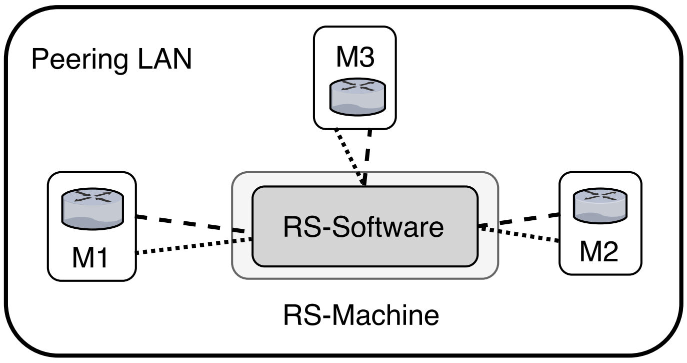


Figure 6.2: The architecture of an IXP infrastructure.

receive a BGP message. We now provide a detailed description of the operations performed by the PrIXP.

A Complete Example. A simplified scenario that we use to illustrate how our system works is depicted in Fig. 6.2. The RS-machine is placed in the middle of the drawing, while the three members (M1, M2, and M3) are connected to the IXP physical infrastructure. For our convenience, we assume that M1, M2, and M3 are also the identifier of the three members, respectively. The rounded rectangle containing the whole drawing represents the peering LAN and we assume that the peering LAN consists of a single switch. Each dashed line represents a BGP session, whereas dotted lines represent the extra communication channel used by PrIXP to query each member. To make use of the RS's functionalities, each member establishes a BGP session with the RS-software. Each member can still establish bi-lateral BGP sessions with the other members as in traditional RSes.

Once all the BGP connections are established, members can use them to exchange routing information among each other. For instance, suppose that M1 and M2 send an announcement towards an IP prefix π to the RS-software, which is responsible for dispatching it according to the members' routing policies. Upon receiving this message, PrIXP asks M1 for the export-policy. Member M1 replies to this request by communicating a set of BGP communities encoding a policy that allows the RS-software to advertise the announcement to M3. After delivering the message, the RS-software stores in its memory that it received a route for π from M1, but it deletes any other additional information (e.g. the export policies and the BGP attributes contained in the announcement).

Now, suppose that also M2 sends an announcement towards π to the RS-software. When the RS-software receives that message, it checks whether there exist other routes announced towards π. Then, it asks each member that announced a route towards π (M1 and M2) for the export policies of their announcement using the extra communication channel. Member M1 communicates again that its announcement must be announced to M3, while M2 instructs the RS-software to propagate its message to both M1 and M3. Upon receiving the export policies, the RS-software knows which routes can be exported to each member. In order to select the best one, the RS-software asks each member with at least two available routes for the local-preference of each route announcement. In our case, the RS-software asks M3 to provide the ranking over the routes announced by M1 and M2. Once M3 provides its local preference, the best route is sent to M3. As for M2, the only route that is available to be exported to it is the one through M3, which is then propagated accordingly. Note that, this last step is performed over the BGP peering. After that computation, the RS-software discards all the information used to propagate the routes, except for the mapping between routes and members who announced them. This operations allows us to minimize the risk of leaking routing policies whenever an attacker can observe the state of the RS-software for a short interval of time. Note that having a single BGP decision process for each member makes our RS-software not affected by the *path hiding* problem.

6.4 Discussion on Security Issues

In this section, we describe some security considerations, addressing the problem of how a member can verify that the RS-software has not been tampered or replaced by another malicious software. To minimize the risk of leaking confidential information, we assumed in Sec. 6.3 that each member is connected to a trusted execution of the PrIXP RS-software. Under our threat model, we assume that the attacker may quickly replace the RS-software to collect confidential information that can be read by the attacker next in the future. For this reason, we also define a RS security architecture, depicted in Fig. 6.3, which is based on recent advancements in remote attestation protocols. A trusted authority issues a certified version of the RS-software that each member can verify on its local machine, implemented according to the description of the PrIXP system in the previous system, represented by a triangle in the picture. Unfortunately, to allow all the IXP members to be able to check that at any time the RS-software is behaving as expected, having just a certified version of the

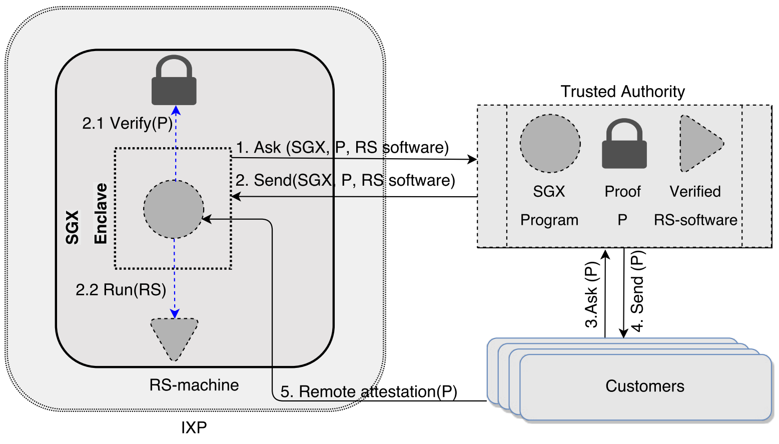


Figure 6.3: Architecture for checking the integrity of the RS software.

RS-software is not enough, because a member does not have any tools to attest at any time that exactly the certified version of the software is running. Indeed, an attacker can suitably replace that certified version of the RS-software. To solve this problem, we rely on the recent advancements on *Remote Attestation*, which allows changes to the RS-software to be detected by authorized parties. Intel Security Guard eXtension (SGX) [MAB$^+$13] is an example of a technology that allows programmers to implement remote attestation procedures.

Each SGX program needs a proof to be executed on a SGX-enabled machine. In our architecture, the trusted authority provides to the RS-machine an SGX program and a proof P, respectively depicted by the circle and the lock in Fig. 6.3.

The integrity check works as follows. First, the RS-machine, which has an SGX processor, sends to trusted authority a request to obtain the RS-software, the SGX program and the proof P. This is represented as the step 1 in the figure. Upon receiving this request, the trusted authority sends back to the RS-machine the RS-software, the SGX program and the proof P. This is step 2 in the figure. At this point, the RS-machine owns all the necessary pieces to correctly run the verified RS-software. This task is accomplished by the SGX program that can run if and only if the proof P has been verified (step 2.1). In order to guarantee that the RS-machine will run the correct version of the RS-software, the SGX-program will check that the hash of the RS-software to be executed corresponds to the one that is hard coded in the SGX-program retrieved from the central authority (step 2.2). At this point, whenever a member wants to check the integrity of the RS-software, it asks the trusted authority for the proof P, which is denoted as step 3 in the figure. Upon receiving the proof P, a member performs a remote attestation against the SGX program running at the RS-machine by using the proof P (steps 4 and 5 in the picture).

Since the proof P used by a member for the remote attestation is the same used by the SGX program at the RS-machine to run the RS-software, the operation succeeds. If an attacker aims at replacing the RS-software, he must also replace the SGX program, otherwise the hash-check would not allow him to run its own RS-software. This implies that a new proof P must be provided to the SGX-machine in order to run the malicious SGX-program. At this point, if the attackers succeed in running its malicious SGX-program and RS-software, the remote attestation performed by any member using the proof P would fail, as the SGX-machine would alert the user the SGX program is not the legitimate one.

In this work, we do not propose any new cryptographic protocol, but we leverage well-established techniques (e.g. TLS for the extra communication channel and SGX for remote attestation). Those techniques can be replaced by any equivalent technologies without altering the PrIXP functionalities.

6.5 Experiments

To assess the effectiveness of PrIXP, we simulated our system (available at [Uni16]) using a trace of BGP updates from one of the largest IXP worldwide with several hundreds of members whose name cannot be disclosed in this thesis. Our simulation aims at estimating how much overhead our methodology introduces in terms of BGP control plane messages. We do not measure CPU overhead or memory utilization, since we do not expect both of them to be a bottleneck as PrIXP only uses simple access to data structures and stores less information than traditional RSes.

First, we implemented a prototype RS-software written in Python, as depicted in Fig. 6.4, including a decision process acting according to the route dispatching mechanism described in Sec. 6.3. To easily manipulate BGP messages within the RS-software, we relied on ExaBGP [EN16], a software tool for easily interfacing and managing BGP sessions in a convenient JSON format. The input of our simulation is a dump of all the routes announced inside a big European IXP in a one hour time interval. We ran two different experiments:

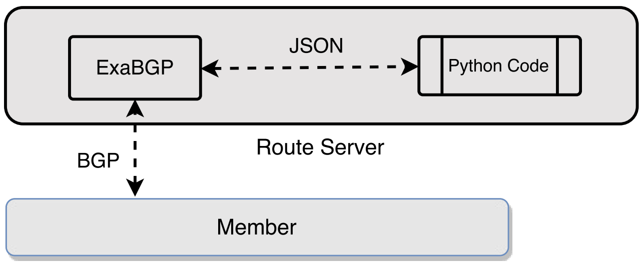


Figure 6.4: Architecture of our RS-software prototype implementation.

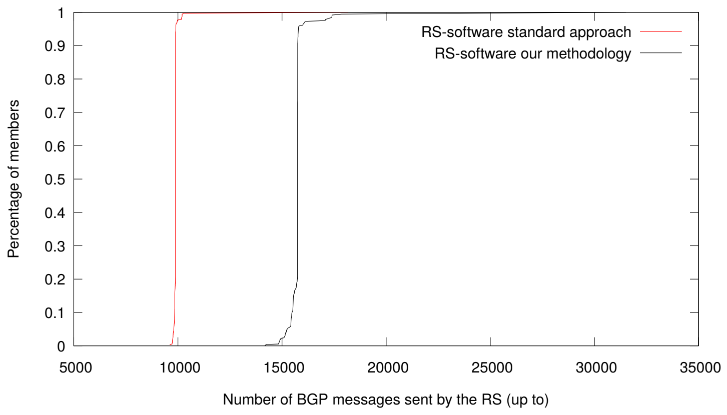


Figure 6.5: CDF of the number of messages issued by the RS-software.

the first one using a traditional RS-software that does not guarantee any privacy, and the second one using PrIXP. During each experiment, we collected the number of exchanged messages to quantify the communication overhead due to the extra channel communication. To put ourselves in the worst-case condition, we assumed that each member is willing to send its route announcements to any other member.

The percentage of members that received at least a certain amount of BGP announcements from the RS-software is depicted in Fig. 6.5. The red line refers to the standard RS-software, whereas the black one represents the CDF for our methodology. We see that in PrIXP around 95% of the members received roughly 5000 BGP announcements more that with respect to the standard

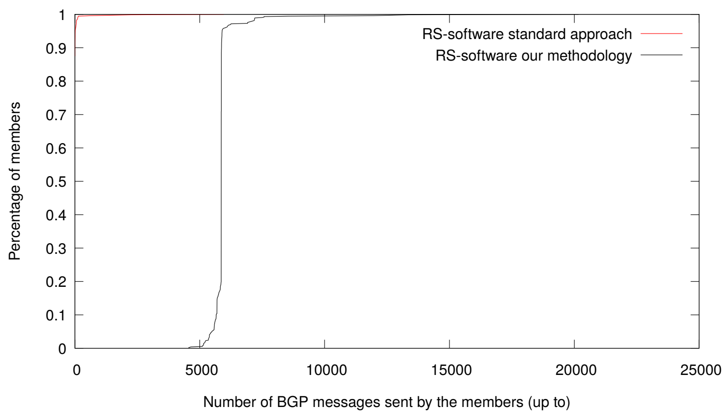


Figure 6.6: CDF of the number of messages issued by members.

RS-software, which is an overhead by a factor of 1.5. We argue that this amount of overhead is affordable for a member, considering the time interval taken into account. The number of messages sent by each member to the RS-software is depicted in Fig. 6.6. Note that the red line is now very close to the leftmost part of the graph, showing that only a few members announce many routes, while the vast majority of the IXP members are not involved in sending many BGP routes. In this case, we observe a significant increase in terms of number of messages, since that amount includes the messages exchanged on the extra communication channel in order to ensure privacy at the RS-software, which is not guaranteed in the standard approach. We argue that it is due to the fact that the PrIXP does not store in memory any routing information at the RS, thus forcing the members to send them when required.

To allow members to verify the integrity of the RS-software, we used *OpenSGX* ([JDK+16, Ope16b]), an open source implementation of SGX. This experiment aims at verifying that the remote attestation mechanism described in Sec. 6.4 behaves as expected. We produced a hash value of the PrIXP implementation. Then, we wrote an SGX program that executes the RS-software only if the hash of the RS-software corresponds to the one of the precomputed hash. To generate the proof P of the SGX program, we used a key issued by the trusted authority, according to Fig. 6.3. We ran the SGX program on a virtual machine acting as RS-machine. After checking the checksum of the RS-software, our SGX program successfully executed it. At that point, we tried to perform a remote attestation from an external client towards the SGX program running on the RS-machine. To do that, we provided the trusted proof P from the external machine to the SGX one, and in that case, the remote attestation succeeded. After that, we altered the code of the RS-software, and the SGX program detected the change as it did not run the malicious RS-software. As the final step, we executed on the RS-machine a malicious SGX program with a proof P' generated using a malicious key, with an altered version of the RS-software. The member detects that the RS-software was tampered since the remote attestation fails.

6.6 Related Work

In this section, we overview the most relevant work to ours along two dimensions: (i) securing the Internet routing computation and (ii) preserving the privacy of the routing policies on the Internet.

Security of Internet routing. Several attempts have been made by the Internet community in order to secure the Internet routing from malicious activity such as IP-prefix hijacks and similar attacks. The set of techniques developed to curb these malicious activities range from Resource Public Key Infrastructure (RPKI) [BA13], which is used to verify whether the originator of a BGP announcement is the legitimate one, to Secure BGP (S-BGP) [KLS00], which allows any entity to verify the authenticity and authorization of BGP control traffic. We note that, beyond large-scale deployment issues with these techniques, none of them can actually be used to guarantee the IXP network will correctly propagate the BGP announcements. The IXP operator can still (i) do not propagate a BGP route or (ii) select any of its known routes as the best one. Nevertheless, an implementation of a RPKI-based route server is in [KK14].

Several efforts have been made to improve the level of security offered by RPKI and S-BGP. These efforts include the most closely related work to ours, SPIDER [ZZG+12], which devised a distributed mechanism that allows the peers of a network to verify a number of nontrivial properties of its interdomain routing decisions (such as adherence to the BGP protocol) without revealing any additional information (beyond those revealed by the underline protocol, i.e., BGP). When casting this mechanism in the IXP setting, SPIDER allows each IXP member to verify that the IXP is not deviating from the BGP protocol (i.e., sending non-best routes), but it requires the IXP members to disclose their routing policies to the IXP operator.

Privacy of Internet routing. In [GSP+12] and [CDC+16], Secure MultiParty Computation (SMPC) techniques have been used in order to compute Internet routing paths without revealing to any party the routing policies of the Internet entities. SMPC is a branch of cryptography that studies the problem of computing a function over their inputs while keeping those inputs private. As the authors themselves recognize [GSP+12], the main drawback of using SMPC lies in the inherent difficulty of scaling it to a large number of participants, as the computational and communication complexity easily becomes a bottleneck, especially when the SMPC function is required to be robust against malicious attackers.

Kim et al. [KSH+15] make extensive use of Intel SGX to preserve the privacy of ISPs' policies and to guarantee the correct propagation of BGP announcements. SGX is a proprietary hardware-based mechanism that allows programmers to create *enclaves* of memory by means of special processor's instructions. In order to limit our dependency with a proprietary building block, we use SGX to remote attestation only, providing the privacy of routing policy in a distributed manner.

6.7 Conclusions and Future Works

During the last decade, IXPs emerged as economically advantageous solutions for interconnecting multiple Internet entities. While RS services have been deployed at IXPs to ease the operators from the burden of managing hundreds of BGP sessions, the usage of such services have been hindered by the privacy concerns regarding the disclosure of the members' routing policies to external commercial parties such as the IXP.

In this work, we designed PrIXP, a RS service that allows to redistribute BGP routing information according to the import/export policies specified by the IXP members while minimizing the risk of leaking that information to any curious or malicious entity. We demonstrated that PrIXP has little message overhead compared to traditional non-secure RSes and it requires only minor modifications at the members' side.

In the next future, we plan to pursue the following directions. First, we intend to improve our prototype implementation, aiming at reducing the control plane overhead introduced by the current version and assessing the computational overhead in our system. Second, we will extend our experimental setup to in order to gather information about other relevant metrics such as the time spent by a member to receive the legitimate routes. Finally, we will devote our efforts towards eliminating any hardware modification at the members side in order to ease the deployment of PrIXP at any IXP by tweaking the BGP protocol.

Part IV

The Role of IXPs in Federated Networks

Chapter 7

SDN, Federated Networks and Internet eXchange Points

Federated networks represent a remunerable operational way allowing federated partners to increase their incomes through a sharing resource process. They have been primarily used in the context of cloud computing; nowadays they are also used to provide connectivity services, like Virtual Private Networks. However, providing such a service by using standard technologies in federated networks requires a non negligible effort from different points of view (e.g. configuration effort).

In this chapter we propose an SDN-based framework aiming at overcoming limitations in currently best practices adopted to issue Virtual Private Networks in federated networks. Relying on the SDN architecture, we propose a method allowing federated providers to quickly and easily create federated networks, reducing unneeded costs (e.g. new hardware), as well as a way for customers to fast access federated services, without any explicit actions from providers. We evaluate our framework by using SDNetkit [ML$^+$17]. We focus on analyzing the impact of our implementation on both control and data plane, in terms of number of control messages exchanged in the network and size of the forwarding tables, respectively.

7.1 Introduction

Federated networks represent a collaborative operational way for Internet Service Providers (ISPs) to increase revenues by sharing resources [GGT10]. A federated network can be defined as a network in which federated partners or members (e.g. ISPs) share their own resources with any other federated member in order to satisfy growing demands from customers or possibly issue value-added services (e.g. services that could not be provisioned without the federated network itself).

One of the most relevant benefits of such a network is the increase of providers' incomes. On the other hand, there are critical steps that must be carried out in order to join a federated network, like identifying, defining cost models, and agreeing on standard operational tasks (e.g. monitoring). Those examples are just a short list of needed steps, but they give the idea that creating a federated network is not so trivial. At the beginning, federated networks were defined to operate in the context of cloud computing. Nevertheless, the promising benefits brought by them encouraged ISPs to provide other services. During the years, several projects arose, like GÉANT [gea17a] and Beacon [bea17], with different aims while sharing the same idea of federation.

After discussing with several Italian ISPs, we asked ourselves whether the benefits of federated networks could be exploited to issue other services (e.g. connectivity). One of the most used connectivity service in today's networks are Virtual Private Networks [RR06] (VPNs). Issuing that service in a network directly managed by a single ISP is not trivial, since many protocols must cooperate in order to set up a VPN. Also, the provisioning of that service is expensive at least in terms of time. As a consequence, it is even more challenging to span a VPN over two or more networks, that are managed by different ISPs by definition of federated network. Actually, such a service is provisioned in a federated fashion by GÉANT [gea17b]. One of the strong points they specify in describing their VPN service is the ability to *fast deliver* it: "5 days are needed". Thus, our question is: Is there a way to reduce such a provisioning time?

In this work we propose a framework allowing federated ISPs to quickly and easily: 1) create a federated network; 2) set up a VPN service; and 3) allow customers to join or leave from the service autonomously, namely without any direct actions (e.g. configuration activity) performed by the ISPs. We define such a service *federated VPN*, namely a VPN allowing customers connected to different federated ISPs' network, but in the same VPN, to exchange traffic with each other. Our framework relies on SDN and is built on top of [dLRB16] and [MLB$^+$17]. The main contributions of this work can be summarized in: 1) providing a configuration language that allows each federated ISP to easily define a federated network as well as quickly configure and provision a federated VPN; 2) providing a set of primitives that allow customers to join or leave from federated VPNs *on-demand*, thus reducing both actions of their ISPs and time required to set up or down the federated VPN; and 3) providing an interaction with the application level (e.g. DNS system) that allows each ISP to keep unchanged any IP address plan previously assigned to its customers.

Our framework does not have any significant impacts on the ISPs' networks. Indeed, it is completely agnostic with respect to any forwarding strategy adopted by the ISP (e.g. IP or MPLS). Also, our framework does not have any impact over any existing configuration: federated VPNs built exploiting our framework coexist with standard VPN set up by using legacy technologies. A customer can be simultaneously served by a federated VPN and a standard one without any limitations. We claim that a strong added-value of our framework, and – consequently – of federated VPNs, is that we allow communication among customers sharing the same IP address plan.

The rest of the chapter is organized as follows. In Sec. 7.2 we review the state of the art. In Sec. 7.3 we discuss today's best practices for federated networks. In Sec. 7.4 we show our framework and how it can be used to tackle the most common problems in today's federated networks. In Sec. 7.5 we present our configuration language and our primitives, explaining how they allow a fast delivery of the service. In Sec. 7.6 we show a complete example, illustrating how SDN interacts with the DNS. In Sec. 7.7 we summarize the benefits of our framework. In Sec. 7.8 we discuss the results of our experiments. Finally, in Sec. 7.9 we draw conclusions and future research perspectives.

7.2 Related Work

In this section, we review the most relevant literature proposing SDN as the architecture to support the provisioning of federated services in federated networks. Also, we compare with proposals to set up VPNs over different ISP's networks.

Federated networks are widely used for cloud computing [cfl12, GGT10, KGK15]. Over the years, several aspects have been addressed, starting from analyzing architectures for federated networks. In [cfl12] authors present a layered architecture in order to provide cloud services (IaaS, PaaS, and SaaS). Such services are provisioned exploiting a collaboration among providers that share their resources, aiming at increasing their incomes.

Providers are interested in federating and providing services in a federated manner because the business model behind such a collaborative network promises costs reduction and remuneration increase. In [GGT10], authors discuss models to guarantee specific levels of remuneration for federated cloud services. Also, toolkits for modeling and simulating cloud services have been discussed in [CRB+11]. Attempts of using SDN to issue federated cloud services have been made, as reported in [KGK15], where authors propose an architecture in which a software agent handles shared resources used to issue the cloud service. Meantime, several research projects, like GÉANT [gea17a] and Beacon [bea17], arose. They build federated networks in which federated services (e.g. VPNs) are issued by sharing resources owned by each federated ISP, investigating the potential of such a model in terms of performance and costs effectiveness.

We argue that other services (e.g. connectivity) beyond cloud computing can take advantages from the federated network model. Indeed, GÉANT network also offers federated VPN services [gea17b] to its customers. In order to provide such a service, they rely on VLANs to cross multiple ISP's network. As they admit, the provision of a VPN in their network requires a non-negligible amount of time (order of days). In [SF14] authors propose a mechanism based on the LISP protocol [FFML13] to span a VPN in a multi-provider network. The main drawback is that each ISP must use LISP.

We believe that the SDN architecture is a key component in dealing with current challenges for federated networks [FBP+10] and for providing new services. In [dLRB16] we proposed a mechanism based on SDN to support end-to-end connectivity spanning several ISP's networks and in [MLB+17] we built on top of that paper a mechanism based on the DNS to simplify the communication among end-hosts. In this work, we extend [dLRB16] and [MLB+17] by providing a framework to easily subscribe to federated VPN services, avoiding delays introduced by provisioning issues. Unfortunately, federated networks still have to deal with challenges [FBP+10] that make the provisioning of federated services difficult and costly, both in terms of money and human resources. Relying on SDN, we address those challenges, proposing a framework that is a first but complete step for today's federated network issues.

7.3 Best Practices for Federated Networks

In this section, relying on the GÉANT project [gea17a], we describe the typical architecture and the best practices adopted to create a federated network.

The main idea behind the GÉANT federated network is what they call *federated PoP* [PGP+]. A federated PoP is a physical place in which all ISPs involved in a federation connect each other. In general, establishing a federated PoP needs many steps, consisting of different activities. For instance, there is the need of establishing connectivity (e.g. by using dark fiber), as well as overcoming technical difficulties (e.g. due to different physical layer technologies). Other steps regard the need of installing and using new hardware (e.g. switches) that will be used by each ISP to connect to each other and all equipments to monitor the services issued by the federation. The network hardware in a federated PoP can be either hardware owned by the provider itself or shared hardware owned by the federation. It is easy to note that the federated PoP architecture strictly recall that of any Internet eXchange Point (IXP), where providers are interconnected in order to allow their customers to exchange traffic.

Surely, federated PoPs allow each ISP to clearly identify a specific way to join a federated network, as well as to isolate the federated network traffic from the standard one; on the other hand, a federated PoP introduces costs for both installing and maintaining (also including configuration effort) the devices used in that place. Moreover, a non trivial agreement process has to be carried out in order to clearly define operations and responsibilities among federated ISPs in the federated PoP. Once a provider connects to a federated PoP and agrees on the policies, it can start to share resources with other providers and to issue services. Even if this model has been recognized to become the standard architecture for federated networks, it also introduces challenges [FBP$^+$10], like: 1) *management*, namely the need of collaboration in standard network operations, like configuration, troubleshooting, and monitoring; 2) *technological differences*, namely the lack of well defined standards could originate problems due to different ways to realize the forward traffic at physical layer; and 3) *user view*, namely the absence of a common interface clearly describing how to access federated services, hiding the collaboration among providers to the final users.

We argue that the aforementioned challenges involve not only technological aspects. On one hand, coordination activities must be carried out in order to plan the architecture (e.g. protocols or components needed to issue services). On the other hand, identifying responsibilities inside the federation itself and agreeing on costs (e.g. federated service prices and the level of remuneration for each provider, possibly based on the resources it shares in the federation) is a task that needs to be accomplished by non-technical staff inside the provider. This observation might have a strong impact especially when a new service is issued for the first time.

Our framework relies on an architecture proposing several mechanisms that solve those problems, making the creation of a federated network, as well as the subscription to and the provisioning of the federated VPN service straightforward and easy. We argue that avoiding the need of having federated PoPs allows federated ISPs to reduce time spent in coordination activities as well as technological and management issues.

7.4 SDN-based Federated Networks

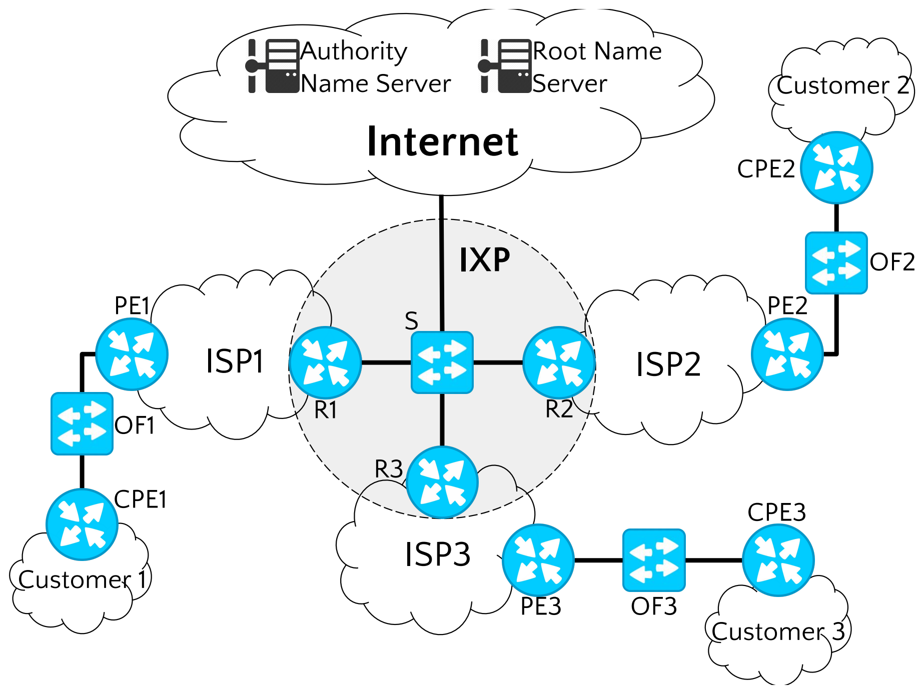
In this section, we present a SDN-based framework dealing with the main problems related to the implementation of federated networks. By relying on the SDN architecture, we argue that our solution is able to address all challenges reported in Sec. 7.3 (and discussed in [FBP+10]), namely: 1) management problems; 2) technological differences problems; and 3) absence of a unified user view.

Of course, Internet can be perceived as the biggest federated network, since providers collaborate with each other in order to provide services to their customers. However, such a collaboration is not based on resource sharing, making Internet different from federated networks built on top of that concept. Also, through the adoption of federated networks, providers are able to issue value-added services, like VPNs spanned over two or more ISPs' networks. Nevertheless, federated networks based on traditional technologies lead to many challenges [FBP+10].

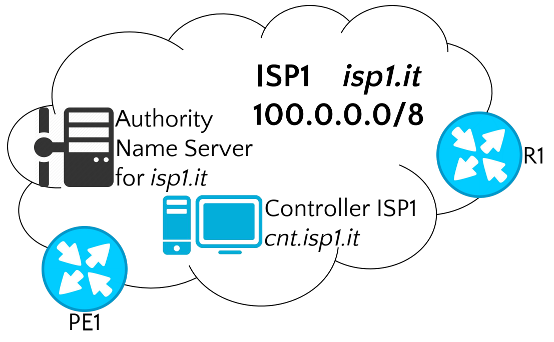
Our framework is completely based on SDN. We choose to rely on that architecture since it brings flexibility in providing services and it also makes the provisioning phase easier. Such a choice allows us to overcome the challenges in current federated networks architecture. Indeed, we identify one main problem in the architecture described in Sec. 7.3, namely the federated PoP. On one hand, such an interconnection point brings several benefits (e.g. clear identification of a place in which providers can federate and clear responsibilities assignment to each federated provider). On the other hand, federated PoPs are *duplicates* of IXPs, requiring further effort for federated providers in terms of expenses and configurations (e.g. buying and managing devices used in the federated PoP). We argue that being connected to an IXP is enough to create a federation and this requirement is easily satisfied by providers.

Our framework relies on this observation. By doing so, we preserve all benefits of a federated PoP (e.g. clear identification of responsibilities) and save addictive costs due to new hardware. Removing the idea of having a federated PoP, combined with the centralized approach offered by SDN, allows us to address and solve the aforementioned challenges. Before going in deep, we present a reference scenario.

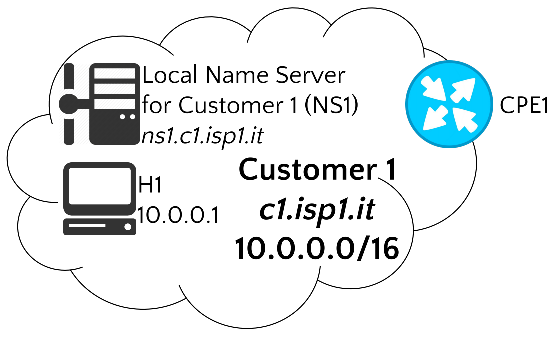
Reference Scenario - A reference scenario for our framework is depicted in Fig. 7.1. We assume that each partner of the federated network is an Internet Service Provider (ISP) offering connectivity to a set of customers. The federated network of Fig. 7.1(a) has exactly three partners whose networks are called ISP1, ISP2, and ISP3, respectively. Such networks run IP-based routing protocols



(a) Overview of a federated network including three ISPs connected through an IXP allowing them to exchange traffic to each other and to access Internet.



(b) A detalied view of the internal architecture of a provider.



(c) A detalied view of the internal architecture of a customer.

Figure 7.1: Reference scenario for SDNS.

inside their backbones (e.g. OSPF for intra-domain routing and BGP for inter-domain routing). Routers R1, R2, and R3 are border routers establishing a BGP peering as in traditional networks not involving federations.

More generally, a federated network can have several partners. We assume that each of them has a border router peering with the border routers of the other partners. As we already said in this section, Internet eXchange Points (IXPs) are a natural place for establishing such peerings, therefore IXPs are convenient premises for setting federated networks. We represent the IXP connecting the federated ISPs through a dashed circle with a light grey background. In the IXP, we assume – without loss of generality – that all ISPs are connected through a legacy (no-SDN) layer 2 switch S. Exploiting the IXP itself, each ISP can also reach Internet. In our scenario, we assume the somewhere in Internet there are (at least) two name servers: 1) a root name server and 2) an authoritative name server for the ISPs' domains.

Routers PE1, PE2 and PE3 are Provider Edges (PEs) and collect the traffic coming from the customers attached to the IPS's network. Each ISP in Fig. 7.1(a) has one customer (Customer1, Customer2 and Customer3, respectively). Each customer is connected to the ISP's network through an IP-speaking router acting as a Customer Premise Equipment (CPE); those devices are CPE1 for Customer1, CPE2 for Customer2, and CPE3 for Customer3. Each of those routers is, in turn, connected to an SDN-enabled device, more specifically an OpenFlow-enabled switch (OF1 for Customer1, OF2 for Customer2, and OF3 for Customer3); placing such devices between the CPEs with the PEs allows us to take the control of all traffic generated by each customer.

Figs. 7.1(b) and 7.1(c) depict the internal architecture of an ISP and of a customer, respectively. Referring to Fig. 7.1(b), each ISP has a public IP subnet used to allow the communication over the Internet and it has a public domain name (isp1.it for ISP1). The same happens for ISP2 and ISP3, even if we do not report in this thesis a specific drawing. Inside each ISP's network, there is an SDN-controller (cnt.isp1.it) having in charge the management of each SDN-enabled device. We assume that each ISP belonging to the federated network has an SDN-controller in order to provide the service. Even for SDN-controllers, the same happens for ISP2 and ISP3.

In this thesis we do not address problems related to robustness in case of controller failures. SDN-controllers have one public IP address used to exchange traffic with each other, and they can reach the SDN-enabled devices by relying on routing protocols running inside ISP's networks. We argue that our architecture is agnostic with respect to all routing protocols running on each ISP's network.

With respect to the internal architecture of a customer (Fig. 7.1(c)), we assume that it has a private IP address subnet used for internal purposes. Also, there is a local name server (NS1 with domain name ns1.c1.isp1.it). The same happens for Customer2 and Customer3, even for the internal IP address subnets: indeed, we allow communication among end-hosts possibly sharing exactly the same IP address. The local name servers might be placed in a publicly accessible portion of the network. If this is the choice, each machine inside the customer must be re-configured pointing to such an external device. Leaving the local name servers inside the local networks, there is no need of this extra effort. Also, placing the those servers behind the SDN-enabled switch allows us to take the control over the DNS traffic generated by the customers.

Referring to Fig. 7.1, when H1 (residing in Customer1) wants to exchange traffic with H2 (residing in Customer2), we say that those customers *join* a federated VPN allowing them to send traffic each other. This operation is steered by the SDN-controllers of each federated ISP, that undertake specific operations in order to set up the federated VPN. Note that by using standard technologies (e.g. Layer 3 [RR06] or Layer 2 VPNs [KR07]), this service cannot be provided, since IP addresses overlap is forbidden. Note that Customer1 can be part of any other VPN provided by ISP1 using standard technologies, as for example MPLS VPN. Also, each ISP can still provide services that are not SDN-based without any restrictions.

In the rest of the section, we address one by one problems related to mangement, technological differences, and unified user view, explaining how we address each of them and which solutions our framework implements.

Management Problems – Management problems happen when common network operations, such as monitoring activities, have to be carried out. Such operations need a strong coordination among members of the federated PoPs [FBP$^+$10] and agreeing to reach that goal might not be a trivial process. It is easy to see that if the federated network is built exploiting a federated PoP, those activities might involve working teams belonging to different providers. Also, systems used to carry out such activities should include the policies of all federated members, leading in an increase of complexity.

Our framework does not introduce any further connection point, except the IXP which each provider is already connected to. By doing so, each federated ISP carries out performance and monitoring activities independently by each other federated provider, reducing both hardware and human resources costs. In addition, as all customers are forced to send traffic passing through SDN-enabled devices (as shown in Fig. 7.1), tracking the traffic belonging to the services provided by the federation is also easy. This is not the only benefit

that our framework brings. Indeed, each provider is able to autonomously accomplish the task related to the recognition of responsibilities, since thare are not shared interconnection points (e.g. federated PoPs) and each federated ISP only manages its own network, keeping costs unchanged.
Technological Difference Problems – Technological problems arise when ISPs adopt different protocols to interconnect each other [FBP+10]. A preliminary step consists in agreeing on shared choices (e.g. in terms of routing protocols), allowing each ISP to interconnect to each other. Unfortunately, there is no a standard interface to access a federated network [FBP+10], resulting in a complication when a service has to be issued.

Our framework does not impose such constraints, since each federated ISP comes with its own infrastructure that is completely independent from each other, allowing providers to choose protocols to use in their network, preventing coordination activities needed by the federated PoPs. The only interconnection point, as we said, is the IXP, where each provider is already connected to in order to access Internet and exchange traffic. Our framework requires to have several SDN-enabled switches, autonomously managed by each providers. Thus, SDN-controllers take the control over the traffic generated by the customers and it is very simple to achieve with SDN.
Unified User View Problems – Unified user view problems occur when a customer does not perceive the federated network as a single one. An example of this problem is the following: consider the case in which a customer wants to join a federated VPN. That customer asks to its provider to set up the federated VPN allowing it to exchange traffic with another customer connected to the network of a federated partner. If such a request takes a non negligible amount of time for being accomplished (e.g. order of hours or days), a customer might perceive that its provider is not able to issue that service autonomously, but it needs to collaborate with other ISPs, making the federated network visible to the users.

A solution to this problem consists in providing a unified way to access federated services that allows each federated ISP to accomplish on-demand customer requests. Also, after receiving such requests, each federated ISP must act as much as possible independently, consequently reducing the collaboration in order to gain in terms of both amount of involved human resources (e.g. technical staff for devices configuration) and required time to provision the service. This is what our framework does. It makes available a set of primitives that each customer uses to access a federated service. By doing so, each ISP has a unified and standard way to accept the customer requests and to start a collaboration with one or more federated ISPs in order to issue the service. In our framework such a task is carried out by the SDN-controllers, that cooperate with each other after a primitive has been received. This collaboration leads on exchanging information related to the federated service and does not require strict constraints, except the reachability among the SDN-controllers (e.g. by exposing them on Internet using public IP addresses).

7.5 Subscribing to a SDN Federated VPN Service

In this section we describe a configuration language for supporting federated networks and VPN services and a set of primitives allowing customers to join or leave such services.

Our configuration language is simple and it just contains information about federated networks and the federated VPN service, without any impact on any existing configuration. The configuration, written with our configuration language, is located at the SDN-controller, and it does not require any additional configuration over standard IP-speaking routers. In particular, our configuration language includes a set of static information (e.g. which are the federated networks the provider belongs to and all information about the SDN-controllers of other federated ISPs) and a set of dynamic ones (e.g. list of federated VPNs), that are updated based on the customer requests performed by using our primitives.

Federated VPN is a collaborative service. Therefore, all SDN-controllers must have the configuration for that service always updated and consistent. To support this requirement, each SDN-controller has a public IP address, allowing it to exchange information with each other SDN-controller in the federation. We also define a set of primitives used to keep the configuration consistent; they are used by customers that want to join (or leave) the service at any time. This makes our proposal more flexible and scalable with respect to standard VPN services, that require changes in the configuration files of the network devices in order to support such an *on-demand* feature. We argue that our framework makes the whole provisioning process more *agile*. For sure, the decision of federating with other providers requires a set of agreements that must be carried out (e.g. cost models) and they are not address by our framework. Nevertheless, our framework provides several mechanisms to make the creation of federated networks and the provisioning of federated services easier.

In the rest of the section we present two building blocks of our framework: the configuration language and the primitives. First, we show our configuration

language, as it provides constructs to define information about the federated network as well as the federated VPNs service. Second, we illustrate our primitives. We choose this order of presentation since the usage of primitives changes the content of the configuration at the SDN-controllers. We argue that our configuration language and our primitives, together with the SDN-based architecture of our framework, represent our solution for solving problems discussed in Sec. 7.4.
A Configuration Language for Federated Networks and Federated VPNs Services – Our configuration language is XML-based. The goal is to specify a set of parameters used to easily set up both federated networks and federated VPNs, without modifying any existing configurations. Our choice of relying on XML does not restrict the adoption of any other formats (e.g. JSON), as long as the semantic stays unchanged. The configuration is the input of the SDN-controller, that – based on its content – allows or denies a customer to access the service. The configuration has the following structure:

|<federations>|
| <federation name="*federation_ name*">|
| <myself></myself>|
| <isps></isps>|
| <vpns></vpns>|
| </federation>|
|</federations>|

The root of the XML tree is the element |<federations>|, containing all federated networks the ISP belongs to. Indeed, each partner can participate in more federated networks at the same time and each customer can join multiple federated VPNs belonging to different federated networks. We define the element |<federation>| as a child of the root element and it contains information about the ISPs in the federated network. We assume that each federated network has a name that is globally unique. Basically, the element |<federation>| represents a federated network and it is added to the configuration after two or more ISPs agree in setting up a federated network. Such an element instantiates the will of joining a federation after the whole process of agreements has been carried out. Inside this element we identify three subtrees: 1) |<myself>|, containing all information about the SDN-controller of the ISP's network in which this configuration is deployed; 2) |<isps>|, containing the list of all the other SDN-controllers belonging to the federated network; 3) |<vpns>|, containing the list of all federated VPNs defined inside the federated network and information on customers belonging to each VPN. We now discuss each subtree in order to show which information belongs to each element and which are the pieces of this configuration that the SDN-controllers have to exchange with each other.

In the |<myself>| element, the network operator specifies some basic parameters for the controller. This element has the following structure:
|<myself>|
| <isp id="*isp_ id*" name="*isp_ name*">|
| <controller name="*ctrl_ name*" ip="*public_ ip*" />|
| <nat fake="*fake_ ip_ subnet*" public="*public_ ip_ subnet*" />|
| </isp>|
|</myself>|
Each ISP is easily recognizable inside a federated network starting from the pair formed by an id, that must be unique inside the federated network itself (e.g. the AS number of the ISP's network) and a name. Element |<controller>| contains some basic information about the SDN-controller, consisting in a name and an IP address that must be public in order to guarantee the reachability of the machine. The element |<nat>| contains two subnets used to NAT the packets.

Intuitively, the fake_ip_subnet is used in scenarios where hosts sharing the same IP address need to exchange traffic, as reported in Fig. 7.1. For this pool of addresses there are no requirements. They can be either private or public IP addresses and they can be used elsewhere in the federated VPN. Indeed, IP addresses in the fake_ip_subnet are used as a temporary replacement of the actual destination IP address. The public_ip_subnet is used to translate private addresses into public ones, as in standard NAT implementation. Note that, by doing so, any forms of tunneling are avoided, resulting in the full MTU being kept available. Note that if the ISP belongs to multiple federations, the |<myself>| element must be declared once for each federation. Indeed that element cannot be promoted as child of the root element |<federations>| since, in general, an ISP participates to different federated networks using different values for the |<nat>| element's parameters.

The element |<isps>| is populated with information about all SDN-controllers belonging to the federated network, except what is already written in the |<myself>| element. Each of those pieces of information is enclosed inside the |<isp>| element, as shown in the following:
|<isp id="*isp_ id*" name="*isp_ name*">|
| <controller name="*ctrl_ name*" ip="*public_ ip*" />|
| <nat public="*public_ ip_ subnet*" />|
|</isp>|
This part of the configuration is very similar to what is inside |<myself>| subtree,

except that the element |<nat>| does not contain the fake_ip_subnet, since it is used from the SDN-controller of the ISP's network in which the end-host that starts the communication resides. We clarify this choice in Sec. 7.6.

To better illustrate the semantic of our configuration language, we now provide a configuration example in accord with Fig. 7.1; at the moment, we set aside the |<vpns>| subtree, since it includes information about the federated VPN service for a specific federated network. We focus our attention on the SDN-controller of ISP1, namely cnt.isp1.it.

```
|<federations>|
| <federation name="ISP1-ISP2-net">|
| <myself>|
| <isp id="1" name="isp1.it">|
| <controller name="cnt.isp1.it" ip="100.100.100.1" />|
| <nat fake="192.168.0.0/24" public="100.200.0.0/24" />|
| </isp>|
| </myself>|
| <isps>|
| <isp id="2" name="isp2.it">|
| <controller name="cnt.isp2.it" ip="200.200.200.1" />|
| <nat public="200.150.0.0/24" />|
| </isp>|
| </isps>|
| </federation>|
|</federations>|
```

By giving this configuration as input, the ISP1's SDN-controller learns that it is part of a federated network called ISP1 – ISP2 – net and a federated partner is a provider called isp2.it whose SDN-controller is cnt.isp2.it with IP address 200.200.200.1. It is important to note that such a configuration is static, namely it is manually configured by each ISP after reaching agreements with other ISPs on creating a federated network. We point out that having such an information in the configuration is enough to establish the federated network.

The last element is |<vpns>| and it contains all information about all federated VPNs in the federated network set up in order to allow customers to exchange traffic. Inside this subtree, a list of |<vpn>| elements can be specified. Each of them has following structure:

```
|<vpn id="vpn_id" name="vpn_name">|
| <isp id="isp_id" name="isp_name">|
| <customer name="customer_name">|
| <site name="site_ name" timestamp="time">|
| <datapath|
| ip="ip_ address" in_port="port_ number"|
| out_port="port_ number" />|
| <subnet private_network="ip_ subnet" />|
| <ns domain="domain" ip_address="ns_ ip" />|
| </site>|
| </customer>|
| </isp>|
|</vpn>|
```

This element contains many details. The first one is composed by generic data representing a way to uniquely identify a federated VPN starting from an id and a name. The element |<isp>| groups information about which customer (potentially, more than one) joins the VPN. Each customer is identified by a globally unique name. Information about where its traffic comes from and goes out are expressed in the |<datapath>| element. In this piece of configuration it is possible to declare which is the private_network used by the customer and information about its local name server, namely the domain whom it is the authority and, optionally, its ip_address.

Still referring to Fig. 7.1, we now provide an example of |<vpns>| element configuration in the case in which both Customer1 and Customer2 ask to join a federated VPN. Such an action is accomplished by each customer by using the primitives we are going to discuss in the rest of the section. We always refer to the configuration of ISP1's SDN-controller.

```
|<vpns>|
| <vpn id="1" name="C1-C2-vpn">|
| <isp id="1" name="isp1.it">|
| <customer name="c1.isp1.it">|
| <site name="s1.c1.isp1.it" timestamp="0">|
| <datapath ip="100.0.0.123" in_port="1" out_port="2" />|
| <subnet private_network="10.0.0.0/16" />|
| <ns domain="ns1.c1.isp1.it" ip_address="10.0.0.3" />|
| </site>|
| </customer>|
| </isp>|
| <isp id="2" name="isp2.it">|
| <customer name="c2.isp2.it">|
| <site name="s1.c2.isp2.it" timestamp="0">|
| <subnet private_network="10.0.0.3" />|
| <ns domain="n2.c2.isp2.it ip_address="10.0.0.3" />|
| </site>|
| </customer>|
| </isp>|
| </vpn>|
|</vpns>|
```

By reading this piece of configuration, ISP1's SDN-controller learns several information related to the federated service. First, a VPN called C1 − C2 − vpn (such a federated VPN has also an id that we assume to be unique in the federation) exists. Second, two customers are interested in exchanging traffic with each other. In particular, those customers are: 1) Customer1 connected to the ISP1's network; those information are taken from elements |<isp>| and |<customer>|. The customer is connected to the ISP1's network through an SDN-enabled switch whose IP address is 100.0.0.123; its traffic comes from SDN-enabled switch port number 1 (the SDN-enabled port connecting OF1 and CPE1) and goes out from SDN-enabled switch port number 2 (the SDN-enabled port connecting OF1 and PE1). Those information are provided by the |<datapath>| element. The subnet used by the customer is reported in the |<subnet>| element, whereas information about which is the local name server for that customer are found in the |<ns>| element and 2) Customer2 connected the ISP2's network; those information are still taken from elements |<isp>| and |<customer>|. In the case in which the customer is a remote one, namely is not directly connected to network of the ISP (ISP1 in this example), information about the SDN-enabled switch are not provided, since the ISP1's SDN-controller does not manage that switch. Even in this case, the subnet used by the customer is reported in the |<subnet>| element, whereas information about which is the local name server for that customer are found in the |<ns>| element. In contrast with the information enclosed inside |<myself>| and |<isps>| subtrees, the content of the |<vpns>| element is dynamically generated. Indeed, federated VPNs' parameters are reported in the configuration exploiting several primitives allowing customers to easily access federated VPN.

Primitives to Join Federated VPN Services – Each ISP belonging to a federated network makes available to all its customers a set of primitives that they can use to join or leave the federated VPN service. Those primitives are received by the SDN-controller, that performs checks in order to allow (or deny) a customer to join the service. For example, referring to Fig. 7.1, every primitive sent by Customer1 will be forwarded to and checked by ISP1's SDN-controller.

We define three main primitives: *Insert*, *Update*, and *Delete*. The *Insert* primitive is used by a customer to join a federated VPN. Using this primitive, the

customer specifies several parameters. With the *Update* primitive, the customer can ask the SDN-controller to modify the parameters previously declared by the *Insert* primitive (e.g. by adding or removing information). Finally, using the *Delete* primitive, a customer exits the federated VPN. By using those primitives, a customer can specify policies in order to allow or deny other customers to exchange traffic with it. Such policies are verified by all the SDN-controllers in the federated network and the result of such an operation is sent back to each customer. We now describe the semantics of these primitives.
Insert – By using this primitive, a customer asks its provider to join the federated VPN service. *Insert* takes as input four parameters: 1) Customer that is the name of the customer; 2) Description is a set of parameters describing in detail information about the customer. In this set, a customer specifies its subnet, and (optionally) the IP address of a local name server. Those information are translated into the element |<site>| contained in the subtree |<vpn>| of the configuration. Note that the information about the |<datapath>| element can be inferred by the SDN-controller exploiting proper data structures defined by the OpenFlow protocol; 3) VPN is the ID of the federated VPN which the customer joins. By looking at this parameter, the SDN-controller can properly identify the |<vpn>| subtree to update. Indeed, there are many VPNs in the federated network, each of them containing different customers. After choosing the federated VPN, the customer can also express a set of policies, declaring the set of customers inside the federated VPN which it is available to exchange traffic with; 4) Time contains information about how much time a customer wants to use the federated VPN service. After that time, the customer is no longer part of the service. This information is stored as the value of the parameter |timestamp| of the subtree |<site>|.

Referring to Fig. 7.1, an example of the primitive *Insert* sent by Customer1 to ISP1's SDN-controller is the following:
| Primitive: **INSERT**|
| Customer:|
| + Name: **c1.isp1.it**|
| + Site: **s1.c1.isp1.it**|
| Description:|
| + Customer IP subnet: **10.0.0.0/16**|
| + Local Name Server:|
| + URL: **ns.c1.isp1.it**|
| + IP Address: **10.0.0.3**|
| VPN: **C1-C2-vpn**|
| Time: **0**|

To create the federated VPN, Customer2 has to send a primitive to ISP2's SDN-controller similar to the previous one.

Upon receiving the *Insert* primitive, each SDN-controller undertakes a set of operations cooperating with all other SDN-controllers in the federation in order to guarantee the access to the federated VPN service to its customer. Referring to the previous example, after receiving the *Insert* primitive sent by Customer1, ISP1's SDN-controller exploits the information contained in the primitive to populate the configuration as previously shown in this section. After that, it sends a copy of this primitive to ISP2's SDN-controller, namely the SDN-controller of the federated partner for the federated network $\mathsf{ISP1} - \mathsf{ISP2} - \mathsf{net}$. We now go in deep explaining how each SDN-controller dynamically populate its configuration. Note that, only the |<vpns>| element of our configuration language is involved in this process.

First, ISP1's SDN-controller checks whether a federated VPN having the name reported in the *Insert* primitive corresponding to the key VPN already exists. If it is not the case, then the SDN-controller creates a new |<vpn>| element assigning to it an id automatically generated and the name reported in the primitive, namely $\mathsf{C1} - \mathsf{C2} - \mathsf{vpn}$. Second, the SDN-controller starts to add information about the customer. In particular, it knows that the customer who sent the primitive is one of its customers, so it takes from |<myself>| subtree information about itself. By doing so, it is able to create the element |<isp id="1" name="isp1.it">| as child of the element |<vpn id="1" name="C1-C2-vpn">|.

After that, the SDN-controller starts to add the information related to the customer. Based on the information associated to the keys Customer and Time reported in the primitive, new elements are added to the configuration, namely the element |<customer name="c1.isp1.it">| and its child |<site name="s1.c1.isp1.it" timestamp="0">|. At this point, the SDN-controller also adds the |<datapath>| element; such an information is inferred by inspecting the traffic coming from the SDN-enabled switch, which the traffic generated by each customer is forced to pass through. Finally, customer's information are included in the configuration, by creating the elements |<subnet private_network="10.0.0.0/16" />| exploiting the key CustomerIPsubnet of the element Description carried by the primitive and |<ns domain="ns.c1.isp1.it" ip_address="10.0.0.3" />| exploiting the keys URL and IPAddress of the sub-element LocalNameServer contained in the primitive. We point out that in case the federated VPN already exists, those steps are skipped, since those information are already in the configuration. ISP2's SDN-controller perfomrs the same operation.

After performing those checks, ISP1's SDN-controller sends the primitive received by Customer1 to ISP2's SDN-controller, as well as ISP2's does the same with the primitive received by Customer2. Once that message reaches the destination SDN-controller, it undertakes several operations over its configuration based on the content of the receipt message. Considering ISP1's SDN-controller, it does the following. First, based on the VPN value of the primitive, it selects the right federated VPN browsing the |<vpns>| element. After selecting the right |<vpn>| element (|<vpn id="1" name="C1-C2-vpn">| in this example), it adds the |<isp>| element for the federated partner. That information is inferred by observing the source of the message. In the example, ISP1's SDN-controller writes in the configuration the following element: |<isp id="2" name="isp2.it">|. Note that this information are already enclosed in the subtree |<isps>|, collecting all federated ISPs of the federation. Finally, the SDN-controller is able to populate the remaining information, namely the |<customer>|, |<site>|, |<subnet>|, and |<ns>| elements by simply browsing the content of the received *Insert* primitive. Note that the same happens at the ISP2's SDN-controller. At this point both the SDN-controller have the needed information to provide the federated VPN services.

We highlight that no human resources have been involved in this procedure and the service is provisioned without any delay potentially introduced by the federated nature of the network. By adopting our framework, the collaboration among providers in order to set up a federated network, as well as to provision a service, is completely delegated to the SDN-controller by means of a point-to-point communication.

Update - By using this primitive, a customer can modify what declared in the *Insert* primitive. Indeed, *Update* takes as input the same parameters of *Insert*. Also, this primitive is used by an SDN-controller to keep the information of its customers updated with all other SDN-controllers in the network. Basically, each SDN-controller processes such a primitive as it was an *Insert*, changing values in the configuration if required.

Delete - By using this primitive, a customer can leave the federated VPN before the Time parameter declared in the *Insert* primitive expires. *Delete* takes three parameters as input: Customer, Description, and VPN. They have the same semantic described for the *Insert* primitive. Upon receiving this message, the SDN-controller sends it to all other SDN-controllers in the federation, in order to make them aware of the intention of the customer to leave the federated VPN. By receiving this primitive, every SDN-controller can update its configuration for the federated VPN specified as a parameter, making the information consistent.

In this thesis, we do not consider security and authentication issues that can be tackled using any standard application level security protocol. For instance we suppose that all the exchanged information flows using either Secure Socket Layer (SSL) or Transport Layer Security (TLS) technologies.

7.6 A Complete Example

In this section we provide a complete example of our framework. We are going to show the whole interaction between two end-hosts belonging to two different customers connected to different ISPs' network. Referring to Fig. 7.1, we show an example of traffic exchange between two hosts residing in different customers connected to different ISP's networks, but sharing the same IP address. We suppose that host H1, whose domain name is h1.c1.isp1.it and its IP address is 10.0.0.1, resides in Customer1 and host H2, whose domain name is h2.c2.isp2.it and its IP address is 10.0.0.1, resides in Customer2. Consequently, we call H1 source and H2 destination. We divide the example in three steps: 1) federated VPN access request performed by source and destination; 2) domain name resolution undertaken by the source; and 3) IP traffic sent by the source towards the destination.

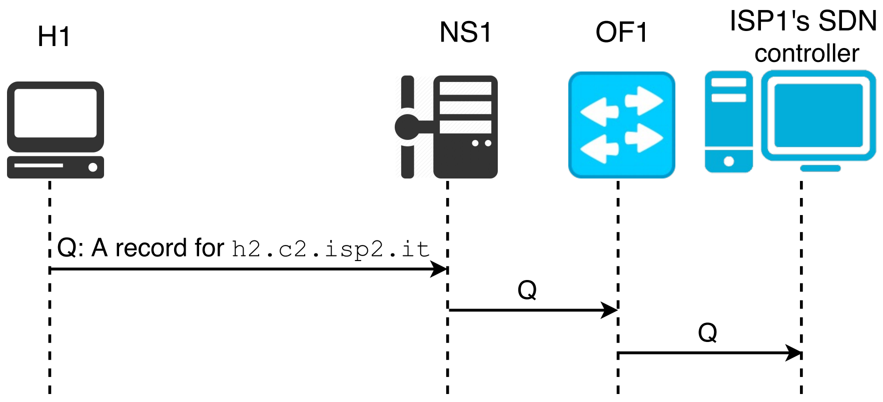
1) Accessing the Federated VPN Service – As described in Sec. 7.5, the first operation carried out by customers that want to join the federated VPN service is to require the access to the service itself. It is done by sending to the SDN-controllers the *Insert* primitive, that is handled accordingly to the description reported in Sec. 7.5. After this step, each SDN-controller has a suitable configuration, allowing it to provision the service.

2) Name Resolution Process – It is very common in the Internet establishing a communication between end-systems starting from the destination URL. Our framework includes a SDN-steered name resolution process that works as follows.

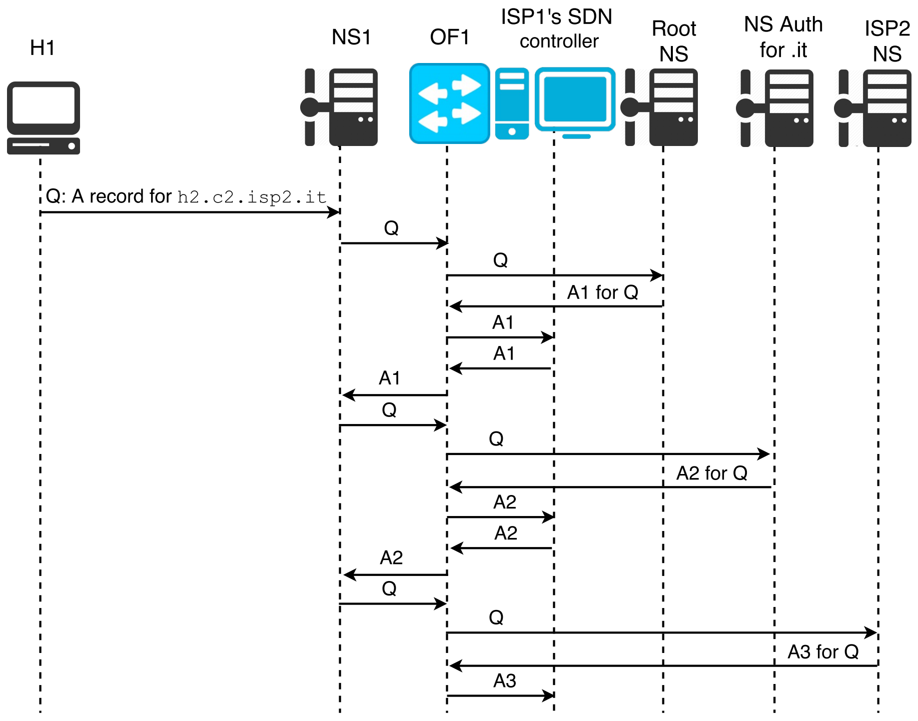
When the source wants to exchange traffic with the destination starting a domain name, it sends a recursive DNS request message to its local name server. In our example, H1 starts a recursive DNS lookup by sending a DNS request to NS1, in order to obtain the H2's IP address (in this example we concentrate on A resource record, but the interaction is analogous to any other type of resource records). According with the DNS name resolution process, that we report for reader convenience, after receiving the recursive DNS request message from H1, NS1 performs a set of iterative DNS queries to obtain the IP address of the destination. The name resolution process undertaken by NS1 starts by contacting the root name server, and it finishes when the authoritative name server for the destination issues a DNS answer message containing the destination's IP address. It is easy to see that in our example such interaction cannot take place: if IP address overlap occurs, we cannot exclude that the local name servers NS1 and NS2 have exactly the same IP address. If this is the case, when NS1 tries to send a DNS request message to NS2, that packet will never exit Customer1's network. A very simple solution is to move out the local name servers from the private network of the customers. This operation consists in changing the IP address of the name servers from a private IP address to a public one. However, such a choice has a non negligible impact on the configuration (e.g. reconfiguration of the end-hosts is also needed) and it is unclear that the customer wants to make public its local name server.

We propose a mechanism relying on SDN to allow local name servers with IP addresses in the same subnet (potentially having exactly the same IP address) to exchange DNS traffic. By observing the DNS traffic, the SDN-controllers can manipulate it in a suitable way, that is transparent for the end users. Our proposal does not require to place any DNS daemon (e.g. BIND [bin17]) at the SDN-controller. This prevents to introduce any other configuration effort. We only need that the whole traffic generated by customers passes through the SDN-enabled switch, in order to be (possibly) forwarded to the SDN-controller. Our proposal is based on two main steps: 1) We determine which is the IPS's network hosting the destination and acquire the IP address of the authoritative name server for the domain of the destination; 2) We resolve the destination's domain name. The second step requires a communication among the SDN-controllers (cnt.isp1.it and cnt.isp2.it in our reference example). In rest of the section we assume that H1 has domain name h1.c1.isp1.it whereas the domain name associated to H2 is h2.c2.isp2.it.
Determining the IP Address of a Name Server – Since the SDN-controller has to interact with local name servers placed in private networks with private IP addresses, it needs two important information: the first one is the customer's network in which that name server resides, and the second is the IP address of that name server. To achieve this, we propose two different approaches, and we exploit the configuration described in Sec. 7.5. Indeed, the element |<site>| contains the whole needed information. The difference between these two approaches resides in the fact that in the first one the IP address of the customer's local name server is in the configuration of the SDN-controller, whereas in the second it is not, as described in Sec. 7.5. We call these two different scenario *Full* and *Partial configuration*, respectively. The interaction among network devices and machines are depicted in Fig. 7.2.

The interaction depicted in Fig. 7.2(a) refers to the *Full configuration* scenario and it is the simplest one. Indeed, once H1 starts the recursive name



(a) Interaction among OpenFlow switch, name server, and SDN-controller in case of *Full configuration* scenario.



(b) Interaction among OpenFlow switch, name servers, and SDN-controller in case of *Partial configuration* scenario.

Figure 7.2: Determining the IP address of a Local Name Server.

resolution process, the iterative query issued by NS1 is intercepted by OF1 and it is sent to the controller. By browsing its configuration, the ISP1's SDN-controller is able to determine the IP address of the name server that is authority for the destination (h2.c2.isp2.it). By recalling what we said in Sec. 7.5, that information is in the configuration file (see [MLB+17] for further details).

The *Partial configuration* scenario is more interesting to address, even if the interaction among network devices and machines (e.g. name servers and SDN-controllers) is more complex, as shown in Fig. 7.2(b). In Sec. 7.5, we argued that the specification of the IP address of the local name servers is optional. Such a choice is motivated by two reasons. First, it simplifies the configuration. Second, since local name server typically has a private IP address, a customer could change it without notifying the ISP, leading to possible misconfiguration problems among the SDN-controllers. Hence, we define a technique to retrieve this information, avoiding such an issue.

At the beginning, H1 sends a recursive DNS request message to NS1, which starts the iterative part of the name resolution process by querying the root name server. With respect to the first approach we discussed, the SDN-enabled switch forwards this packet in the ISP1's network without sending it to the SDN-controller, so that it can reach the root name server. Upon receiving that DNS request message, the root name server answers with a DNS answer message containing information about the authoritative name server for that domain. Upon receiving the DNS answer message coming from the root name server, OF1 forwards this packet to ISP1's SDN-controller. Since we are assuming that the authoritative name servers of all the customers are private, the SDN-controller inspects the content of the DNS answer, aiming at verifying whether it contains some information on the authoritative name server for the destination's domain. If it is not the case, the SDN-controller sends that packet to OF1, by instructing that device to forward the DNS answer to NS1. This process carries on until a DNS answer containing information about the IP address of NS2 (reported as a *glue record* of a NS DNS record) reaches the SDN-enabled switch, allowing ISP1's SDN-controller to understand which is the IP address of NS2. In this case, the DNS answer message is not forwarded to NS1, preventing it to exchange traffic with a name server potentially having the same IP address, but residing in a different network (we recall our assumption that customers residing in different IPS's networks might share the same IP subnet without any restrictions).

There are differences between the two approaches we presented. First, in the *Full configuration* approach there are no other name servers involved in the communication except NS1. Also, the SDN-controller looks at the DNS request

messages produced by NS1. Second, in the *Partial configuration* approach, other name servers are involved in the communication and the SDN-controller inspects the DNS answer messages sent by those name servers. In summary, in the *Partial configuration* approach, we observe a higher number of DNS and, consequently, OpenFlow messages with respect to the *Full configuration* approach. Reducing the amount of information in the configuration impacts the number of messages, but such an impact does not affect the scalability (our framework does not introduce additional DNS messages).
Resolving Domain Names in Presence of IP Addresses Overlap – Up to now, we described how a SDN-controller determines the IP address of the authority name server for the destination and information about which is the ISP's network hosting that name server. Now, we can describe in detail how our SDN-based technique performs the name resolution process. With respect to the standard DNS name resolution process, in our approach the communication between NS1 and NS2 is mediated by the SDN-controller of the source, namely cnt.isp1.it in our example. Note that this mediation is needed, since the IP address of NS2 is in the same subnet of NS1, so if NS1 tries to directly send a DNS request message to NS2, that packet will never leave Customer1's network.

Once ISP1's SDN-controller acquires the IP address of NS2 (the authority name server for the destination), it issues a DNS request message Q directed to that name server based on the DNS request message produced by the source and it sends that DNS message to ISP2's SDN-controller by using a dedicated communication channel. This is possible because the public IP address of each SDN-controller in the federated network is part of the configuration. Upon receiving Q, ISP2's SDN-controller forwards it to the correct name server (this information is part of the configuration), which replies with H2's IP address. Such a DNS answer reaches ISP2's SDN-controller (it passes through OF2).

After receiving the DNS answer message issued by NS2, ISP2's SDN-controller sends that DNS message to ISP1's SDN-controller. Consider that, before forwarding the DNS answer message to NS1, ISP1's SDN-controller must check whether the destination host has an IP address that is in the same subnet of the source. Such a check is mandatory, since we allow the communication with a potentially fully overlap of IP addresses, and in this case H1 and H2 exactly share the IP address 10.0.0.1. Hence, ISP1's SDN-controller has to change the IP address contained in the DNS answer message, preventing H1 to send traffic inside its network, or to itself. To do that, each controller owns a set of fake IP addresses to use for this purpose, that are declared in the configuration as shown in Sec. 7.5. ISP1's SDN-controller picks an IP address from the fake set and replaces the original H2's IP address with the fake one,

keeping this association in suitable internal data structures. At the same time, it sends to the SDN-enabled switch a set of rules to forward the traffic according to this IP address replacement action. In this way, H1 is not aware of the fact that it is sending traffic to a destination with an IP address in the same subnet. It is interesting to note that, by using this technique, also NS1 and NS2 can share the same IP address, since the interaction between these two name servers is mediated by the SDN-controllers. At this point, H2's domain name has been resolved and H1 is able to send traffic.
3. Sending IP Traffic to the Destination – Once the source acquires the IP address of the destination, it starts to send traffic. Since the communication is being established between end-hosts with private IP addresses, translation strategies are needed. We now describe the Network Address and Port Translation (NAPT) strategies that we apply to support communications between hosts in different customer sites within a federated network. These strategies are used to alter the IP addresses (and, possibly, the TCP/UDP ports) of exchanged packets in such a way that traffic between hosts with private addresses can be routed on a public IP network. Our approach differs from the standard usage of NAPT [SH99] for at least two aspects: 1) we alter both the source and the destination IP address and port of outbound packets (rewriting the private source IP address is not strictly required, but we still do it in order to prevent packets with private addresses from traversing a public network), and 2) we allocate resources from the pool of public IP addresses and ports using various different strategies. Address translations are performed by SDN-enabled switches according to packet manipulation rules in their SDN flow tables. As soon as the source emits a packet to establish a TCP/UDP connection towards the destination, the SDN-controllers install on OF1 and OF2 suitable translation flow entries to support the communication between the hosts. We describe three strategies, called: 1) Client Port Preservation (CPP); 2) Client Announces Port Selection (CAPS); and 3) Lazy Address and Port Selection (LAPS). In the rest of the section, we present the CPP strategy (the description of the CAPS and LAPS can be found in [dLRB16]) and we refer to specific OpenFlow messages, whose definitions are in [Ope14b].
Client Port Preservation – This NAPT strategy is inspired by the "port preservation" approach in [AJ07]: for this reason we call it *Client Port Preservation* (CPP). According to this approach, the internal (private) source TCP/UDP port number of an outbound packet should be kept untouched, as long as this is possible: only if two hosts use the same source port number, then they should be mapped to two different public IP addresses. We describe the CPP strategy exploiting the sequence diagram in Fig. 7.3.

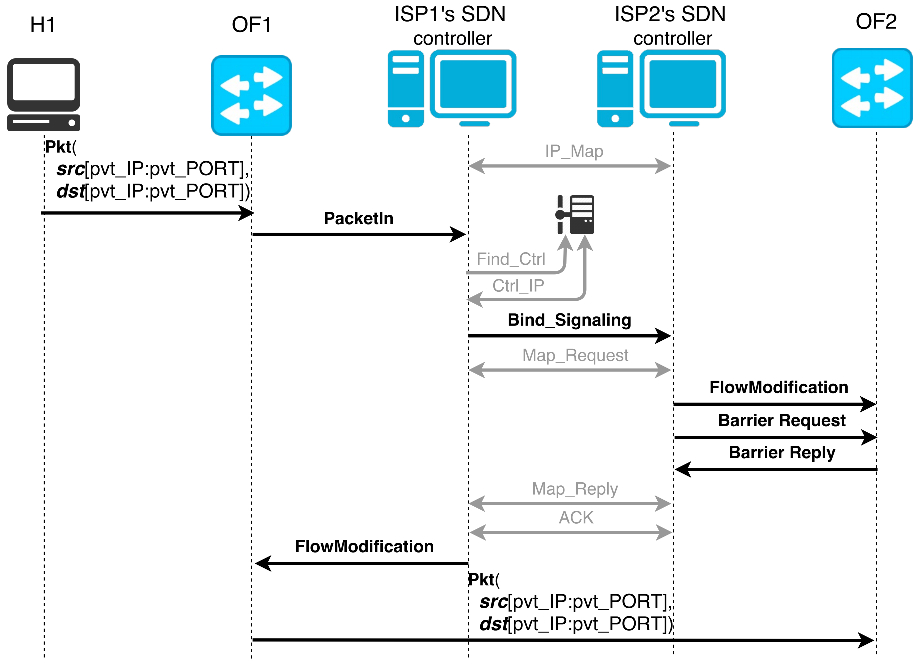


Figure 7.3: Messages exchanged to support communication between H1 and H2, for different translation strategies.

The horizontal arrows in the figure represent messages exchanged among OF1, ISP1’s SDN-controller, ISP2’s SDN-controller, and OF2 to support such a connection. Black arrows represent messages that are common to all our address translation strategies, whereas gray arrows represent those that are required only by some of them. The CPP strategy works as follows. Each controller has a pool of public IP addresses that can be used to perform address translation, as reported in Sec. 7.5. Before any packet exchanges takes place, all the SDN-controllers involved in the federated VPN mutually exchange messages (IP_Map) carrying information about their private address space: thus, every SDN-controller becomes aware of the existence and location of every IP subnet in the federation. After that, assume that a packet for a new TCP/UDP connection is received by OF1. We indicate with **src**[pvt_IP:pvt_PORT] the private IP address and TCP/UDP port of the packet’s source host (H1), and with **dst**[pvt_IP:pvt_PORT] the private IP and port of the packet's destination host (H2). OF1 buffers the received packet and forwards a copy of it to ISP1's SDN-controller (PacketIn message). It picks from its own pool an available public IP address **src**[pbl_IP] to be associated with **src**[pvt_IP], keeping port **src**[pvt_PORT] untouched, then it notifies the binding between **src**[pvt_IP] and **src**[pbl_IP] to ISP2's SDN-controller (Bind_Signaling message). ISP1's SDN-controller also asks ISP2's SDN-controller for a public IP address and port to be used to contact the destination host (Map_Request message). At this point, ISP2's SDN-controller associates a public IP address **dst**[pbl_IP] from its own pool to **dst**[pvt_IP], and an available port **dst**[pbl_PORT] to **dst**[pvt_PORT], and sends FlowModifications to OF2 to install two packet manipulation rules. One rule applies to packets going from OF1 to OF2 and performs the following address translations (left side represents matched fields whereas right side represents how they are rewritten): **src**[pbl_IP:pbl_PORT], **dst**[pbl_IP:pbl_PORT]$\rightarrow$ **src**[pvt_IP:pvt_PORT], **dst**[pvt_IP:pvt_PORT] (note that **src**[pbl_PORT]=**src**[pvt_PORT]). This rule restores the original private source and destination IP/port of a packet when it reaches Customer2, so that the packet can correctly reach the destination host and any source-based traffic engineering policies can be applied. The other rule applies to response packets for the same TCP/UDP connection that go from OF2 to OF1, and accomplishes the opposite translations. Next, a BarrierRequest is issued by ISP2's SDN-controller, which waits for OF2 to confirm the installation of the above rules using a BarrierReply. At this point, ISP2's SDN-controller replies to ISP1's SDN-controller with a Map_Reply message, informing that host **dst**[pvt_IP:pvt_PORT] can be reached by sending packets to **dst**[pbl_IP:pbl_PORT]. ISP1's SDN-controller sends FlowModifications to OF1 to install the following rules affecting packets going from OF1 to OF2: **src**[pvt_IP:pvt_PORT], **dst**[pvt_IP:pvt_PORT]$\rightarrow$ **src**[pbl_IP:pbl_PORT], **dst**[pbl_IP:pbl_PORT], and similar rules for packets from OF2 to OF1. Finally, the installed rules are applied to the packet buffered at OF1, which is eventually forwarded: since all its IP addresses are public, it can successfully traverse the backbones of ISP1 and ISP2 (and, possibly, the Internet) to reach OF2, where the original source and destination addresses are restored.

7.7 Takeaway

By relying on the SDN architecture and proving both a configuration language and a set of primitives, we built a framework that is able to make the process of creating federated networks and subscribing to a federated VPN service simple and straightforward. Our framework aims at reducing the effort of providing federated VPN services, as well as simplifying the creation of a federated networks. Recalling the main challenges of today's federated networks introduced in Sec. 7.3, we now summarize how we solve those issues.

Management Problems – Avoiding additional interconnection points, each federated providers is still able to manage its network as it prefers, without any constraints in terms of collaboration with other federated partners. Our choice to delegate to the SDN-controllers the task of handling the federation and every federated service issued relying on such a network allows us to be independent from any kind of collaboration, having benefits for many operations, e.g. monitoring.

Also, agreeing on remuneration coming for the federated VPN service is very simple, since that traffic is easily recognizable starting from the IP addresses used during the translation phase. A strong point in favor of such a choice is that each provider acts independently from each other in order to decide which public IP addresses are used for that purpose. Remember that such choices are exchanged among federated ISPs at the beginning, allowing them to be aware about which traffic belongs to the federation.

Technological Differences Problems – As our framework does not require any *ad-hoc* place to interconnect, except the IXP that has a well defined interface for exchanging information (e.g. BGP protocol), each provider can use any routing protocol or transmission technology without coordinating with other federated partners. Also, the SDN architecture plays a key role. Indeed, being able to take centralized decision and sharing them among SDN-controllers exploiting a dedicated communication channel allows us to easily reach interoperability. Also, the choice of using NAT strategies to realize VPNs allows the SDN-enabled switches to issue IP packets, making the traffic forwarding completely independent from specific data plane protocols used in the backbone (e.g. MPLS).

Unified User View – We argue that our configuration language and primitives represent a solid way to provide a standard interface to access the federated VPN service. Furthermore, delegating the coordination activities to the SDN-controllers reduces the amount of time needed for provisioning the service. By relying on our framework, we argue that users perceive the federated VPN service as issued by a single provider, keeping hidden the collaboration among ISPs.

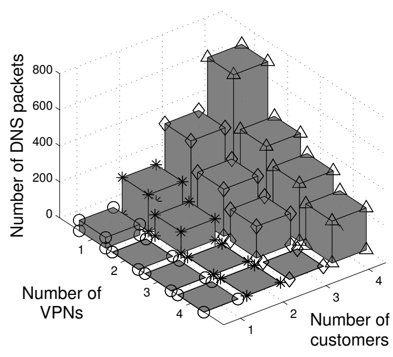
7.8 Evaluation

To validate the effectiveness of our framework, we implemented a prototype SDN-controller by relying on the Ryu framework [ryu17] and OpenFlow 1.3 [Ope14b]. We focus our evaluation activity on both control and data plane, analyzing how many messages (OpenFlow and DNS) are exchanged in the federated network and which is the impact on each OpenFlow-enabled switch of setting up a federated network. We run our experiments on SDNetkit [ML$^+$17], an SDN-enabled enanchement of Netkit [net17], a widely used network emulator. Within SDNetkit, we used BIND [bin17] to implement name server functionalities and OpenVSwitch [ovs15] to implement OpenFlow devices.

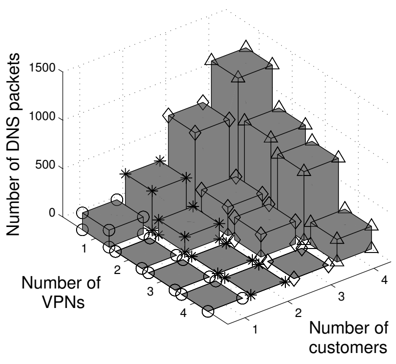
We run our SDN-controller on topologies reflecting the reference scenario depicted in Sec. 7.4. Our implementation is composed by three main components: 1) *Primitive Handler*, which handles primitives sent by customers; 2) *DNS Handler*, which handles DNS messages; and 3) *Routing Handler*, which computes the routing. Those components cooperate in order to allow customers to quickly and easily join a federated VPN and exchange traffic with each other participant of the VPN.

In our experiments, we focus on considering three different coordinates: 1. number of ISPs in the federated network, 2. number of customers per ISP, and 3. number of VPNs in the federated network. We run several simulations on different topologies, that are built according to the following criteria. First, we build a federated network consisting of two ISPs. In such a federated network, we connect to each ISP a number of customers varying in the range $[1, 4]$. Then, we set up a number of VPNs varying in the range $[1, 4]$. Also, we assume that a customer can be part of a single VPN. Hence, the number of VPNs has to be determined according to the number of customers per ISP (e.g. with a single customer per ISP, we cannot create more than one VPNs, with two customer we can set up at most two VPNs, and so on). Each customer consists of a host and a local name server, authority for that host. Second, we did the same in a federated network consisting of three ISPs. During each simulation, we perform DNS resolution and standard ping among any pair of hosts belonging to the same VPN, in order to issue the maximum number of DNS queries. We now briefly describe which is the impact of our SDN-controller on both control and data plane.

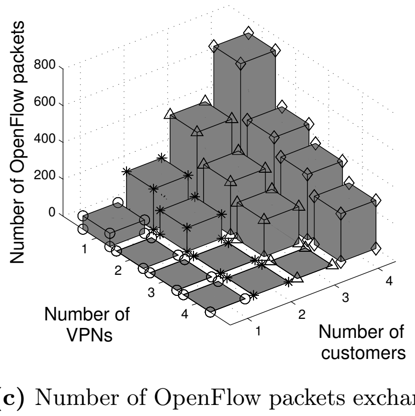
Control plane impact – To evaluate the impact of our implementation on the control plane, we measure the amount of DNS and OpenFlow packets exchanged in the network in order to allow a source (e.g. H1 residing in Customer1) to exchange traffic with a destination (e.g. H2 residing in Customer2). To



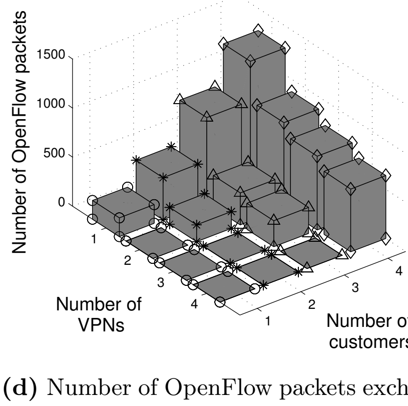
(a) Number of DNS packets exchange in the *Full configuration* scenario.



(b) Number of DNS packets exchange in the *Partial configuration* scenario.

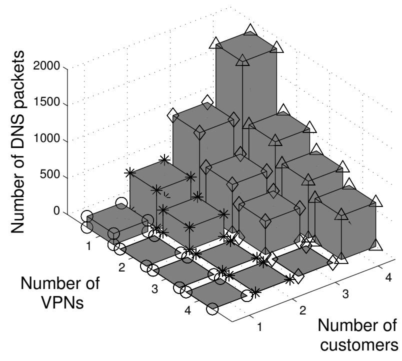


(c) Number of OpenFlow packets exchange in the *Full configuration* scenario.

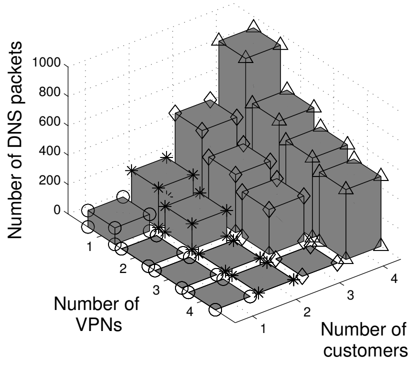


(d) Number of OpenFlow packets exchange in the *Partial configuration* scenario.

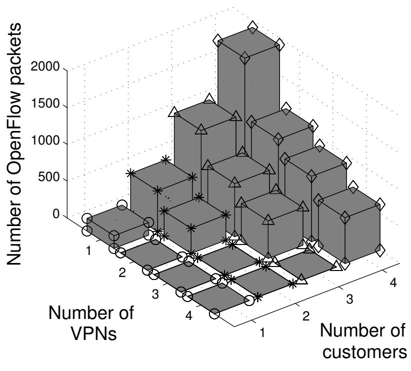
Figure 7.4: Control plane impact in a federated network consisting of two ISPs.



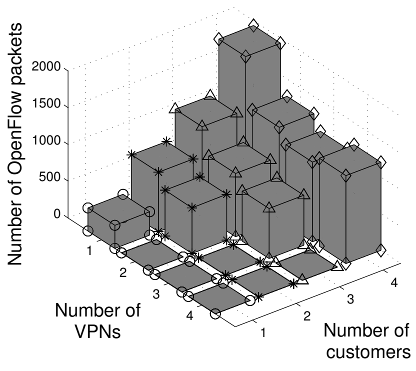
(a) Number of DNS packets exchange in the *Full configuration* scenario.



(b) Number of DNS packets exchange in the *Partial configuration* scenario.

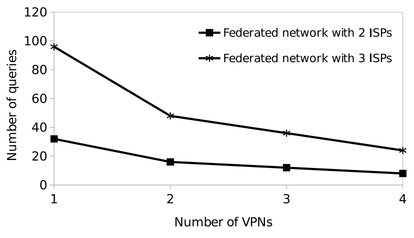


(c) Number of OpenFlow packets exchange in the *Full configuration* scenario.

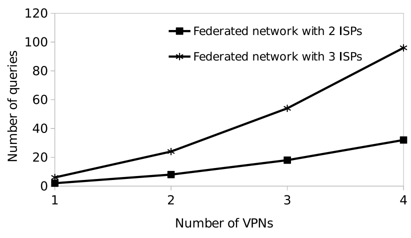


(d) Number of OpenFlow packets exchange in the *Partial configuration* scenario.

Figure 7.5: Control plane impact in a federated network consisting of three ISPs.



(a) Federated network with four customers per ISP.



(b) Federated network with a single VPN.

Figure 7.6: Number of queries in the federated network.

accomplish such an evaluation, we count the number of DNS and OpenFlow packets on each interface of each OpenFlow-enabled switch in the network, namely OF1, OF2, and OF3. With respect to the OpenFlow packets, we point out that we only consider PacketIn, PacketOut, and FlowModification messages, since our implementation affects those types of messages (e.g. OpenFlow handshake and keepalive messages are independent by any controller implementations). Also, we consider both *Full* and *Partial configuration* scenarios, as discussed in Sec. 7.6.

Fig. 7.4 shows the total number of DNS (Figg. 7.4(a) and 7.4(b)) and OpenFlow (Figg. 7.4(c) and 7.4(d)) messages exchanged in a federated network with two ISPs. We observe that the number of messages (both DNS and OpenFlow) grows with respect to the number of customers connected to each ISP, while it decreases when the number of VPNs increase. We ascribe this behavior to the fact that the number of messages strictly depends on the number of queries in the network. Indeed, if more queries are performed by many sources, more DNS packets are issued in order to get the IP address of each desired destination. Regardless from which scenario (*Full* or *Partial configuration*) we are considering, our SDN-controller has to interact with such DNS packets, implying an increasing number of OpenFlow messages. Those considerations are validated by looking at Fig. 7.6. Indeed, we observe that the number of queries grows with respect to the number of customers (Fig. 7.6(b)), while decreases with respect to the number of VPNs (Fig. 7.6(a)). This is due to the fact that increasing the number of VPNs means reducing the number of destinations per VPN, reducing the total number of queries, and - consequently - the total number of messages.

The same considerations are valid in the case of a federated network consisting of three ISPs. The total number of DNS and OpenFlow messages exchanged in the network is depicted in Fig. 7.5. Of course, in this case we observe a greater number of messages (almost 2000). Such an increase with respect to Fig. 7.4 has to be ascribed to the fact that adding a new ISP in the federated networks results in increasing the number of customers belonging to the federated network itself and, consequently, the total number of destinations. Since we already discussed how the number of destinations affects the number of queries and which is their impact on the total number of messages exchanged in the network, those results are perfectly aligned with what we expect, also considering results shown in Fig. 7.6. It is worth to observe that the scalability of our framework is the same of the DNS service. Indeed, we do not add any DNS messages to the name resolution process, as in the *Partial configuration* scenario. Rather, in the *Full configuration* scenario, we prevent several DNS messages (e.g. DNS messages directed to the root name servers) to be forwarded in the network. On one hand, having the IP address of a local name server in the configuration saves a lot of DNS and OpenFlow messages. On the other hand, retreiving that information from the DNS resolution process is a plus from a configuration point of view (see Sec. 7.6), but it represents a cost considering the number of exchanged messages, that is – in any cases – the same of any DNS service.
Data plane impact – By default, our prototype SDN-controller installs two rules: one for ARP packets and another one for DNS packets. Indeed, all traffic belonging to those classes needs to be processed by the SDN-controller in order to allow the SDN-enabled switches to exchange traffic with the neighbor routers (ARP packets) and to allow the *DNS Handler* to properly steer the name resolution process (DNS packets).
Primitive Handler – This component has in charge the task of processing primitives sent by customers. After analyzing the content of each primitive, it writes proper information in the SDN-controller configuration, as shown in Sec. 7.5. Basically, this component does not have any impact on the data plane.
DNS Handler – This component is in charge of steering the name resolution process. After resolving a domain name, the *DNS Handler* installs a rule to send IP traffic toward the destination to the controller. This rule is needed, since it triggers the *Routing Handler*, whose task is to act as shown in Sec. 7.6. Note that, for each possible destination, one rule is needed, resulting in a number of rules that is linear with respect to the number of destinations in the federated VPN.
Routing Handler – Routes in the networks are computed by this component. In particular, for each pair of end-hosts, it installs two rules. The first handles

the traffic directed to the destination, whereas the second allows the traffic to come back from the destination to the source. Thus, the number of the rules is quadratic with respect to all possible combinations among end-hosts. Since this situation is not so common in computer networks (destinations are less than sources), the number of rules is linear with respect to the number of pair ⟨source,destination⟩. Optimizations might be carried out attempting to reduce that amount of rules.

7.9 Conclusions and Future Work

In this work, we propose a framework enabling fast creation of federated networks. We show that the today's federated network architecture can be simplified by adopting SDN. Also, we demonstrate that our framework does not impact any existing configuration, as well as any existing architecture. It does not require architectural changes, except the adoption of SDN-controllers, that is a reasonable assumption.

As research perspectives, we intend to go deeply in improving our current implementation, providing a more complete software enabling federations to use it in order to issue federated services. We believe that in a world where IPv4 address exhaustion is being a problem – also due to the slow IPv6 adoption [goo17, rip17] – our solution represents a valid alternative that allows ISPs to provide value-added services to their customers, without introducing any scalability issues.

Chapter 8

When details make the difference: How to handle ARP in a Software Defined Network

The Address Resolution Protocol (ARP) enables communication between IP-speaking nodes in a local network by reconstructing the hardware (MAC) address associated with the IP address of an interface. This is not needed in a Software-Defined Network (SDN), because each device can forward packets without the need to learn this association. We tackle the interoperability problem arising between standard network devices (end systems, routers), that rely on ARP, and SDN datapaths, that do not handle ARP packets natively. In particular, we propose a general approach to handle ARP in a SDN, that is applicable in several network scenarios, is transparent for existing devices, and can coexist with any packet forwarding logic implemented in the controller. Our approach reduces ARP traffic by confining it to the edge of SDNs and requires a minimal set of flow entries in the datapaths. We argument about its applicability and confirm it with experiments performed on SDN datapaths from a range of different vendors.

8.1 Introduction

Software Defined Networking (SDN) is a recently affirmed architectural approach to computer networking. It physically separates the control plane of a network device, implemented by a dedicated software (*controller*), from the data plane, which is realized by switches (*datapaths*) that are only capable of forwarding packets and performing basic manipulations. According to OpenFlow [Ope15b], one of the most adopted specifications of SDN, each datapath forwards packets according to match-action rules (*flow entries*) contained in a *flow table*: the match condition selects incoming packets based on their headers, whereas the action can forward, manipulate, or drop matched packets. When a datapath has no matching rule to handle a packet, it sends a message to the controller, asking for the action to undertake. The controller can either choose to install new rules in the datapath's flow table or ask the datapath to send (forward or generate) a single packet.

Like many other groundbreaking approaches, SDN is conceived to interoperate with existing technologies, provided that appropriate software in the controller allows SDN datapaths to talk with standard network devices. This also applies to the Address Resolution Protocol (ARP) [Plu82]: in fact, in an IP network ARP is required to reconstruct the association between a packet's destination IP address and the corresponding hardware (MAC) address, so that the subsequent IP node along a network path can correctly receive and process the packet. On the other hand, SDN datapaths do not need to learn this association (consequently, they do not need ARP), because their hardware is designed to accept, process, and forward packets based only on the headers appearing in match conditions of OpenFlow rules. Filling this gap is especially important considering that, while many operators have already accomplished or are at least considering a migration to SDN-based infrastructures and most vendors have refreshed their product offers with OpenFlow-compliant devices, the vast majority of end systems will continue to use IP for a reasonably long time. Some SDN controller frameworks already implement basic handling of ARP packets. However, these implementations often just reproduce the standard behavior of the ARP protocol (possibly including the ARP cache) with OpenFlow, thus inheriting the associated scalability issues.

We propose a general approach to handle ARP traffic in a network that consists of a mixture of legacy and SDN-enabled devices. In particular, we design an ARP component for an SDN controller that has the following features: it reduces ARP traffic by confining it to the edge of a SDN, thus limiting its impact and improving scalability; it can be applied in several scenarios, including of course partial SDN deployments as well as networks involving multiple IP subnets; it is transparent for existing IP devices; it can coexist with any additional packet forwarding logic implemented in the controller; and it requires a minimal set of rules on the datapaths. To our knowledge, this is also the first time that a general method to handle ARP packets in a SDN is specified in detail.

The chapter is organized as follows. In Section 8.2 we review the related work. In Section 8.3 we discuss current SDN-based ARP implementations and describe our design for an ARP-aware SDN controller. In Section 8.4 we argument on the applicability of our approach, confirming it by experimental tests. Conclusions and lines for future work are in Section 8.5.

8.2 Related Work

The OpenFlow specification gives no guidelines about handling ARP traffic in a SDN, therefore various ad-hoc approaches have been proposed in the literature. A recently published patent [TS14] claims a method for implementing an OpenFlow controller that is able to handle ARP requests and cache IP-MAC associations. This controller basically re-implements the traditional ARP traffic handling, with the associated scalability issues and without taking advantage of SDN. The authors of [XLZ15] and [KAK14] propose methods to limit the distribution of broadcast packets by propagating them along a pre-computed spanning tree or according to host location information learned by the controller. Although these methods are applied also to ARP requests, such packets still need to be carried across the network. In [CKL15] the authors propose to maintain on the controller a complete IP-to-MAC table for all the virtual machines in a data center: the controller can therefore reply to ARP requests without forwarding them to the target node. However, this approach is only applicable in a data center network, where the IP-to-MAC table is known from static management information and is updated by accessing a virtual machine state manager. Similar approaches are proposed in [MNG14, KS13]. The Open Networking Foundation, which releases the OpenFlow specification, started a project to foster migration of existing network services to SDN, but the documents produced so far (e.g., [Ope14a]) describe general guidelines and omit most technological details. Most notably, all the above contributions focus on a well-defined scope of application, consisting of a single broadcast domain, namely a single IP subnet. Our approach overcomes this limit, as it can be applied in a SDN spanning multiple IP subnets, and is also compatible with additional functions (e.g., ICMPv6 neighbor discovery) executed by the end systems.

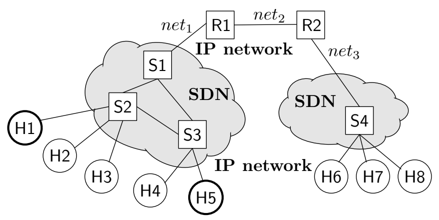
8.3 Handling ARP in a Software Defined Network

Before describing our general architecture for an ARP-aware SDN controller, we discuss some simple approaches to handle ARP in a SDN with currently available technologies.

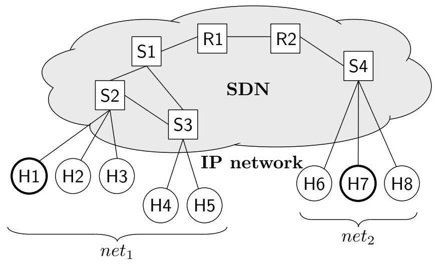
The simplest solution consists in proactively installing on the datapaths a flow entry that instructs them to process ARP packets using the standard networking stack. However, this approach has some drawbacks: it requires datapaths to support the NORMAL reserved output port [Ope15b] as a flow entry action, which the OpenFlow specification declares as optional; it requires broadcasting ARP traffic over a potentially large SDN; and it introduces a dependence on legacy technologies, which pure OpenFlow datapaths cannot satisfy. Some SDN controller frameworks [ryu17, flo15, et 15, MRF+13] offer readily usable basic ARP handling features. Since learning IP-MAC address associations and learning the location of end systems are closely related functions, both of them are usually implemented inside a backward learning module of the controller: this module looks at the source MAC address `sha` of packets received by a datapath *dpid* and builds a correspondence between `sha`, the packet's source IP (if applicable), *dpid*, and the port from which the packet was received by *dpid*. In order to better describe the ARP functions offered by such a module, we refer to the scenarios in Fig. 8.1. 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 switches), and labels net_i represent IP subnets. Nodes enclosed in gray clouds are SDN-enabled datapaths, and we assume that they are managed by a single logical controller, while the other nodes represent IP devices.

Most ARP handling approaches in the literature, as well as implementations in the controllers, consider the scenario in Fig. 8.1a, in which multiple independent SDNs exist and each of them handles ARP traffic only for a single IP subnet. In particular, in this setting most backward learning modules handle ARP in a way similar to [TS14]. On the other hand, Fig. 8.1b illustrates a scenario in which a single SDN supports traffic exchange among multiple IP subnets, a commonly overlooked scenario which is supported by our approach.

Using Fig. 8.1a as a reference, we first briefly describe the ARP handling mechanism available in Ryu [ryu17], a controller framework with a wide and very active community of users and developers (other frameworks adopt a similar approach). Assume that the flow tables of all datapaths are empty, and suppose H1 sends an ARP request asking for the MAC address of H5. Upon receiving the packet, datapath S2 forwards it to the controller using 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 8.1: Our reference scenarios, comprising switches (S1, S2, ...), routers (R1, R2), and hosts (H1, H2, ...). Devices enclosed in gray clouds are SDN-enabled.

OpenFlow message. The controller then performs the backward learning and instructs S2 to flood the packet (i.e., send it out of all the ports), without installing flow entries. The packet reaches S1 and S3, which again are instructed by the controller to flood it. The process continues until the packet reaches H5, which sends an ARP reply to S3. Since S3 has no flow entries that match the packet, it sends it to the controller. The latter performs the backward learning and, based on the location of H1, which it has learned along the path from H1 to H5, it installs a flow entry on S3 to send the packet out of a single port. In the same way, flow entries are installed on other datapaths to forward the ARP reply along a single path computed by the controller. Any subsequent unicast ARP requests issued by H1 for H5 will also be forwarded along a single path, based on the location of H5 that the controller has learned and on the flow entries it correspondingly installed.

This mechanism has several shortcomings. First, ARP packets are transported all the way to the target host, thus limiting the extent of the SDN to a single IP subnet: a packet directed to a different subnet must either reach an IP router and then enter another SDN (like in Fig. 8.1a), or reach an SDN datapath that accomplishes the tasks of a router. Second, ARP requests are broadcast over a potentially large topology, introducing a forwarding overhead. To mitigate this, the backward learning module of some controller frameworks [et 15] also learns IP-MAC associations, so that the controller can immediately reply to ARP requests for known hosts, similarly to an ARP cache. However, this still works for a single subnet. Third, the illustrated mechanism does not work in the presence of loops, because broadcast packets would cycle forever. Some frameworks [et 15] prevent this by computing a spanning tree and blocking selected ports, but such ports may then be unable to send any packets. Our approach is free from all these issues because it keeps ARP traffic confined to the edge of the SDN.

An SDN Controller Component for Handling ARP Traffic

We now illustrate our proposed architecture for an SDN controller that supports ARP traffic exchange with legacy IP nodes in a variety of scenarios. In order to better isolate its functions, the controller consists of the following modules.

Backward learning – As described above, it is triggered for every packet that a datapath forwards to the controller, and it maintains an association between the MAC address of a node (host or router) and the datapath and port it is connected to.

ARP processing – It handles every ARP packet received by edge datapaths, namely those that are directly connected to IP nodes. To support this module, a flow entry must be installed on edge datapaths instructing them to forward all ARP packets to the controller. This module accomplishes two tasks: the first is to learn associations between IP and MAC addresses based on the information carried by ARP packets (both requests and replies). The second is to immediately respond to ARP requests from IP nodes by forging ARP replies. Since SDN datapaths do not need to know the recipient’s MAC address to forward a packet, the forged ARP replies specify a fixed fake MAC address M: this is enough to make a host start sending data packets, while the recipient’s MAC resolution is accomplished in background by the discovery module. M can be chosen arbitrarily, and will never be seen by any other IP nodes except the one that sent the ARP request. The controller’s MAC address or a reserved MAC address such as `ff:ff:ff:ff:ff:fe` are reasonable conflict-free choices.

This ARP responding functionality confines to edge datapaths all the ARP requests coming from IP-speaking nodes.

Discovery – 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 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. 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.

Routing – Once ARP exchanges are completed and a source IP node starts sending data packets to a destination IP node, this module computes a path from the datapath to which the source is connected to the datapath to which the destination is connected. Appropriate packet forwarding rules are installed on all the involved datapaths. The routing path is based on the network topology (which is automatically reconstructed by the controller) and on information acquired by the discovery module, and is selected based on arbitrary routing policies or any other forwarding logic implemented in the controller.

With this approach, ARP packets are confined at the edge of the SDN and traffic is correctly forwarded even if the source and destination nodes are in different IP subnets.

Example of ARP Traffic Processing

We now run through a complete example of how ARP packets are handled by our controller, using the diagram in Fig. 8.2 as a guide to follow the sequence of message exchanges. We consider Fig. 8.1b as a reference scenario, in which R1 and R2 are SDN switches representing the separation between multiple subnets. At a first stage, when all the flow tables are empty and before any traffic is exchanged, flow entries are installed on all the datapaths (gray band **A** in Fig. 8.2), instructing them to send ARP packets to the controller. Now,

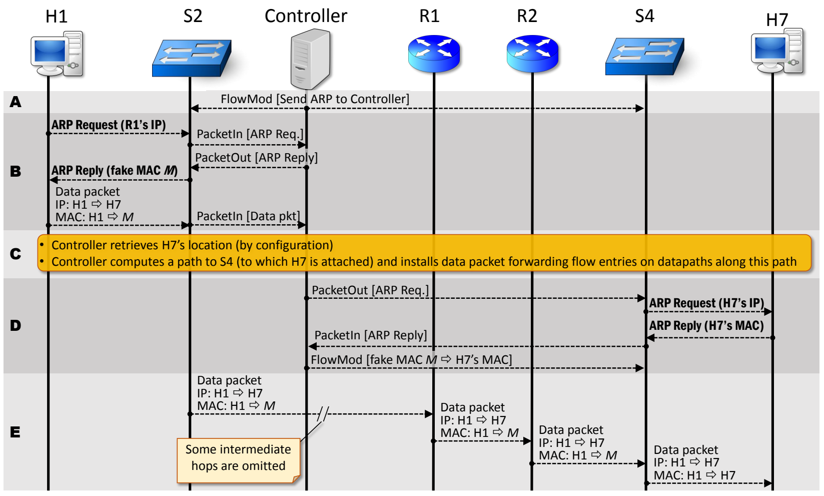

H1
S2
Controller
R1
R2
S4
H7
A
FlowMod [Send ARP to Controller]
B
ARP Request (R1's IP)
PacketIn [ARP Req.]
PacketOut [ARP Reply]
ARP Reply (fake MAC *M*)
Data packet
IP: H1 ⇨ H7
MAC: H1 ⇨ *M*
PacketIn [Data pkt]
C
• Controller retrieves H7's location (by configuration)
• Controller computes a path to S4 (to which H7 is attached) and installs data packet forwarding flow entries on datapaths along this path
D
PacketOut [ARP Req.]
ARP Request (H7's IP)
ARP Reply (H7's MAC)
PacketIn [ARP Reply]
FlowMod [fake MAC *M* ⇨ H7's MAC]
E
Data packet
IP: H1 ⇨ H7
MAC: H1 ⇨ *M*
Some intermediate hops are omitted
Data packet
IP: H1 ⇨ H7
MAC: H1 ⇨ *M*
Data packet
IP: H1 ⇨ H7
MAC: H1 ⇨ *M*
Data packet
IP: H1 ⇨ H7
MAC: H1 ⇨ H7

Figure 8.2: Sequence of messages exchanged in order to support ARP traffic in the SDN in Fig. 8.1b using our approach. ARP messages, indicated in bold face, are evidently confined to the edge of the SDN.

suppose host H1 in subnet net_1 wants to send traffic to host H7 in subnet net_2. H1 sends an ARP request asking for the MAC address of its default gateway R1 (gray band **B** in Fig. 8.2), which allows the controller to record H1's location and IP-MAC association. H1's ARP request is not forwarded any further, since the ARP processing module in the controller instructs S2 to immediately send back an ARP reply specifying the fake MAC M. At this point H1 can start sending traffic. Once the first IP packet from H1 is received by S2, the controller can determine that the destination IP of this packet is H7's (note that this piece of information could not be derived from ARP packets). Based on configuration information, the controller knows the location of H7's subnet net_2. Therefore, it can install on selected datapaths flow entries that carry packets from H1 to S4 along a path that is computed by the Routing module (gray band **C** in Fig. 8.2). In the meantime, the controller also asks S4 to broadcast an ARP request asking for H7's MAC address (gray band **D** in Fig. 8.2). Once H7 replies with its physical address, the Discovery module records this piece of information and the last forwarding rule is installed on S4: this rule also replaces the fake MAC M in packets traversing S4 with H7's actual MAC, thus supporting delivery of traffic to H7 (gray band **E** in Fig. 8.2). For convenience, the controller may also install rules to support traffic flowing in the opposite direction, from H7 to H1. We point out that only datapaths S2 and S4 are reached by ARP traffic.

8.4 Applicability: Considerations and Tests

Application Scenarios – We showed that our controller can handle ARP traffic in the presence of a single IP subnet (Fig. 8.1a) or multiple IP subnets (Fig. 8.1b). As an easy extension, our approach also works with partial SDN deployments. For example, if in the scenario of Fig. 8.1a host H1 wanted to communicate with host H7, we could still apply our ARP handling approach independently for each SDN area: this would require every controller instance to know the address of a proper next hop IP router (e.g., R1's), which however is needed irrespective of whether our ARP handling approach is used or not. As an ultimate adaptation, even if the traffic needed to cross several Autonomous Systems (ASes) in the Internet, each AS could run an independent instance of our controller to handle ARP, regardless of whether neighboring ASes use SDN or not and without the need of any communications among controllers of different ASes.

As a side note, using a fixed fake MAC address M in the presence of legacy switches is not a problem either, because they will just forward packets on the interface where the latest ARP reply (carrying the fake MAC) has been received.

Scalability – We argue that our approach scales well with the size of the network. In fact, our controller installs few rules on edge datapaths to operate: one for sending ARP packets to the controller and one for each destination IP address, which replaces the fake MAC address M with the actual recipient's MAC (current controllers usually need a pair of rules for every pair of communicating hosts). No rules for handling ARP are installed in any other datapaths, and no ARP packets are ever received by the controller from those datapaths (unlike other controllers). If flow table capacities are constrained, MAC rewriting rules can be moved from edge datapaths to any other datapaths along the path between two hosts. ARP broadcasts are still required to locate IP nodes, but their scope can be limited using approaches such as [XLZ15], [KAK14], or even [ICA+11].

The scalability of our approach can also be assessed in terms of amount of ARP traffic it allows to save. In a traditional IP network, the number of links traversed by broadcast ARP requests is the total count of edges of all the traversed LANs:

$$L_{\mathrm{B}}^{\mathrm{IP}} = \sum\nolimits_{i=1}^{N} |E_i| = \sum\nolimits_{i=1}^{N} |V_i|^2 \tag{8.1}$$

where V_i and E_i are the sets of nodes and links belonging to LAN i, and N is the number of LANs in which subsequent ARP requests are triggered until the destination host is reached. On the other hand, with our approach the number of links traversed by broadcast ARP requests decreases to:

$$L_{\mathrm{B}}^{\mathrm{SDN}} = 1 + |H_d| \tag{8.2}$$

where $H_d \subset V_d$ is the set of hosts in the LAN that contains the destination node d. Regardless of the LANs traversed in computing $L_{\mathrm{B}}^{\mathrm{IP}}$, it is obviously always $L_{\mathrm{B}}^{\mathrm{IP}} > L_{\mathrm{B}}^{\mathrm{SDN}}$.

Even more easily, in a traditional IP network unicast ARP packets (including both ARP replies and gratuitous ARP requests) will traverse at least the following number of links:

$$L_{\mathrm{U}}^{\mathrm{IP}} = \sum\nolimits_{i=1}^{N} |P_i| \tag{8.3}$$

where $P_i \subseteq V_i$ is a set of vertices along the shortest path from the source of the ARP packet to its destination. On the other hand, using our approach, just 2 links are involved in such a traffic (i.e., the links connecting the source and destination hosts to the respective SDN switches), and of course $L_{\mathrm{U}}^{\mathrm{IP}} \geq 2$.

Traffic Engineering and Network Dynamics – The fake MAC M appears as the destination MAC in all data packets sent by IP nodes. Therefore, by replying with different fake addresses to ARP requests coming from different IP nodes, it is possible to exploit M as a label that drives traffic engineering decisions along a packet's routing path. Moreover, mechanisms such as ICMPv6 NUD can be exploited by the controller to keep IP-MAC associations and node location information up-to-date as nodes are connected or disconnected.

Experimental Testbed and Results

We implemented a prototype SDN controller according to the design in Section 8.3 using the Ryu framework [ryu17]. We then ran experiments using hardware switches from 3 different vendors with about a decade of experience in manufacturing network devices (an NDA prevents us from explicitly mentioning them). The switches were compliant with OpenFlow 1.3. Our experiments were aimed at assessing the compatibility of our controller with the switches as well as the interoperability between SDN switches and IP hosts. We therefore considered a simple linear topology: host1 — dp1 — dp2 — host2. We instantiated dp1 and dp2 with all the possible combinations of the available switches. host1 and host2 were Ubuntu 14.10 machines with IP addresses in different subnets (`10.0.0.0/24` and `20.0.0.0/24`). Each host used an arbitrary IP address in its own subnet as default gateway. Controller-switch communication was realized out of band. We used `ff:ff:ff:ff:fc:ac` as fake MAC address M.

We verified the correct handling of ARP traffic by the controller by running a simple ping between host1 and host2 immediately after setting up the network (experiments involving other protocols are omitted since our approach only addresses handling of ARP traffic). In our tests, we successfully checked that:1) ARP requests generated by host1 were sent to the controller; 2) datapaths correctly delivered forged ARP replies containing the fake MAC address M to the requesting host; 3) ARP caches of the hosts were populated with the fake MAC; 4) ARP packets were kept confined to the edge of the SDN; 5) datapaths correctly installed flow entries for the forwarding of ICMP traffic, and these entries were correctly matched; 6) datapaths correctly installed entries to replace the fake MAC address with the recipient's one, and the destination MAC addresses of ICMP packets were actually rewritten as expected; 7) hosts exchanged ICMP packets.

Based on these 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 and transparent for the end hosts.

8.5 Conclusions and Open Problems

We describe the design of an SDN controller that handles exchange of ARP packets in the presence of SDN switches and IP devices. Our approach confines such packets to the edge of the SDN, works in a variety of scenarios also involving multiple IP subnets, is transparent for IP nodes, and is compatible with any packet forwarding logic already realized by the SDN controller. Functional tests on a range of SDN switches from different vendors confirm the viability of our approach.

Consolidating our proposed design requires further investigation on several aspects. As a first step, our prototype controller implementation can be improved and further tests can be performed on more complex network topologies. As discussed in Section 8.4, fake MAC addresses could be exploited to distinguish traffic classes and route them distinctly throughout the SDN. Our approach could be extended to support the IPv6 counterpart of ARP (i.e., neighbor discovery), as well as other ICMPv6 features (e.g., router renumbering). Interoperability with other well-known mechanisms used in IP networks poses interesting challenges: for example, path MTU discovery is difficult to implement with plain OpenFlow. Some network configuration information (e.g., the IP subnets attached to each datapath) could be derived from routing protocols (e.g., OSPF), that may already be in use to support in-band communication between the switches and the controller.

Chapter 9

On the Practical Applicability of SDN Research

Software-Defined Networking (SDN) is an evolving, yet 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 and to the associated challenges, a whole stream of scientific literature is devoted to SDN and its most widely adopted incarnation, OpenFlow. However, little attention has been put in verifying the applicability of the proposed approaches on current hardware: we argue this is as a considerable overlook.

We therefore bring the following contributions: i) a critical review of key contributions in the field of SDN in terms of applicability issues stemming from publicly documented limitations of OpenFlow implementations; ii) a methodology for testing the readiness of devices for operating in a Software-Defined network, which comprises an OpenFlow compliance test as well as custom tests; iii) an application of the methodology to a range of devices from different vendors, which unveils lots of anomalous behaviors that network operators should consider when selecting SDN devices.

9.1 Introduction

Software-Defined Networking (SDN) has been the future of network architectures for a few years, promising unmatched infrastructure scalability, supporting improved flexibility of management, ensuring vendor independence, and boosting the design of novel network services. Now that it is the present, it continuously feeds multiple research areas with unprecedented challenges and it is adopted by almost all device vendors.

In the SDN model, the functional logic of a network device, called *datapath*, is realized by a piece of software called *controller*, while the device itself only performs packet forwarding based on a set of match-action conditions called *flow entries*. When a datapath does not know how to handle a packet, it submits it to the controller. The controller may either ask the datapath to emit a copy of the packet out of a specific interface, or install a new flow entry in the datapath's *flow table* that instructs the datapath about how to independently handle future packets belonging to the same *flow*: such an entry watches certain bits of the header fields of received packets while applying wildcards on other bits, and it executes an action on matching packets (e.g., forward out of a port, drop). The most widely adopted specification of SDN is OpenFlow [Ope15b], which also defines a protocol for controller-datapath communication.

Despite the fervent activity in the scientific community on devising novel network architectures and services that take advantage of SDN, most papers validate their proposals on ad-hoc testbeds, and little attention has been devoted to determining the practical applicability of these approaches using currently available devices. On the other hand, even if OpenFlow is now somewhat mature, vendors seem to lag behind in terms of functionalities supported on their devices. Without precise indications on which features are supported, network administrators interested in switching to SDN may have a hard time trying to find the selection of SDN-enabled devices that best fits their needs.

Preserving this dual (scientific and technological) perspective, in this work we bring the following contributions:1) we contrast a selection of the most important contributions in the literature about SDN with publicly available documentation from device vendors, highlighting the consequent applicability issues that scientific contributions may incur; 2) we define a methodology for testing the readiness of a device to operate in an SDN-based infrastructure, combining existing OpenFlow conformance test tools with other custom tests; 3) we draw a picture of the current status of OpenFlow implementations by applying our methodology to commercially available devices manufactured by 7 major vendors, and compile a catalog of observed anomalies that confirm the above mentioned applicability issues and can serve as a useful reference for prospective SDN adopters when selecting network devices. We believe that the ensemble consisting of the documented implementation limitations and of the operational issues we revealed in our tests can be a key element for ensuring practical applicability of future research results.

The rest of the chapter is organized as follows. In Section 9.2 we review vendor-declared limitations and discuss their impact on the applicability of approaches in the SDN literature. Section 9.3 summarizes the basic concepts of OpenFlow. In Section 9.4 we introduce our methodology for determining the readiness of a device to operate in an SDN-enabled scenario. The results of the application of our methodology to a range of commercially available datapaths are documented in Section 9.5, while the observed anomalies are illustrated in Section 9.6. Section 9.7 reviews related work in the field of SDN devices testing, while Section 9.8 draws conclusions and ideas for future extensions of our contributions.

9.2 A Review of SDN Literature with an Eye on User Manuals

At the time of writing, all the major vendors have been including SDN-enabled switches in their device offer for quite a long time, usually adhering to the OpenFlow specification. However, despite the fact that OpenFlow has been around for at least 5 years, vendors still explicitly declare the existence of limitations in their implementations, because of the involved technical difficulties and because new versions of the specification are released at a somewhat rapid pace. To the extent of our knowledge, the level of awareness of such limitations in the literature about SDN is still modest. We argue that the lack of a deployment on real devices induced several authors to overlook many restrictions imposed by vendors that can have a non-negligible impact on the applicability of some groundbreaking approaches. In this section we review these restrictions based on public documentation and discuss their impact on some of the most relevant contributions about SDN.

We identified a selection of top OpenFlow switch vendors that declare implementation limitations in user manuals. In random order, they are: HP [Hew14], Dell [Del15], Brocade [Bro15], Arista Networks [Ari15], and Extreme Networks [Ext15]. We highlight that this selection of vendors is not necessarily related with the selection of devices considered in the compliance tests described in Sections 9.5 and 9.6. We isolated the most relevant and frequently occurring limitations and associated them with the vendors declaring them. The results are in Table 9.1. The capacity of flow tables often depends on hardware constraints, but may be further restricted due to fixed partitioning schemes of the internal memory used to implement them. Some vendors support a single flow table, thus limiting scalability and making it more difficult to

Table 9.1: Common limitations in OpenFlow implementations and vendors declaring them.

Limitation	HP	Dell	Brocade	Arista Networks	Extreme Networks
Limits on the size of the flow tables		●	●		
Only a single flow table supported				●	●
Restrictions in the structure of match conditions		●	●		●
Lack of MPLS support	●		●	●	
OpenFlow and L2/L3 protocols cannot coexist on the same port			●	●	
Lack of support for the NORMAL reserved port			●		●

match a packet's inner headers (e.g., IP) after popping outer headers (e.g., MPLS). Most notably, match conditions are often restricted to comply with predefined patterns (e.g., matching on MAC addresses may not be permitted when considering ICMP packets) and memory consumption highly depends on their structure (e.g., only a limited number of flow entries that do not match the input port may be installed). 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 may be prohibited, and the NORMAL reserved port (used to process a packet using the traditional non-OpenFlow networking stack) may be unsupported: the lack of these two features poses significant limits on realizing in-band communication between the datapaths and the controller.

We now analyze the impact of the above limitations on contributions from the literature. First of all, there is a class of papers [KLRW13, KHK13, YRFW10, CMT$^+$11, IMS13, NSBT14] that addresses limitations in the capacity of TCAMs, where flow entries are typically stored. These papers aim at reducing the size of flow tables by replacing the flow entries installed on the datapaths with more compact, equivalent entries. However, these papers omit to consider both the restrictions imposed on the structure of match conditions (see, e.g., [Ext15]) and the additional space consumption caused by the usage of wildcards [Del15].

In more detail, [KLRW13] and [NSBT14] propose, respectively, a flow entry placement algorithm and a formulation of an optimization problem to compact flow entries, ignoring vendor-imposed dependencies between the matched fields: flow entries resulting from compaction may therefore be impossible to install on the datapaths. In [YRFW10, CMT$^+$11, IMS13] the authors propose methods for massively compressing flow tables using wildcards. However, the potential of these approaches is restricted by the fact that wildcards consume a lot of hardware resources, reducing the capacity of flow tables by one or two orders of magnitude (see, e.g., [Del15]). The flow table decomposition approach proposed in [KHK13] pursues the opposite strategy, because match conditions are manipulated to reduce the number of wildcard bits, however it is not immune from potential violations of vendor restrictions. Some of the above mentioned contributions cope with NP-hard theoretical problems: it would be interesting, although out of the scope of this work, to analyze whether the computational complexity of these problems is reduced by the introduction of vendor constraints.

Unlike the aforementioned works, in [NHL$^+$13] the authors evaluate their approach using hardware switches. They discuss the diverse memory requirements of exact matches as opposed to wildcards and, not surprisingly, suggest usage of the former whenever possible.

Another relevant class of papers [GVS$^+$14, LRVB15, DPSBM13] addresses the practical problems involved in deploying SDN in specific application scenarios. In [GVS$^+$14] 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 that are translated into flow entries. Some of these entries are used to forward traffic based on the destination MAC address, and they do not pose scalability issues (see, e.g., [Del15, Ext15]). Instead, flow entries that support control traffic match on layer-4 fields, implying a drastic reduction in the number of installable flow entries. Despite this limiting factor, this solution may still be considered deployable under the realistic assumption that an average-sized IXP does not have more than 500 participants. In [LRVB15] 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 heavy use of MPLS-specific operations for traffic forwarding. According to Table 9.1, this choice makes the approach compatible only with devices from selected manufacturers.

Both [GVS+14] and [LRVB15] 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 the possibility to run OpenFlow and layer-2 (e.g., Spanning Tree Protocol) or layer-3 protocols (e.g. OSPF) together on the same ports, further limiting the applicability of the proposed solutions. Even in a wireless mesh network scenario [DPSBM13] that proposes an OpenFlow-based architecture capable of reacting to network partitions, the authors rely on support for hybrid OpenFlow/traditional operation mode as well as the NORMAL port.

9.3 The OpenFlow Specification

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 [Cis]) 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 [Ope15b].

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. We already illustrated the basic operation of flow tables in Section 9.1, but we omitted a few additional features that are useful for our tests. 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. Although support for these counters is declared as optional, the outcome of the tests we executed on the datapaths (see Section 9.4) 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. 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, we only need to recall one feature: 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.

9.4 Device Testing Methodology

Limitations in vendor implementations, discussed in Section 9.2, are not the only aspect that must be taken into account for the deployment of novel SDN-based architectures. As described in Section 9.6, some devices may exhibit undocumented behaviors that can further impair practical adoption of SDN. In this section we define a device testing methodology that can be systematically applied to any datapaths. The methodology consists of two main phases: a deep test of compliance with the OpenFlow specification, and the verification of the availability and correct operation of several additional features that improve the maintainability or even enable the deployment of SDN-based network scenarios.

OpenFlow Compliance Test

As anticipated in Section 9.3, OpenFlow is a continuously evolving specification: since its introduction in 2009 up to the time of writing this work, at least 13 different versions of the specification have been published by the Open Networking Foundation. It is therefore expected that vendors can implement the latest specifications only after some time since their publication. While most devices are declared to comply with the earliest OpenFlow 1.0 specification, an increasingly larger number of vendors is adopting more recent versions. In particular, OpenFlow 1.1 is an important milestone, considering that it introduced the ability to define multiple flow tables and to define bitmasks on MAC and IP addresses, it improved handling of VLAN tags, and it added support for handling MPLS labels, a set of features that are important for most practical applications. In practice, most hardware manufacturers and controller developers left behind a gap in the sequence of implemented OpenFlow versions, skipping from OpenFlow 1.0 directly to OpenFlow 1.3. Therefore, we refer our compliance test to OpenFlow version 1.3 [Ope13b].

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. For the purpose of our evaluation we used the OpenFlow switch test tool included in the Ryu controller framework [Ryu15], shortly referred to as "Ryu test suite" in the following (other testing tools are reviewed in Section 9.7). During internal communications, some vendors confirmed that they also use Ryu to check the compliance of their devices. We used version 3.18 of Ryu, which includes as many as 991 different test cases that span the OpenFlow 1.3 specification. Running the Ryu test suite involves 3 different devices connected according to the topology in Figure 9.1:

a *target* datapath, namely the switch that is currently subject to the Ryu test; a *tester* datapath, namely a separate OpenFlow switch that is used to generate test packets to be sent to the target datapath, and to recollect any packets forwarded by the target datapath; and a *server* machine, where the Ryu test suite is being executed. While the test is running, Ryu acts as an

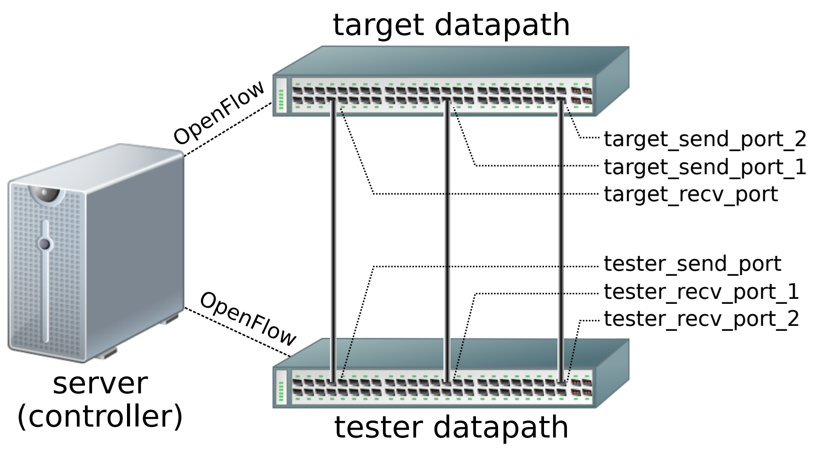


Figure 9.1: Network topology required in order to run the Ryu test suite.

OpenFlow controller connected to both the target and the tester switch. The typical actions performed to execute a test case are as follows:

1) Ryu sends an `OFPT_FLOW_MOD` message to the target datapath, to install a flow entry that matches packets entering a specific port (target_recv_port in Figure 9.1) and with specific headers. This flow entry usually applies a single action, which is to forward matching packets out of a target_send_port_1. In addition, it may apply manipulations to packet headers (e.g., push/pop of VLAN tags, TTL alteration, etc.). Moreover, Ryu also sends an `OFPT_FLOW_MOD` message to the tester datapath, to install a flow entry that matches packets entering port tester_recv_port_1 and sends them to the controller.

2) Ryu sends an `OFPT_PACKET_OUT` message to the tester datapath, soliciting it to produce a test packet to be sent to the target datapath.

3) The tester datapath sends the requested test packet out of its tester_send_port to the target datapath.

4) The flow entry on the target datapath is matched, and the packet is forwarded out of target_send_port_1.

5) The tester datapath receives the test packet through `tester_recv_port_1`. The flow entry installed at step 1 matches the packet, and the tester datapath sends an `OFPT_PACKET_IN` message to Ryu, enclosing the test packet.

6) Ryu checks whether the headers and contents of the received (and, possibly, altered) copy of the test packet coincide with the expected ones, and reports the outcome of the test case accordingly.

Test cases in the Ryu test suite are fully specified in the JSON format (see [Ryu15]), making them easy to understand and customize. Considering the extension of the test cases included in the standard Ryu distribution, we had no need to add or modify any of them. The only change we applied was to turn statically specified port numbers into parametric ones, so that they could conveniently be changed from the Ryu command line to match the OpenFlow port numbers exposed by the target and tester datapaths.

The Ryu test suite considers four classes of test cases:

[*action*] – Tests in this class install on the target datapath flow entries that perform simple packet forwarding and various types of packet modification (e.g., TTL alteration, push/pop of VLAN and MPLS headers).

[*group*] – For these tests Ryu installs on the target datapath flow entries that forward a copy of the test packet out of multiple ports, thus testing support for group actions; this test class requires the additional `target_send_port_2` on the target datapath and the corresponding `tester_recv_port_2` on the tester datapath.

[*match*] – The flow entries installed on the target datapath to perform these tests consider an extensive assortment of match conditions applied to various header fields, exploit multiple flow tables, apply bitmasks on the matched fields, and alternatively forward packets out of a port or directly to the controller.

[*meter*] – To perform tests in this class, the meter table of the target datapath is suitably populated, then Ryu emits packets from the tester datapath at a configured rate and checks if the target datapath withstands a certain packet transfer throughput.

In order to check the ability of the target datapath to apply 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).

Table 9.2: A sample of the custom tests that we defined and the functionalities they verify.

Functional tests	
`ct_normal`	Support for the NORMAL reserved port as a flow entry action
`ct_multi_ctrl`	How the datapath behaves with multiple configured controllers
`ct_hidden`	Whether the datapath maintains 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 and multiple VLAN headers
`ct_mpls`	Support for pushing/popping single and multiple MPLS headers (and determination of limits on the size of the label stack)
`ct_metadata`	Support for metadata in match conditions and actions
Performance tests	
`ct_perf_switch`	Whether the size of the flow table affects switching performance
`ct_cpu`	Whether the CPU is involved in flow entry matching and packet switching processes
`ct_flow_insert`	Time required by the datapath to install entries in the flow table

Custom Tests

The Ryu test suite is more than enough to determine the OpenFlow features supported by a datapath. However, we believe a few additional tests are required to construct a more complete picture of the readiness of a device to operate in an SDN-based network architecture. We defined a set of custom tests, some of which are listed in Table 9.2, that we executed both to verify additional datapath features required to exploit the full potential of SDN, and to investigate deeper in the causes of failures reported by the Ryu test.

For example, we exploited test `ct_normal` to determine the ability of a datapath to communicate in-band with the controller, namely on the same network links that are used to switch data packets using OpenFlow. There are at least two prerequisites to achieve this: support for *hybrid mode* (i.e., standard and OpenFlow packet switching on the same device), which can be checked in the documentation, and support for the NORMAL reserved port, which is optional by the specification and can be verified by this test. 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 a set of default flow entries that are not normally visible in any flow tables and yet influence its behavior.

We defined targeted tests for a small subset of the Ryu test cases, which we used to investigate the cause of reported failures. These tests simply 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 (e.g., matching only the VLAN tag) and using a network sniffer for monitoring the test packets entering and exiting that datapath. This helped us reveal that certain Ryu test cases failed just because the test flow entries mixed unsupported features with other features that would be supported if used alone, or because the value of a header field was off by one unit, which we do not consider to be detrimental for the deployability of SDN.

We also carried out performance tests, aimed at establishing how well a datapath performs in the presence of flow tables of varying size. For the `ct_perf_switch` test we connected 10GbE ports on the target datapath to create two loop topologies, as shown in Figure 9.2. Correspondingly, we installed very simple flow entries matching on the input port and forwarding packets along the loops. For each topology, we first used `OFPT_PACKET_OUT` messages to make the target datapath inject as many test packets in every loop as required to saturate the bandwidth of the involved ports. We used small IPv4 packets with a payload of 46 bytes as test packets, in order to increase the frequency of lookups in the flow table. Once 100% port usage was steadily reached, we started adding a progressively increasing number of entries to the flow table: each of these entries simply matched a different non-existent IP address, and its action was to forward packets out of a random port. 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

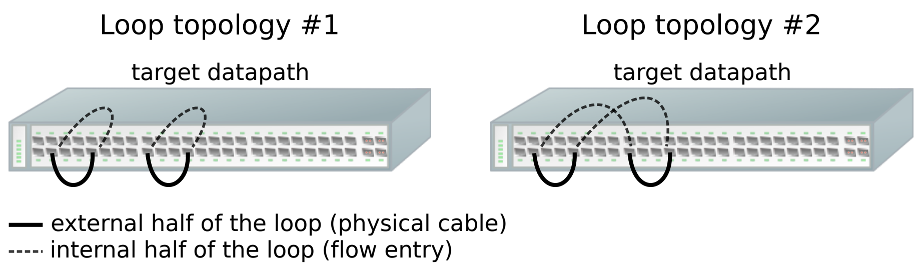

Loop topology #1
target datapath
Loop topology #2
target datapath
external half of the loop (physical cable)
internal half of the loop (flow entry)

Figure 9.2: Network topologies used during the performance tests.

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 on the target datapath, to determine whether packet forwarding was hardware accelerated (test `ct_cpu`). Finally, for the `ct_flow_insert` test we cleared the flow table of the target datapath and then measured the time required to install a fixed number of flow entries. Each flow entry matched IP packets with randomly chosen IP addresses (duplicates were avoided) and had a single action, CONTROLLER. The measurement was accomplished by sending an `OFPT_BARRIER_REQUEST` message immediately after each `OFPT_FLOW_MOD` and by assessing the time elapsed between adding the first flow entry and receiving the last `OFPT_BARRIER_REPLY`. We repeated this measurement for an increasingly higher number of installed flow entries, in order to determine whether each tested datapath had a breakdown point.

9.5 Outcome of Device Tests

We applied the methodology in Section 9.4 to perform an extensive session of tests on a range of hardware datapaths. In this section we describe the results of these tests, which largely confirm the limitations discussed in Section 9.2 and unveil additional ones.

Devices Under Test

We considered 7 hardware datapaths manufactured by 7 major vendors. 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.

Table 9.3: 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

Some of them ran a customized version of Open vSwitch (OVS) [ovs15] 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 specified in Table 9.3. 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 [Ryu16]).

Test Setup

We performed the OpenFlow compliance tests as described in Section 9.4, connecting each target datapath as specified in Figure 9.1. As the sole exception, we could not connect the `target_send_port_2` of S2, which prevented us from running some classes of test cases (e.g., `group` and `meter`). Moreover, we could not execute any performance tests on this datapath.

We used OVS version 2.3.90 as the tester datapath, run 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, 2GB of RAM, and one 100Mbps network interface. We used Ubuntu Server Linux 14.04.2 LTS as the operating system on both servers. The control channel between Ryu, the tester datapath, and the target datapath was realized by a dedicated 100Mbps management network.

Before being able to execute the Ryu test suite, we had to reconstruct for every datapath the OpenFlow port numbers associated with the physical interfaces that connected the target and the tester datapath. This task was particularly important considering that OpenFlow port numbers may be assigned based on arbitrary conventions (see, e.g., [Ext15]) and not necessarily in the same order as the corresponding physical interfaces. We then assigned the roles of the ports according to Figure 9.1 and passed the selected port numbers to Ryu.

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 9.4 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 9.3 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 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. Even more 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

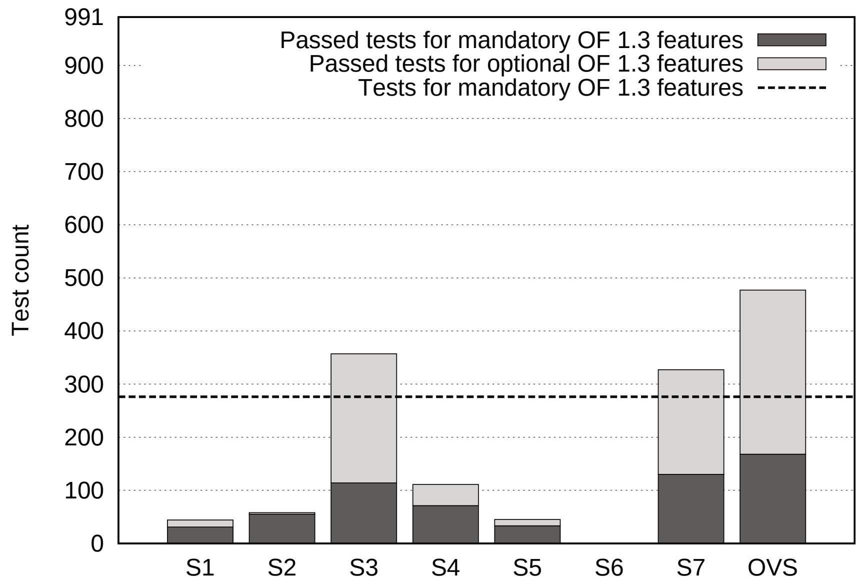


Figure 9.3: Count of passed Ryu tests.

this plot as well as the following ones is due to a subtle bug in the OpenFlow implementation, that we analyze in the final part of this section.

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 9.4. Figure 9.4 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 in 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 9.3).

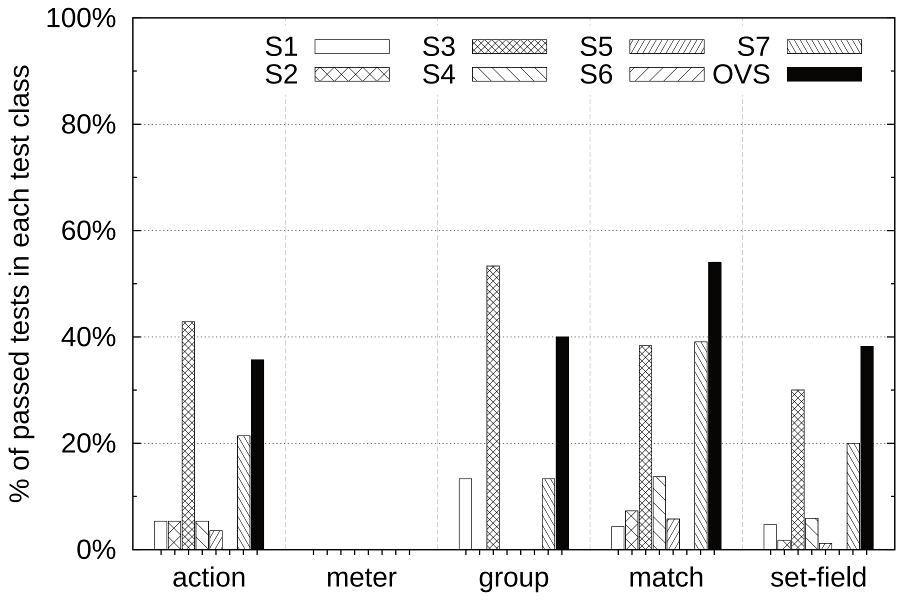


Figure 9.4: Percentage of passed OF 1.3 Ryu tests per test class.

However, 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 9.5. 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[1]. For example, the MPLS class in Figure 9.5 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 (note that this is different from saying that MPLS is the class of tests that verify support for MPLS-related functionalities). From the

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

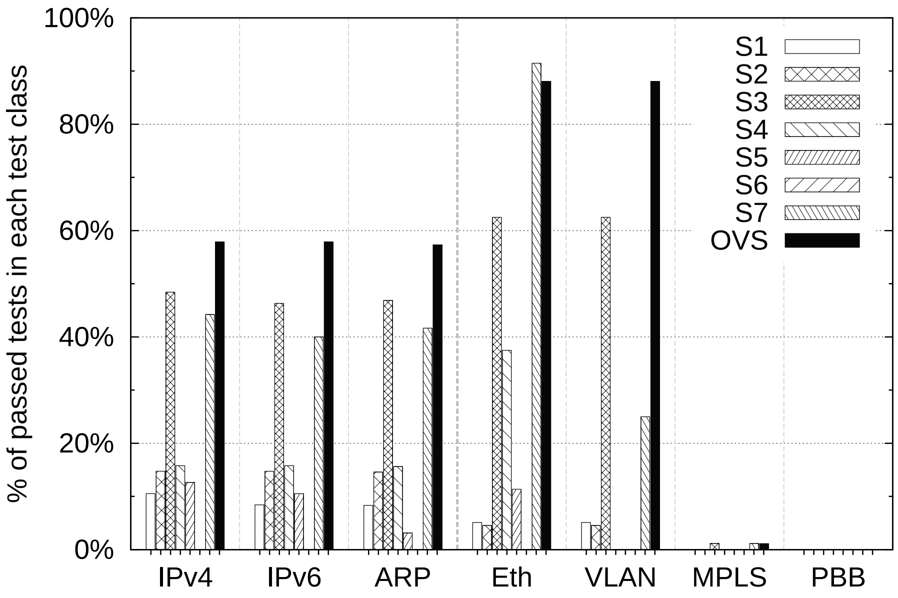


Figure 9.5: Percentage of passed OF 1.3 Ryu tests for test packets carrying a specific protocol.

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 observations made in Figure 9.4, if we exclude OVS, datapath S3 always performed better than any others, with the sole exception of the Eth class, where S7 remarkably exceeded 90% of passed tests.

We now describe the results obtained during the performance tests. 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 after reaching saturation resulted in the anomalous behaviors described in Section 9.6. S6 was so flawed that we could not reliably exploit the installed flow entries to realize the loops in Figure 9.2. 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 9.6 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: as described in Section 9.4, we used flow entries with a very simple structure to limit this bias. The plot, whose X axis is 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 [KPK15]). `S4` exhibited an almost exponential increase in the insertion times, while `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.

9.6 A Catalog of Experienced Issues

While running the tests described in Section 9.5, we experienced a number of unexpected behaviors that we consider useful to document here. We debugged these issues with the help of the targeted tests described in Section 9.4, 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 `OFPT_PACKET_OUT` message).

Functional Bugs Some of the unexpected behaviors were due to missing or improperly implemented features. In some cases such issues were solved with firmware upgrades (or even downgrades). For example, on one datapath every flow entry was automatically removed soon after being installed, and on another datapath setting the table-miss flow entry to send packets to the controller had no effect (but an entry in a standard flow table did). In other cases we were just stuck with unusable sets of features. In particular, we often

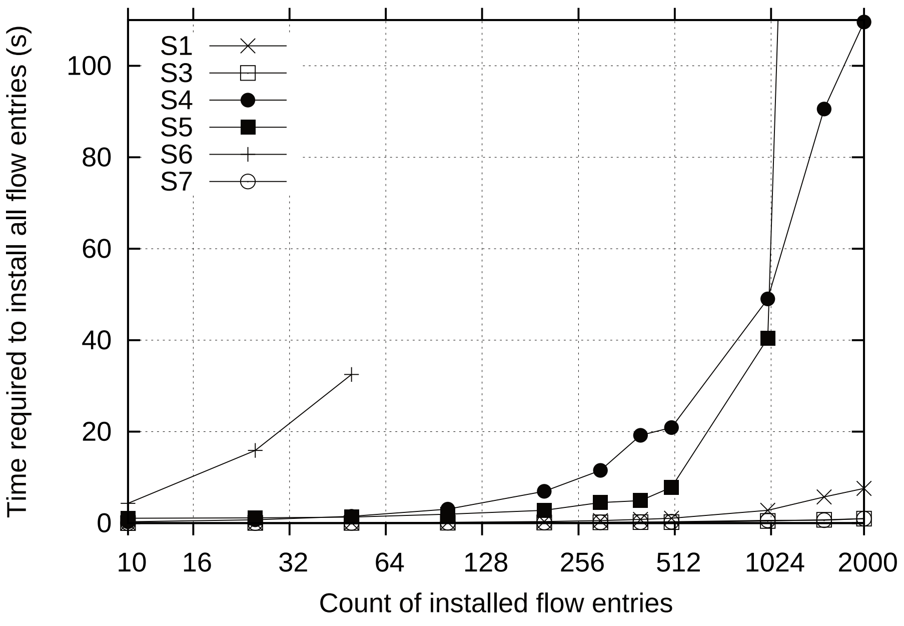


Figure 9.6: Flow entry insertion performance.

observed packets that were inconsistently matched by the flow entries: for example, some entries only worked if the match condition included the Ethernet type; other entries matching the VLAN VID only worked if their action was CONTROLLER; in one case, matched flow entries did not apply the action "push VLAN header"; in another case, matched entries did not forward packets as supposed to, apparently because of additional matched fields besides the packet's input port. We suspect that these behaviors could be due to improperly handled TCAM entries, but also argue that such situations should be avoided by prohibiting flow entry installation altogether. In extreme cases, we saw the same set of flow entries non-deterministically matching or missing the same test packets (causing the Ryu tests to succeed only on a fraction of the runs), and we witnessed a single 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 it 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, OFPT_PACKET_IN messages generated by a flow entry with action CONTROLLER always had the "reason" field incorrectly set to OFPR_NO_MATCH, meaning that all the flow entries were missed. On S6, the "length" value of the "match" field carried by OFPT_PACKET_IN messages (which contains the datapath port through which a packet was received) was always incorrectly computed, causing OFPT_PACKET_IN 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).

We are aware that implementation flaws may occur as more recent versions of the specification are released, but we argue that at the current level of maturity of the OpenFlow specification certain basic functionalities are expected to work out of the box.

Violations of the Specification Other misbehaviors were evidence of violations in the implementation of the OpenFlow specification. As such, they caused some Ryu tests to fail, but 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 to assign IDs starting from 1. While this does not affect the datapath operation, it 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 failing Ryu tests because 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).

In the presence of VLAN headers, 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 weird conditions that we could discover only while carrying out performance tests. `OFPT_PACKET_OUT` 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 extra 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 [KPK15].

During test `ct_flow_insert`, not all the datapaths could keep the pace with `OFPT_FLOW_MOD` 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.

Additional Considerations We conclude the section with a few additional advices about implementation-specific choices that, although not disruptive, in our opinion should be taken into account when considering a datapath for production use.

First of all, we ascertained that not all datapaths are capable of handling IPv6 with OpenFlow (actually, only 3 out of the 7 we tested): this can impair the applicability of SDN on modern network architectures.

The default action undertaken by the table-miss flow entry (i.e., drop or send to controller) may vary across vendors, potentially resulting in lost packets if not set in advance to the intended value. Moreover, losing connection with the controller may result in the flow table being cleared (and no emergency flows being installed).

Some datapaths require carefully planning the amount of installed flow entries in advance, because their TCAMs need to be partitioned to reserve a space for the flow tables. Since partitioning schemes are typically constrained to predefined profiles, and every flow entry may consume several hardware entries in the TCAM, choosing the optimal setup may not be an obvious task. As an additional restriction, on some devices TCAM memory blocks are allocated to specific types of flow entries (e.g., matching L2 or L3 fields only), and some network modules may not support mixing these types.

As a side note, usually no more than 2 or 3 controllers can be configured for a single datapath, thus imposing limits on scalability and redundancy. Moreover, a secure (e.g., SSL) communication channel may only be supported for a subset of the configured controllers.

9.7 Related Work

Assessing the level of compliance of a device with the OpenFlow specification is a useful piece of information for the research community, but is a priority especially for device manufacturers. For this reason, several efforts are being devoted to the development of automated systems that accomplish this task. We review here the most important ones, classifying them according to the coordinates in Table 9.4.

Ryu [Ryu15], which we thoroughly described in the previous sections, is a very active and feature-rich community effort to build an SDN controller framework that includes a device testing tool. Indeed, to our knowledge it is the open source project offering the highest number of test cases, which are specified in JSON format and therefore easy to customize. The development community has also published at [Ryu16] the results of compliance tests run on a small set of devices. However, they are not complemented by a deep investigation of their outcome. oftest [Pro16] offers a much more limited set of test cases and, differently from Ryu, it requires root privileges to be executed, because test packets are generated (using the packet manipulation tool Scapy) and monitored on the same machine on which the test is executed. Supporting additional test cases is also not immediate, because it requires writing Python code. In July 2013 the Open Networking Foundation (ONF) launched an OpenFlow conformance test program [Ope15a], specifying a number of test cases comparable to oftest. A conformance test for OpenFlow 1.0.1 is currently available at [Ope13a], approved SDN testing laboratories have been set up, and a freshly updated list of certified products is available at [Ope16a], but

Table 9.4: Comparison of currently available OpenFlow compliance test suites.

Test suite	**OpenFlow version(s)**	**Test cases**
Ryu [Ryu15]	1.3, 1.4	991 (Ryu 3.18)
oftest [Pro16]	1.2, 1.3, 1.4 in the works	About 200
ONF OpenFlow 1.0.1 conformance test [Ope13a]	1.0.1	208
Open SDN network virtualization test [SOL]	N/A	<100
Ixia IxANVL OpenFlow test suite [Ixi15]	1.0.1, 1.3.2	>194 (OF 1.0); 528 (OF 1.3)
OFLOPS [RSU$^+$12]	1.0	<50
Veryx ATTEST OpenFlow conformance test [Ver14]	1.0.0, 1.3.1, 1.3.2, 1.3.3	400
Spirent TestCenter OpenFlow compliance test [Spi16]	1.0.1, 1.3, 1.4	>950 (OF 1.3); >100 (OF 1.4)

we are not aware of any publicly available associated test software. The test specification for OpenFlow 1.3 is planned to comprise as many as 1100 test cases but, as of January 2015, it is still under development. The Open SDN Network Virtualization Test [SOL] focuses on performance rather than functionality. The project team has also published the results of tests conducted during a one-shot event that took place in 2013, but there seems to have been no further activity since then, and the used test tools are not publicly available. As an evolution of [SOL], Ixia includes in its commercially available Automated Network Validation Library (IxANVL) comprehensive test suites for OpenFlow 1.0 and 1.3 [Ixi15], consisting of a regularly updated set of more than 500 test cases and comprising test cases from oftest. OFLOPS [RSU$^+$12] is an effort to define a modular architecture for a testing tool. The authors developed 14 readily usable modules, but they consider few basic functionalities, and the application of the tool documented in [RSU$^+$12] is only focused on performance aspects (e.g., flow table update rate). The Veryx ATTEST Conformance Test Suite for OpenFlow switch [Ver14] supports pure OpenFlow as well as hybrid switches and has a rich set of test cases. However, it is exceeded by the Spirent TestCenter OpenFlow compliance test suite [Spi16], which comprises more than

950 tests for OpenFlow 1.3, updated on a quarterly basis.

A very recent contribution [KPK15] analyzes the performance of flow table updates in a sample of 3 hardware switches. Similarly to our work, it includes a discussion of interesting findings about anomalous behaviors that the authors discovered during their tests.

We are not aware of any other contributions that analyze the readiness of network devices to operate in an SDN scenario from a functional point of view with the same accuracy that we pursue in this work.

9.8 Conclusions and Future Work

In this work we raise two important questions: one is that, besides flow table capacities, research in the SDN field is largely unaware of several restrictions encountered when it comes to deploying a proposed architecture on real devices. The other is that network architects should choose network devices very carefully if they aim at switching to an SDN-based infrastructure. We address these questions with a review of publicly available documentation from various vendors, with a device testing methodology, and with a comprehensive set of tests executed on a range of heterogeneous devices, that revealed several anomalous behaviors.

As an obvious continuation of this work, our 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 [Ope13a], we would like to publicly release this specification. On the methodological side, we will be considering how existing SDN-based architectures such as [LRVB15] can be fit to comply with vendor-imposed restrictions.

Part V

Other Research Activities and Publications

Chapter 10

Other Research Activities

This chapter presents additional research activities that took place during the doctorate program. The following sections are not part of the previous chapters of this thesis because they have little or no overlap with the topic of Internet eXchange Points. Nonetheless, they describe interesting approaches all related to the Software Defined Network topic, a topic strongly present also in this thesis. Moreover many of the solutions presented in this chapter played an important role for the development of works previously presented in this thesis. For these reasons we think they could be interesting for the reader.

10.1 SDNetkit: A Testbed for Experimenting SDN in Multi-Domain Networks

Mininet is the *de-facto* standard simulation environment for experimenting with SDN enabled networks based on the OpenFlow protocol. Although Mininet is powerful and not resource hungry, it has a strong limitation: it is not possible to use it for networks in which both OpenFlow and standard distributed routing protocols (e.g. Open Short Path First, OSPF) simultaneously run.

In this work we present SDNetkit, an enhanced release of the widely used Netkit network emulator that overcomes the limitation imposed by Mininet. We improved Netkit by adding all needed software to run OpenFlow based networks (e.g. OpenVSwitch and the Ryu framework). We show two use cases in which OpenFlow and standard protocols coexist. In particular, we address interoperability problems by presenting one use case in which OpenFlow nodes interact with standard ones (e.g. OSPF routers) in multi-domain networks, as well as one use case in which the OpenFlow protocol and OSPF run on the same machine, discussing some problems related to specific configurations. We believe that having the possibility to experiment SDN also in presence of interoperability scenarios results in opening to new research perspectives.

Researchers and practitioners interested in SDN need systems to perform experiments with OpenFlow [MAB$^+$08] and, over the years, Mininet [min15] was the leader among those systems. The majority of the experiments on SDN, in particular based on the OpenFlow protocol, have been carried out on top of that light and easy to use simulation environment, that provides a very simple interface to the final users. Nevertheless, Mininet is not the only SDN-ready simulator: many others exist, like [Jur13, WCY13, LO15, WDS$^+$14, AS13, min17], even if Mininet is one of the most popular and supported ones. By using Mininet, it is possible to create arbitrary topologies, as well as running several OpenFlow-enabled devices, connecting them at will and run customized SDN controllers written by exploiting any available SDN framework (e.g. Ryu [ryu17]).

To provide OpenFlow functionalities, Mininet relies on OpenVSwitch [ovs15] that is a software switch equipped with an implementation of the OpenFlow protocol. Mininet can be seen as an *orchestrator* of OpenVSwitch instances, taking care of creating suitable connections among them according to what the user declares in the definition of the topology. However, it suffers from two strong limitations. First, with Mininet it is not possible to test topologies where legacy devices (e.g. IP-speaking routers running traditional routing protocols) coexist with OpenFlow-enabled devices. Second, it assumes that the controller is back-to-back with each device in the network.

We argue that these two weaknesses pose severe limitations in experimenting interesting scenarios, like *interoperability* among SDN-enabled and legacy devices, as well as how the communication among controller and SDN-enabled devices is affected by network changes (e.g. failures). For simplicity, we call the latter scenario *network control*.

It is reasonable to think that no currently operating provider will fully migrate its network to an SDN-enabled one in just a single step. As it often happens, an incremental process will take place, in which SDN-enabled and legacy devices will coexist for a certain amount of time. Hence, when emulating networks involving multiple ISPs, where each of them is at an intermediate step of the migration process towards SDN, it is crucial to have the possibility to emulate networks composed both by SDN-enabled devices and by legacy ones. This results in the interest in experimenting interoperability scenarios that, at this moment, cannot be emulated by any freely available SDN/OpenFlow simulator. On the other hand, the assumption that the SDN controller is directly connected to every SDN-enabled device in the network might not be always satisfied for several reasons. Having a management network connecting among them an SDN controller and devices leads to interesting experiments, like reactivity to failures in the network and which is the impact of those failures on the communication among controllers and SDN-enabled devices in terms of how much time the controller spends in producing a new data plane.

In this work, we present SDNetkit, an emulator built on top of the widely used Netkit network emulator [net17]. We enhanced Netkit by adding SDN functionalities. More specifically, we added OpenFlow software that makes Netkit SDN-ready. With SDNetkit is possible to overcome the limitations imposed by Mininet. In SDNekit it is possible to set up topologies in which legacy and SDN-enabled devices coexist, that is not feasible with most used simulation systems. Also, it is possible to simultaneously run the OpenFlow protocol and standard routing protocols (e.g. by using Quagga) on the same device. SDNetkit opens to the possibility of having SDN controllers and SDN-enabled devices not directly connected, giving the opportunity to test both interoperability and control network scenarios. To the best of our knowledge, SDNetkit is the first freely available emulator providing such functionalities. For the first release of SDNetkit, we provide basic software for SDN capabilities. Essentially, with SDNetkit is possible to run OpenVSwitch instances to set up OpenFlow devices and use the Ryu framework [ryu17] as the controller. We point out that this is just an initial choice. Indeed, there are no limitations in adding other software to provide SDN/OpenFlow functionalities (e.g. it is possible to add other SDN frameworks for implementing custom controllers, other SDN-enabled switch implementations, or general software for testing SDN).

10.2 Supporting End-to-End Connectivity in Federated Networks using SDN

Federated networking is a promising approach to resource sharing that supports cost-effective services involving multiple parties. Research in this field largely focused on architectures and cost models, making limited progress on the technological side. On the other hand, the widely adopted Software-Defined Networking (SDN) model found its most successful application in data centers, exhibiting very little penetration in other scenarios.

We leverage the unexplored potential of SDN on the edge of a network to introduce an approach that supports end-to-end connectivity among different federated partners. Our approach is based on simple Network Address and Port Translation (NAPT), making it applicable in standard IP networks. It is also very flexible, because it exploits SDN, and scalable, because address translations are performed on Customer Premises Equipment, where SDN is being progressively supported by device vendors. We define various alternative NAPT strategies and evaluate their effectiveness with simulations as well as emulated scenarios.

A federated network is an ensemble of independent but collaborating partners that share resources in order to optimize their usage, improve the quality of services, and reduce provisioning costs. While some design principles date back to 2008 [HEY08], federations have recently emerged in the field of cloud computing [cfl12], and their benefits in terms of cost effectiveness [GGT10] and performance [CRB$^+$11] are being largely investigated also in the context of research projects (e.g., GÉANT’s Future Network Research[1], BEACON[2]). Despite the interest of the research community in federated networks, limited progress has been done on technologies for realizing them.

We introduce an end-to-end connectivity service among the networks of federated partners which is based on applying ad-hoc Network Address and Port Translation (NAPT) strategies both to the source and to the destination IP address and port of packets exchanged by the partners. In this way, such packets can be routed on a standard IP network (including the Internet) without the need for encapsulation. Supporting this scenario requires a level of coordination among the involved parties which we accomplish by leveraging Software-Defined Networking (SDN) on the network edge, where it has been marginally exploited so far. A similar technique has been used in [SVR15], although with a different goal (inbound traffic engineering) and applied solely to the packets’ source addresses. The choice of SDN makes our approach flexible and scalable, since very little signaling information needs to be exchanged to negotiate end-to-end connections and address translations are performed by Customer Premises Equipment (CPE). While a similar connectivity service can be implemented using, e.g., IPsec Virtual Private Networks (VPNs), our approach is much simpler to deploy, does not introduce any technological dependencies, and, if required, can be combined with additional services such as encryption. Its feasibility is confirmed by the progressive introduction of SDN-capable CPE devices on the market (see, e.g. [Nov16]). We define various alternative NAPT

[1]`http://geant3.archive.geant.net/Research/Future_Network_Research/Pages/FederatedNetworkArchitectures.aspx`

[2]http://www.beacon-project.eu

strategies and evaluate their effectiveness using simulations as well as tests inside emulated network scenarios.

10.3 SDNS: Exploiting SDN and the DNS to Exchange Traffic in a Federated Network

Federated networks have primarily emerged to support cloud computing services, in order to reduce costs for providers, as well as to increase their incomes. Up to now, the research activity has been mostly focused on architectures and cost models, setting aside technological aspects.

In this work, we propose SDNS, an SDN-system that opens the application fields of federated networks to federated connectivity services. By exploiting the centralized architecture offered by SDN and relying on the OpenFlow protocol, the most adopted enabler for SDN, we propose a way to easily interact with the Domain Name System (DNS) traffic in order to allow communication among multiple customers connected to different providers in presence of any type of IP address plan. We tested the scalability of our approach in a prototype implementation based on Netkit, a widely adopted simulation environment. We measured several control-plane overhead metrics, like the number of DNS, OpenFlow, and SDNS messages exchanged in the network. Our experiments show that the scalability of our method is essentially the same of the DNS service.

A federated network is composed by two or more independent, but collaborating, members that share their resources in order to provide services, aiming at enforcing the quality of the services themselves, reducing provisioning costs and increasing revenues. For simplicity, but without loss of generality, we can assume that each member of a federated network is an Internet Service Provider (ISP), making its own resources available to the federation (e.g. storage, network devices, etc.). In the past years, several federated networks arose and started to issue services. Example of those networks are GÉANT [gea17a], Beacon [bea17], and NATO federated network [nat]. The common idea behind those networks is to easily provide different federated services to their customers.

Federated networks are widely used for cloud computing, where an entity, acting as a broker, handles resources in order to provide services to customers. After discussing with several national ISPs and analyzing with them some requirements, we argue that other services, like connectivity, can take advantage from the adoption of ISPs' federations. One of the most critical problems in providing connectivity services to customers is to ensure that no overlaps in the

IP addresses (typically private addresses) occur. This is obviously enforced by a single ISP, since it can carefully assign IP addresses to its customers avoiding overlaps, but this condition is no longer guaranteed when more ISPs join in a federation. Thus, a mechanism allowing point-to-point communications in the presence of multiple independent customers connected to different providers in such a situation is needed.

On the other hand, Software Defined Networking (SDN) is an architecture opening new perspectives for federated networks. By exploiting a centralized entity having the control of the entire network, together with the ability of programming the routing logic, SDN is able to enforce flexibility, as well as reduction in terms of configuration complexity, in providing services. In [dLRB16] a method for enabling connectivity services in federated networks relying on SDN is presented.

In this work, we build on [dLRB16] a system allowing ISPs to easily span Virtual Private Networks (VPNs) services over multiple networks handled by different providers. Achieving this goal is not trivial by using standard technologies and typically requires a lot of effort. Since we are addressing such a problem in federated networks, we call such a service *Federated VPN*. We leverage DNS for unique resource identification, with no changes on the operation of end hosts. Also, we do not require any changes to the organization of DNS servers (e.g. authoritative information). Our proposal is completely transparent for network devices (which continue to process standard IP packets) and we do not introduce any MTU restrictions, since we do not use any tunnels.

We present SDNS, an SDN-based method enabling federated VPN serivces. Our system allows customers to have any IP addressing plan. To achieve this goal we exploit the DNS, involving multiple SDN controllers in the name resolution process. Using the centralized approach offered by SDN, we can easily interact with all traffic exchanged in the network, including all DNS packets generated by each end-host. By looking at that traffic, we inspect the content of the DNS requests and answers and, in the case of IP addresses overlaps, we manipulate the content of the DNS packets overcoming undesired situations and, by using suitable Network Address Translation (NAT) techniques, we generate a set of rules that make the routing consistent with the original IP addresses. SDNS allows ISPs to provide novel features, like *on-demand* federated VPNs set up for a pre-defined amount of time.

SDNS relies on the Ryu framework [ryu17] and the OpenFlow protocol version 1.3 [MAB$^+$08], the most adopted protocol enabling SDN in the networks. Our proposal requires a minimum set of configurations related to the federated connectivity service that is placed at the SDN-controller, without any impact on existing configurations, like routing protocols configurations. We argue that this enables SDNS to be easily deployed in production networks, also considering that it does not require any technological dependencies.

10.4 Leveraging SDN to Monitor Critical Infrastructure Networks in a Smarter Way

In critical infrastructures, communication networks are used to exchange vital data among elements of Industrial Control Systems (ICSes). Due to the criticality of such systems and the increase of the cybersecurity risks in these contexts, best practices recommend the adoption of Intrusion Detection Systems (IDSes) as monitoring facilities. The choice of the positions of IDSes is crucial to monitor as many streams of data traffic as possible. This is especially true for the traffic patterns of ICS networks, mostly confined in many subnetworks, which are geographically distributed and largely autonomous. We introduce a methodology and a software architecture that allow an ICS operator to use the spare bandwidth that might be available in over-provisioned networks to forward replicas of traffic streams towards a single IDS placed at an arbitrary location. We leverage certain characteristics of ICS networks, like stability of topology and bandwidth needs predictability, and make use of the Software-Defined Networking (SDN) paradigm. We fulfill strict requirements about packet loss, for both functional and security aspects. Finally, we evaluate our approach on network topologies derived from real networks.

ICSes are the core of critical infrastructures. They are composed by many elements that interact by means of a communication network, which we call *ICS network*. Main elements of an ICS are embedded devices that control actuators or gather data from sensors. Special servers are in charge to collect data from these embedded devices, show them to the control room operators, record them in a database, change settings according to operators requests, etc. While the data that flow in an ICS network are very specific, standard networking technologies can be adopted for its implementation.

In the past decade, a growth of cyber-attacks directed toward ICSes has been observed [cscertcssp11]. For the security of the ICS networks, best practices suggest to deploy network-based IDSes [SLP$^+$15]. In regular networks it is acceptable to observe traffic in a small number of relevant points. However, for reliability reasons, in ICSes, Supervisory Control And Data Acquisition (SCADA) servers are close to sensors and actuators, hence, traffic is mostly local. Further, attacks to ICSes are potentially carried out by organizations (e.g., governments, intelligence agencies, terrorist groups) that can have insiders and that can carefully design attacks so that they pass unobserved by sparsely deployed IDSes. Tapping traffic close to all embedded devices and servers can easily lead to prohibitive costs. Certain solutions [nex11] make possible to route traffic replicas using the same ICS network towards one, or a few, IDSes, but they are not able to guarantee the successful delivery of critical ICS traffic in all cases.

In this work, we present a methodological approach and an architecture to (i) allow an operator to choose which traffic has to be observed within an ICS network without installing new hardware, (ii) enable the use of the spare bandwidth in the network to forward the traffic to be observed toward an IDS, while avoiding packet loss for regular traffic, and (iii) guarantee that the IDS receives all the traffic that the operator configured to be observed in order not to introduce false negatives due to packet loss. Our solution takes advantage of the fact that topology and bandwidth usage are quite stable in ICS networks (see for example [TCB08]), allowing us to assume in advance knowledge of ICS network's traffic, since it derives from ICS design, and to perform a global off-line optimization of switching paths. Furthermore, we support the usage of the ICS network for additional and occasional traffic, which are always considered potentially dangerous. We assume that this traffic can be served with a best-effort approach while maximizing the endeavor in observing it. We propose an architecture that exploits the Software-Defined Network (SDN) approach as prescribed by the OpenFlow specifications [Spe13]. We evaluated our methodology against four network topologies, derived from real topologies and augmented with realistic networks in the domain of electrical distribution. Our experiments show that our optimization problem can be easily solved for those scenarios in reasonable time and our approach makes efficient use of the bandwidth when the topology allows it.

Chapter 11

List of Publications

- Gabriele Lospoto, Habib Mostafaei, Roberto di Lallo, Massimo Rimondini, Giuseppe Di Battista. "SDNetkit: A Testbed for Experimenting SDN in Multi-Domain Networks" *IFIP/IEEE International Symposium on Integrated Network Management (IM 2017), 2017.*
- Habib Mostafaei, Gabriele Lospoto, Andrea Brandimarte, Roberto di Lallo, Massimo Rimondini, Giuseppe Di Battista. "SDNS: Exploiting SDN and the DNS to Exchange Traffic in a Federated Network" *IEEE Conference on Network Softwarization (IEEE NetSoft 2017), 2017.*
- Marco Chiesa, Roberto di Lallo, Gabriele Lospoto, Habib Mostafaei, Massimo Rimondini, Giuseppe Di Battista. "PrIXP: Preserving the Privacy of Routing Policies at Internet eXchange Points" *IFIP/IEEE International Symposium on Integrated Network Management (IM 2017), 2017.*
- Roberto di Lallo, Federico Griscioli, Gabriele Lospoto, Habib Mostafaei, Maurizio Pizzonia, Massimo Rimondini. "Leveraging SDN to Monitor Critical Infrastructure Networks in a Smarter Way" *IFIP/IEEE International Symposium on Integrated Network Management (IM 2017), 2017.*
- Roberto di Lallo, Gabriele Lospoto, Massimo Rimondini, Giuseppe Di Battista. "How to Handle ARP in a Software-Defined Network." *In Conference on Network Softwarization (NetSoft 2016), IEEE, 2016.*
- Roberto di Lallo, Gabriele Lospoto, Massimo Rimondini, Giuseppe 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.*

- Roberto di Lallo, Mirko Gradillo, Gabriele Lospoto, Claudio Pisa, Massimo 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.*

- Marco Di Bartolomeo, Giuseppe Di Battista, Roberto di Lallo, Claudio Squarcella. "Is It Really Worth to Peer At IXPs? A Comparative Study." *In Proc. ISCC, 2015.*

SUBMITTED:

- Habib Mostafaei, Gabriele Lospoto, Roberto di Lallo, Massimo Rimondini, Giuseppe Di Battista "A Framework for Multi-Provider Virtual Private Networks in Software-Defined Federated Networks".

IN SUBMISSION:

- Gianni Antichi, Timm Bottger, Marc Bruyere, Ignacio Castro, Roberto di Lallo, Eder Leao Fernandes, "Through the Lens of IXPs: a Decade of Internet Evolution".

Bibliography

[ACF$^+$] Bernhard Ager, Nikolaos Chatzis, Anja Feldmann, Nadi Sarrar, Steve Uhlig, and Walter Willinger. Anatomy of a large european ixp. In *Proc. ACM SIGCOMM 2012*.

[ACF$^+$12] Bernhard Ager, Nikolaos Chatzis, Anja Feldmann, Nadi Sarrar, Steve Uhlig, and Walter Willinger. Anatomy of a Large European IXP. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2012.

[AG] M.Z. Ahmad and R. Guha. A tale of nine internet exchange points: Studying path latencies through major regional ixps. In *Local Computer Networks (LCN), 2012*.

[AJ07] F. Audet and C. Jennings. Network Address Translation (NAT) Behavioral Requirements for Unicast UDP. IETF RFC 4787, January 2007.

[AKW] Brice Augustin, Balachander Krishnamurthy, and Walter Willinger. Ixps: Mapped? In *IMC 2009*.

[AKW09] Brice Augustin, Balachander Krishnamurthy, and Walter Willinger. IXPs: Mapped? In *Internet Measurement Conference (IMC)*. ACM, 2009.

[Ale] `http://www.alexa.com/`.

[AMS16] AMS-IX. Amsterdam internet exchange point members, Sept 2016.

[ana] `www.analysysmason.com`.

[Ari15] Arista Networks, Inc. Arista EOS 4.15.3F user manual, Nov 2015.

[ark] Caida ark. `http://www.caida.org/projects/ark/`.

[AS13] Vitaly Antonenko and Ruslan Smelyanskiy. Global network modelling based on mininet approach. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 145–146. ACM, 2013.

[atl] RIPE Atlas. `http://atlas.ripe.net`.

[BA13] R. Bush and R. Austein. The Resource Public Key Infrastructure (RPKI) to Router Protocol. IETF RFC 1997, June 2013.

[BCT$^+$16] Timm Böttger, Felix Cuadrado, Gareth Tyson, Ignacio Castro, and Steve Uhlig. Open connect everywhere: A glimpse at the internet ecosystem through the lens of the netflix cdn. In *arXiv preprint arXiv: 1606.05519*, 2016.

[bea17] Beacon European Project, 2017.

[bin17] Bind name server software, Jan 2017.

[Bro15] Brocade. Brocade NetIron Software Defined Networking (SDN) configuration guide, May 2015.

[CAZ$^+$14] Jakub Czyz, Mark Allman, Jing Zhang, Scott Iekel-Johnson, Eric Osterweil, and Michael Bailey. Measuring IPv6 Adoption. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2014.

[CCGF14] Ignacio Castro, Juan Camilo Cardona, Sergey Gorinsky, and Pierre Francois. Remote Peering: More Peering without Internet Flattening. In *Conference on emerging Networking EXperiments and Technologies (CoNEXT)*. ACM, 2014.

[CDA$^+$16] Marco Chiesa, Christoph Dietzel, Gianni Antichi, Marc Bruyere, Ignacio Castro, Mitch Gusat, Thomas King, Andrew W Moore, Thanh Dang Nguyen, Philippe Owezarski, et al. Inter-domain networking innovation on steroids: empowering ixps with sdn capabilities. In *Communications Magazine, Volume: 54, Issue: 10*. IEEE, 2016.

[CDC+16] Marco Chiesa, Daniel Demmler, Marco Canini, Michael Schapira, and Thomas Schneider. Towards securing internet exchange points against curious onlookers. In *Applied Networking Research Workshop (ANRW 2016)*, 2016.

[CDP13] Luca Cittadini, Giuseppe Di Battista, and Maurizio Patrignani. MPLS virtual private networks. In *Recent Advances in Networking, Volume 1*, ACM SIGCOMM eBook, pages 275–304. ACM, 2013.

[cfl12] Cloud federation in a layered service model. *Journal of Computer and System Sciences*, 78(5):1330 – 1344, 2012.

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

[CKL15] H. Cho, S. Kang, and Y. Lee. Centralized ARP proxy server over SDN controller to cut down ARP broadcast in large-scale data center networks. In *Proc. ICOIN*, Jan 2015.

[CMT+11] Andrew R. Curtis, Jeffrey C. Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. DevoFlow: Scaling flow management for high-performance networks. *SIGCOMM Comput. Commun. Rev.*, 41(4):254–265, August 2011.

[CRB+11] Rodrigo N. Calheiros, Rajiv Ranjan, Anton Beloglazov, CÃľsar A. F. De Rose, and Rajkumar Buyya. Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Pract. Exper.*, 41(1):23–50, 2011.

[CRS12a] Juan Camilo Cardona Restrepo and Rade Stanojevic. A History of an Internet eXchange Point. In *Computer Communication Review, Volume: 42, Issue: 2*. ACM, 2012.

[CRS12b] Juan Camilo Cardona Restrepo and Rade Stanojevic. A history of an internet exchange point. *SIGCOMM Comput. Commun. Rev.*, 42(2):58–64, 2012.

[CSB+] Nikolaos Chatzis, Georgios Smaragdakis, Jan Böttger, Thomas Krenc, and Anja Feldmann. On the benefits of using a large ixp as an internet vantage point. In *Proceedings of IMC 2013.*

[CSB$^+$13] Nikolaos Chatzis, Georgios Smaragdakis, Jan Böttger, Thomas Krenc, Anja Feldmann, and W Willinger. On the Benefits of Using a Large IXP as an Internet Vantage Point. In *Internet Measurement Conference (IMC)*. ACM, 2013.

[cscertcssp11] Industrial control systems cyber emergency response team control systems security program. Ics-cert incident response summary report 2009-2011. Technical report, ICS-CERT, 2011.

[CSFW] Nikolaos Chatzis, Georgios Smaragdakis, Anja Feldmann, and Walter Willinger. There is more to ixps than meets the eye. *SIGCOMM Comput. Commun. 2013.*

[CSFW15] Nikolaos Chatzis, Georgios Smaragdakis, Anja Feldmann, and Walter Willinger. Quo vadis Open-IX? In *Computer Communication Review, Volume: 45, Issue: 1.* ACM, 2015.

[CTL96] R. Chandra, P. Traina, and T. Li. BGP Communities Attribute. IETF RFC 1997, August 1996.

[dat] `http://www.theguardian.com/uk/2013/jun/21/gchq-cables-secret-world-communications-nsa`.

[DD] Amogh Dhamdhere and Constantine Dovrolis. Ten years in the evolution of the internet ecosystem. In *Proceedings of IMC 2008.*

[DD11] Amogh Dhamdhere and Constantine Dovrolis. Twelve years in the evolution of the internet ecosystem. IEEE/ACM, 2011.

[DDdS15] Marco Di Bartolomeo, Giuseppe Di Battista, Roberto di Lallo, and Claudio Squarcella. Is it really worth to peer at ixps? a comparative study. In *Proc. 20th IEEE Symposium on Computers and Communication (ISCC 2015)*, pages 421–426, 2015.

[Del15] Dell. OpenFlow deployment and user guide 3.0, Feb 2015.

[dep] `drpeering.net/AskDrPeering/blog/articles/Ask_DrPeering/Entries/2013/11/30_Tier_1_ISP_Telecom_Italia_has_De-Peered_Everyone.html`.

[dLRB16] Roberto di Lallo, Gabriele Lospoto, Massimo Rimondini, and Giuseppe Di Battista. Supporting end-to-end connectivity in federated networks using SDN. In Melike Erol-Kantarci, Brendan

Jennings, and Helmut Reiser, editors, *Proc. IEEE/IFIP Network Operations and Management Symposium (NOMS 2016)*, pages 759–762, 2016.

[DPSBM13] Andrea Detti, Claudio Pisa, Stefano Salsano, and Nicola Blefari Melazzi. Wireless mesh software defined networks (wmSDN). In *Proc. WiMob*, 2013.

[EN16] Exa-Networks. Exabgp, Sep 2016.

[et 15] M. McCauley et al. POX SDN platform, Mar 2015.

[Ext15] Extreme Networks, Inc. ExtremeXOS 15.7 user guide, Feb 2015.

[FBP$^+$10] Lars Fischer, Bartosz Belter, Milosz Przywecki, Maribel Cosin, Paul Van Daalen, Marijke Kaat, Ivana Golub, Branko Radojevic, Srdjan Vukovojac, and Andreas Hanemann. Building federated research networks in europe. In *Terena Networking Conference*, 2010.

[FFML13] D. Farinacci, V. Fuller, D. Meyer, and D. Lewis. The Locator/ID Separation Protocol (LISP). RFC 6830 (Experimental), January 2013.

[flo15] Project Floodlight, Mar 2015. http://www.projectfloodlight.org/floodlight/.

[FRA16] FRANCE-IX. France internet exchange point members, Sept 2016.

[GALM08] Phillipa Gill, Martin Arlitt, Zongpeng Li, and Anirban Mahanti. The flattening internet topology: Natural evolution, unsightly barnacles or contrived collapse? In *Passive and Active Network Measurement*. Springer, 2008.

[GCF$^+$] Arpit Gupta, Matt Calder, Nick Feamster, Marshini Chetty, Enrico Calandro, and Ethan Katz-Bassett. Peering at the internet's frontier: A first look at isp interconnectivity in africa. In *Proc. PAM 2014*.

[gea17a] Géant European Project, 2017.

[gea17b] Géant European Project VPN services, 2017.

[GGT10] I. Goiri, J. Guitart, and J. Torres. Characterizing cloud federation for enhancing providers' profit. In *Proc. CLOUD*, 2010.

[GILO11a] Enrico Gregori, Alessandro Improta, Luciano Lenzini, and Chiara Orsini. The impact of ixps on the as-level topology structure of the internet. *Computer Communic.*, 34(1):68 – 82, 2011.

[GILO11b] Enrico Gregori, Alessandro Improta, Luciano Lenzini, and Chiara Orsini. The impact of IXPs on the AS-level topology structure of the Internet. In *Computer Communications, Volume: 34, Issue: 1*. Elsevier, 2011.

[GM] Phillipa Gill and Anirban Mahanti. Observations on round-trip times of tcp connections. In *Proc. SPECTS 2006.*

[goo17] Google ipv6 statistics, Jan 2017.

[GSP+12] Debayan Gupta, Aaron Segal, Aurojit Panda, Gil Segev, Michael Schapira, Joan Feigenbaum, Jenifer Rexford, and Scott Shenker. A new approach to interdomain routing based on secure multi-party computation. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, HotNets-XI, pages 37–42, New York, NY, USA, 2012. ACM.

[GVS+] Arpit Gupta, Laurent Vanbever, Muhammad Shahbaz, Sean P. Donovan, Brandon Schlinker, Nick Feamster, Jennifer Rexford, Scott Shenker, Russ Clark, and Ethan Katz-Bassett. Sdx: A software defined internet exchange. In *Proc. SIGCOMM 2014.*

[GVS+14] Arpit Gupta, Laurent Vanbever, Muhammad Shahbaz, Sean P. Donovan, Brandon Schlinker, Nick Feamster, Jennifer Rexford, Scott Shenker, Russ Clark, and Ethan Katz-Bassett. SDX: A software defined Internet exchange. *SIGCOMM Comput. Commun. Rev.*, 44(4):551–562, August 2014.

[GZ13] V. Giotsas and S. Zhou. Improving the discovery of IXP peering links through passive BGP measurements. In *Proc. INFOCOM*, 2013.

[Hew14] Hewlett-Packard Development Company. HP OpenFlow 1.3 administrator guide (wired switches K/KA/WB 15.16), Oct 2014.

[HEY08] H.M. Hassan, M. Eltoweissy, and M. Youssef. Towards a federated network architecture. In *Proc. INFOCOM Workshops*, 2008.

[Hom] `http://www.dhs.gov/critical-infrastructure-sectors`.

[ICA+11] G. Ibanez, J. A. Carral, J. M. Arco, D. Rivera, and A. Montalvo. ARP-path: ARP-based, shortest path bridges. *IEEE Communications Letters*, (15), July 2011.

[IMS13] A.S. Iyer, V. Mann, and N.R. Samineni. SwitchReduce: Reducing switch state and controller involvement in openflow networks. In *Proc. IFIP Networking Conference*, 2013.

[int] `http://www.internetsociety.org`.

[Ixi15] Ixia. IxANVL OpenFlow test suite, Dec 2015.

[JDK+16] Prerit Jain, Soham Desai, Seongmin Kim, Ming-Wei Shih, J Lee, Changho Choi, Youjung Shin, Taesoo Kim, Brent Byunghoon Kang, and Dongsu Han. Opensgx: An open platform for sgx research. In *Proceedings of the Network and Distributed System Security Symposium, San Diego, CA*, 2016.

[JHRB16] E. Jasinska, N. Hilliard, R. Raszuk, and N. Bakker. Internet exchange BGP route server. IETF draft-ietf-idr-ix-bgp-route-server-10, Apr 2016.

[Jur13] P. Jurkiewicz. Link modeling using ns-3, Sep 2013.

[KAK14] K. Kataoka, N. Agarwal, and A.V. Kamath. Scaling a broadcast domain of Ethernet: Extensible transparent filter using SDN. In *Proc. ICCCN*, Aug 2014.

[KGK15] M. Koerner, C. Gaul, and O. Kao. Evaluation of a cloud federation approach based on software defined networking. In *2015 IEEE 40th Local Computer Networks Conference Workshops (LCN Workshops)*, pages 657–664, Oct 2015.

[KHK13] Y. Kanizo, D. Hay, and I. Keslassy. Palette: Distributing tables in software-defined networks. In *Proc. INFOCOM*, 2013.

[KK14] Kyoungha Kim and Yanggon Kim. The security appliance to bird software router. In *Proceedings of the 8th International Conference on Ubiquitous Information Management and Communication*, page 37. ACM, 2014.

[KLRW13] Nanxi Kang, Zhenming Liu, Jennifer Rexford, and David Walker. Optimizing the "One Big Switch" abstraction in software-defined networks. In *Proc. CoNEXT*, 2013.

[KLS00] S. Kent, C. Lynn, and K. Seo. Secure border gateway protocol (s-bgp). *IEEE Journal on Selected Areas in Communications*, 18(4):582–592, April 2000.

[KMS$^+$09] Rupa Krishnan, Harsha V Madhyastha, Sridhar Srinivasan, Sushant Jain, Arvind Krishnamurthy, Thomas Anderson, and Jie Gao. Moving beyond end-to-end path information to optimize cdn performance. In *Internet Measurement Conference (IMC)*. ACM, 2009.

[KPK15] Maciej Kuźniar, Peter Perešíni, and Dejan Kostić. What you need to know about SDN flow tables. In *Proc. PAM*, 2015.

[KR07] K. Kompella and Y. Rekhter. Virtual Private LAN Service (VPLS) Using BGP for Auto-Discovery and Signaling. RFC 4761 (Proposed Standard), January 2007.

[KRV$^+$15] Diego Kreutz, Fernando M. V. Ramos, Paulo Esteves Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey. *Proc. of the IEEE*, 103(1):14–76, 2015.

[KS13] G. Khetrapal and S. K. Sharma. Demistifying routing services in software-defined networking, 2013. Aricent, Inc. White Paper.

[KSF14] Juhoon Kim, Nadi Sarrar, and Anja Feldmann. Watching the ipv6 takeoff from an ixp's viewpoint. *CoRR*, abs/1402.3982, 2014.

[KSH$^+$15] Seongmin Kim, Youjung Shin, Jaehyung Ha, Taesoo Kim, and Dongsu Han. A first step towards leveraging commodity trusted execution environments for network applications. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*, HotNets-XIV, pages 7:1–7:7, New York, NY, USA, 2015. ACM.

[LHD$^+$13] Matthew Luckie, Bradley Huffaker, Amogh Dhamdhere, Vasileios Giotsas, et al. AS Relationships, Customer Cones, and Validation. In *Internet Measurement Conference (IMC)*. ACM, 2013.

[LIJM$^+$10] Craig Labovitz, Scott Iekel-Johnson, Danny McPherson, Jon Oberheide, and Farnam Jahanian. Internet inter-domain traffic. In *Computer Communication Review, Volume: 40, Issue: 4*. ACM, 2010.

[LIN16] LINX. London internet exchange point route server members, Sept 2016.

[LLD$^+$14] Aemen Lodhi, Natalie Larson, Amogh Dhamdhere, Constantine Dovrolis, et al. Using peeringdb to understand the peering ecosystem. In *Computer Communication Review, Volume: 44, Issue: 2*. ACM, 2014.

[LO15] Bob Lantz and Brian O'Connor. A mininet-based virtual testbed for distributed sdn development. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 365–366. ACM, 2015.

[LRVB15] Gabriele Lospoto, Massimo Rimondini, Benedetto Gabriele Vignoli, and Giuseppe Di Battista. Rethinking Virtual Private Networks in the software-defined era. In *Proc. IM*, 2015.

[MAB$^+$08] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, 2008.

[MAB$^+$13] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, pages 10:1–10:1, New York, NY, USA, 2013. ACM.

[MHC] M.Z. Masoud, Xiaojun Hei, and Wenqing Cheng. A graph-theoretic study of the flattening internet as topology. In *Proc. 19th IEEE ICON, 2013.*

[min15] Mininet, Jan 2015.

[min17] Mininext, mininet extended, Mar 2017.

[MIP$^+$06] Harsha V Madhyastha, Tomas Isdal, Michael Piatek, Colin Dixon, Thomas Anderson, Arvind Krishnamurthy, and Arun Venkataramani. iPlane: An information plane for distributed services. In *Operating Systems Design and Implementation (OSDI).* USENIX, 2006.

[MIX] MIX - Milan Internet eXchange. `http://mix-it.net/`.

[MIX16] MIX. Milan internet exchange point members, Sept 2016.

[ML$^+$17] Habib Mostafaei, Gabriele Lospoto, , Roberto di Lallo, Massimo Rimondini, and Giuseppe Di Battista. SDNetkit: A testbed for experimenting sdn in multi-domain network. In *Workshop on Multi-Provider Network Slicing and Virtualization (MPNSV 2017)*, 2017.

[mla] MLab. `www.measurementlab.net`.

[MLB$^+$17] Habib Mostafaei, Gabriele Lospoto, Andrea Brandimarte, Roberto di Lallo, Massimo Rimondini, and Giuseppe Di Battista. SDNS: Exploiting sdn and the dns to exchange traffic in a federated network. In *Proc. IEEE Conference on Network Softwarization (NetSoft 2017)*, 2017.

[MNG14] H. M. B. Moraes, R. Nunes, and D. Guedes. DCPortalsNg: Efficient isolation of tenant networks in virtualized datacenters. In *Proc. COCORA*, 2014.

[MRF$^+$13] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing Software-Defined Networks. In *Proc. NSDI*, 2013.

[MRWK03] Zhuoqing Morley Mao, Jennifer Rexford, Jia Wang, and Randy H Katz. Towards an accurate as-level traceroute tool. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 365–378. ACM, 2003.

[NaM] NaMeX - Nautilus Mediterranean eXchange. `http://namex.it/`.

[nat] NATO Federated Mission Networking.

[net17] Netkit: The poor man's system to experiment computer networking, Jan 2017.

[nex11] Cisco nexus 7000 series nx-os system management configuration guide. Technical report, Cisco Systems Inc., 2011.

[NHL$^+$13] Yukihiro Nakagawa, Kazuki Hyoudou, Chunghan Lee, Shinji Kobayashi, Osamu Shiraki, and Takeshi Shimizu. DomainFlow: Practical flow management method using multiple flow tables in commodity switches. In *Proc. CoNEXT*, 2013.

[NMN$^+$14] B.AA Nunes, M. Mendonca, Xuan-Nam Nguyen, K. Obraczka, and T. Turletti. A survey of software-defined networking: Past, present, and future of programmable networks. *Communications Surveys Tutorials, IEEE*, 16(3):1617–1634, Third 2014.

[Nov16] NoviFlow, Inc. NoviNID 106 OpenFlow 1.3 CPE Switch, Jan 2016.

[NSBT14] Xuan-Nam Nguyen, Damien Saucez, Chadi Barakat, and Thierry Turletti. Optimizing rules placement in OpenFlow networks: Trading routing for better efficiency. In *Proc. HotSDN*, 2014.

[Ope13a] Open Networking Foundation. Conformance test specification for OpenFlow switch specification 1.0.1, Jun 2013.

[Ope13b] Open Networking Foundation. OpenFlow switch specification, version 1.3.3, Sep 2013.

[Ope14a] Open Networking Foundation. Migration use cases and method, 2014.

[Ope14b] Open Networking Foundation. OpenFlow Switch Specification, version 1.3.4, Mar 2014.

[Ope15a] Open Networking Foundation. OpenFlow conformance test program, Jan 2015.

[Ope15b] Open Networking Foundation. OpenFlow switch specification, version 1.5.1, Apr 2015.

[Ope16a] Open Networking Foundation. ONF OpenFlow conformant: Certified product list, Jan 2016.

[Ope16b] OpenSGX. Opensgx: An open platform for intel sgx, Sep 2016.

[ovs15] Open vSwitch, Sep 2015.

[pch] `www.pch.net/home/index.php`.

[Pee12] Dr. Peering. Peering policy clauses collected from 28 companies. Internet Peering Workshop, 2012.

[PGP$^+$] Milosz Przywecki, Ivana Golub, Darko Paric, Maribel Cosin, Paul van Daalen, Gerben van Malenstein, Peter Kaufmann, Ralf Paffrath, and Jari Miettinen. Federated pop: A successful real-world collaboration.

[Plu82] D. Plummer. Ethernet Address Resolution Protocol. IETF RFC 826, 1982.

[Pro16] Project Floodlight. OFTest, Jan 2016.

[rip] RIPE Stat. `http://stat.ripe.net`.

[rip17] Ripe ipv6 statistics, Jan 2017.

[RLH06] Y. Rekhter, T. Li, and S. Hares. A Border Gateway Protocol 4 (BGP-4). RFC 4271 (Draft Standard), January 2006.

[RR06] E. Rosen and Y. Rekhter. BGP/MPLS IP Virtual Private Networks (VPNs). RFC 4364, February 2006.

[RSF$^+$14a] Philipp Richter, Georgios Smaragdakis, Anja Feldmann, Nikolaos Chatzis, Jan Boettger, and Walter Willinger. Peering at Peerings: On the Role of IXP Route Servers. In *Internet Measurement Conference (IMC)*. ACM, 2014.

[RSF$^+$14b] Philipp Richter, Georgios Smaragdakis, Anja Feldmann, Nikolaos Chatzis, Jan Boettger, and Walter Willinger. Peering at peerings: On the role of ixp route servers. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC '14, pages 31–44, New York, NY, USA, 2014. ACM.

[RSU$^+$12] Charalampos Rotsos, Nadi Sarrar, Steve Uhlig, Rob Sherwood, and Andrew W. Moore. OFLOPS: An open framework for Openflow switch evaluation. In *Proc. PAM*, 2012.

[rus] `http://arstechnica.com/security/2014/11/wtf-russias-domestic-internet-traffic-mysteriously-passes-through-china/`.

[Ryu15] Ryu project team. Ryu SDN framework, Mar 2015.

[Ryu16] Ryu SDN Framework Community. OpenFlow switch certification, Jan 2016.

[ryu17] Ryu: component-based software defined networking framework, Jan 2017.

[sam] SamKnows. `http://samknows.com`.

[SF14] N. Shen and D. Farinacci. LISP Multi-Provider VPN Use-Cases, July 2014.

[SH99] P. Srisuresh and M. Holdrege. IP Network Address Translator (NAT) Terminology and Considerations. IETF RFC 2663, August 1999.

[SLP$^+$15] Keith Stouffer, Suzanne Lightman, Victoria Pillitteri, Marshall Abrams, and Adam Hahn. Guide to industrial control systems (ics) security – nist special publication (sp) 800-82 revision 2. Technical report, NIST, 2015.

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

[Spe13] OpenFlow Switch Specification. Version 1.3.3 (wire protocol 0x04), Sept 2013.

[Spi16] Spirent Communications, Inc. Spirent TestCenter – OpenFlow switch compliance test suite, Jan 2016.

[Sup16] Cisco Support. Configuring and verifying the bgp conditional advertisement feature, Sep 2016.

[SVR15] Peng Sun, Laurent Vanbever, and Jennifer Rexford. Scalable programmable inbound traffic engineering. In *Proc. SOSR*, 2015.

[SW09] Yuval Shavitt and Udi Weinsberg. Quantifying the importance of vantage points distribution in internet topology measurements. In *INFOCOM*. IEEE, 2009.

[TCB08] Steven Tom, Dale Christiansen, and Dan Berrett. Recommended practice for patch management of control systems. *DHS control system security program (CSSP) Recommended Practice*, 2008.

[Tec16] Juniper TechLibrary. Configuring conditional installation of prefixes in a routing table, Apr 2016.

[TS14] K. M. Tankala and S. K. Sing. Address resolution in software-defined networks. Patent no. WO2014115157 A1, 2014.

[Uni16] Roma Tre University. Computer networks research groups, Sept 2016.

[Ver14] Veryx Technologies. ATTEST OpenFlow switch conformance test suite, Dec 2014.

[WCY13] Shie-Yuan Wang, Chih-Liang Chou, and Chun-Ming Yang. Estinet openflow network simulator and emulator. *IEEE Communications Magazine*, 51(9):110–117, 2013.

[WDS+14] Philip Wette, Martin Draxler, Arne Schwabe, Felix Wallaschek, Mohammad Hassan Zahraee, and Holger Karl. Maxinet: Distributed emulation of software-defined networks. In *Networking Conference, 2014 IFIP*, pages 1–9. IEEE, 2014.

[XLZ15] Y. Xu, X. Lu, and T. Zhang. A novel efficient SDN based broadcast scheme. In *Proc. CSIC*, Jun 2015.

[YRFW10] Minlan Yu, Jennifer Rexford, Michael J. Freedman, and Jia Wang. Scalable flow-based networking with DIFANE. *SIGCOMM Comput. Commun. Rev.*, 41(4), August 2010.

[ZZG+12] Mingchen Zhao, Wenchao Zhou, Alexander J.T. Gurney, Andreas Haeberlen, Micah Sherr, and Boon Thau Loo. Private and verifiable interdomain routing decisions. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, pages 383–394, New York, NY, USA, 2012. ACM.