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

Improving privacy, provisioning, and scalability in inter-domain networks

Habib Mostafaei

Improving privacy, provisioning, and scalability in inter-domain networks

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

Roma Tre University
Dept. of Engineering

Spring 2019

Committee:
Prof. Giuseppe Di Battista

Reviewers:
Prof. Antonio Pescapé (University of Naples Federico II)
Prof. Marco Chiesa (KTH Royal Institute of Technology)

To Arezou

Acknowledgments

First and foremost I thank Allah, the Lord of the worlds, the compassionate, and the merciful, for empowering me to complete this work.

I express my deepest gratitude to my Advisor, Professor Giuseppe Di Battista, who gave me the opportunity to join computer networks group of Roma Tre University and do research in the field of inter-domain routing. He taught me many research concepts of computer networks and supported my transition to new research field. He always has great ideas on the real networking problems.

A special thank goes to Gabriele Lospoto who was a great friend and almost like a second advisor for me. Learning and implementing new concepts in our research works went very fast with his invaluable feedbacks. I also worked with Massimo "Max" Rimondini for the first eight months of my PhD. Max is a great researcher and friend.

I am glad to have had opportunity to collaborate with other professors of compunet lab like Professor Maurizio Pizzonia and Professor Maurizio Patrignani. I would like to thank them.

I am very thankful to Professor Antonio Pescapé which I did research with him during the first year of my PhD education and even a bit earlier. We had very fruitful collaborations. I am also very thankful to Professor Marco Chiesa which I did PrIXP and DeSI projects with him. I appreciate both professors for reviewing this thesis. Your comments helped me improve the quality of this work.

I am very thankful to Professor Michael Menth for his hospitably during my visiting stay at the University of Tuebingen. He was my advisor the last year of my PhD and we worked on programmable networks. We had very fruitful and impressive discussions and collaborations. He taught me how to criticize the results of a running algorithm and the context of a research paper before submitting it to a venue.

I am very thankful to my Italian friends at compunet and MICLab labs who helped me in doing my bureaucratic with different offices within and out of university. They had to have several phone calls to solve my issues. Federico Griscioli, Vincenzo Roselli, Giordano Da Lazzo, Valentino Di Donato, Cosimo Pallazo, Patrizio Angelini. There is a great atmosphere at compunet lab with its members. We had great time together specially during lunch and coffee time and I would like to thanks Marco Di Bartolomeo, Diego Penino, Roberto Di Lallo, Professor Fabrizio Frati.

I would like to thank the members of chair of Communication networks at the university of Tuebingen, Daniel Merling, Fredrick Hauser, Mark Schmidt, Andreas Stockmayer, and Wolfgang Braun. We had several fruitful discussions during the time that I was in Tubingen.

Finally, the loveliest thank goes to my wife Arezou who stayed with me during the whole period of my PhD education. She had to often share her time with me to finish my ongoing projects. Arezou was extremely supportive and patient.

Contents

Contents **ix**

List of Tables **xii**

List of Figures **xiii**

1 Introduction **1**
1.1 Contributions 2
1.2 Outline 4

2 Background **5**
2.1 Inter-domain Networks 5
2.2 Inter-domain Routing 5
2.3 Route Server 8
2.4 Border Gatewy Protocol (BGP) 9
2.5 Congestion Control 10
2.6 Network Emulation 12
2.7 Software-Defined Networking (SDN) 12

3 Privacy of Routing Policies at IXPs **15**
3.1 Introduction 16
3.2 Background: Route Server Architecture 17
3.3 Enforcing Privacy of Routing Policies 21
3.4 Discussion on Security Issues 24
3.5 Experiments 26
3.6 Related Work 29
3.7 Conclusions and Future Works 31

4 Experimenting SDN in Inter-Domain Networks **33**
4.1 Introduction . . . 34
4.2 Related Work . . . 36
4.3 From Netkit to SDNetkit . . . 38
4.4 A Simple Example and Success Stories . . . 40
4.5 Configuration Considerations . . . 45
4.6 Limitations of Network Emulators . . . 47
4.7 Conclusions and Future Work . . . 48

5 Multi-Provider VPNs in Software-Defined Federated Networks **49**
5.1 Introduction . . . 50
5.2 Related Work . . . 52
5.3 Best Practices for Federated Networks . . . 54
5.4 SDN-based Federated Networks . . . 55
5.5 Subscribing to a SDN Federated VPN Service . . . 60
5.6 A Complete Example . . . 69
5.7 Takeaway . . . 76
5.8 Evaluation . . . 77
5.9 Conclusions and Future Work . . . 83

6 A Decentralized SDN Architecture for IXPs **85**
6.1 Introduction . . . 86
6.2 Related work . . . 88
6.3 SDX based IXP Architectures . . . 91
6.4 A New SDN Architecture for Internet eXchange Points . . . 93
6.5 A Routing Policy Model . . . 95
6.6 From Policies to Forwarding Rules . . . 98
6.7 The Architecture of our SDN-controller . . . 107
6.8 Applicability Considerations . . . 109
6.9 Evaluation . . . 111
6.10 Conclusion . . . 117

7 Activity-based Congestion Management (ABC) **121**
7.1 Introduction . . . 121
7.2 Related Work . . . 122
7.3 Activity-Based Congestion Management (ABC) . . . 123

7.4 SDN and Data Plane Programmability Using P4 126
7.5 P4-Based Implementation of ABC 128
7.6 Evaluation Methodology 131
7.7 Performance Evaluation 132
7.8 Conclusion 135

8 Conclusions **137**

List Of Publications **141**

Bibliography **145**

List of Tables

4.1 A comparison among simulation systems in terms of type (e.g. centralized, distributed, or remote testbed) and supported OpenFlow versions. 36

6.1 Comparison of SDN-based solutions for inter-domain routing. . . . 89

List of Figures

2.1 The interconnection of two domain networks via a private peering link. 6
2.2 Interconnection of domains at an Internet eXchange Point (IXP). . 7
2.3 IXP architecture with a route server. 8
2.4 Congestion occurs when the traffic exceeds the network capacity. . 11

3.1 Reference architecture of route server. 19
3.2 The architecture of an IXP infrastructure. 23
3.3 Architecture for checking the integrity of the RS software. 25
3.4 Architecture of our RS-software prototype implementation. 27
3.5 CDF of the number of messages issued by the RS-software. 28
3.6 CDF of the number of messages issued by members. 28

4.1 Overview of the SDNetkit architecture. 39
4.2 Topology for the Hybrid node. `S1` is the device in which OpenFlow and OSPF simultaneously run. 41
4.3 Topology for the Hybrid topology. `S1` and `S2` are pure OpenFlow-enabled switches, whereas all other devices are IP-speaking nodes. 43
4.4 Topology used to experiment a single network for forwarding both control and standard traffic. 46

5.1 Reference scenario for SDNS. 57
5.2 Logical architecture of our SDN-controller, consisting of a set of components each devoted to a specific task. 61
5.3 An example of insert primitive. 68
5.4 Interaction among OpenFlow switch, name servers, and SDN-controller in case of *Partial configuration* scenario. 71

5.5 Messages exchanged to support communication between H1 and H2, for different translation strategies. 75
5.6 Control plane impact in a federated network consisting of two ISPs. 79
5.7 Control plane impact in a federated network consisting of three ISPs. 80
5.8 Number of queries in the federated network. 81

6.1 Our architecture in which SDN is moved in the provider's side in order to avoid policies sharing and to easily identify responsibilities. 94
6.2 Graph representation of a policy only containing the *AND* operator. 102
6.3 Tree representation of a policy containing the *OR* operator. 103
6.4 Example of how to use metadata to encode network reachability information. 105
6.5 High level view of the internal architecture of our SDN-controller. 107
6.6 A detailed view of the internal architecture of the PolicyHandler. . 108
6.7 The topology used for the functionality tests. It simulates a simple IXP consisting of four members, each announcing just one prefix. . 112
6.8 Percentage of consumed RAM in the proactive approach with a growing number of BGP announcements. 113
6.9 Percentage of consumed RAM in the reactive approach with a growing number of BGP announcements. 114
6.10 Percentage of consumed RAM at C2 in both approaches increasing the number of policies. 116
6.11 Percentage of consumed RAM in the proactive approach with a growing number of BGP announcements. 117
6.12 Percentage of consumed RAM in the reactive approach with a growing number of BGP announcements. 118
6.13 Time analysis about the time spent by DeSI to perform its activity. 119

7.1 Activity metering and tagging is performed only by ingress nodes. Both ingress and core nodes apply activity AQM during packet forwarding. Figure taken from [MZ18]. 124
7.2 P4 processing pipeline. 127
7.3 Resource sharing of Client 0 and Client 1, both sending CBR traffic. 133
7.4 Resource sharing of Client 0 and Client 1, both sending TCP traffic. 135
7.5 Resource sharing of Client 0 and Client 1, Client 0 sends CBR traffic while Client 1 has 1 TCP connection. 136

Chapter 1

Introduction

Communications in today's network and on the Internet spans over different networks. An inter-domain network consists of several individually controlled domains in which each of them has its own physical infrastructure and routing protocols. There is a central control for each domain which takes the control of the whole network. However, several domain networks interconnect to exchange their traffic based on business agreements. Each domain network in the context of this work can be considered as an Internet Service Provider (ISP). Each ISP may connect its network to an Internet eXchange Points (IXPs) to improve its connectivity benefiting from peering to the IXP.

Within the network, new technologies evolved during the last years providing more flexible control on the network operations and efficiently exploiting the network resources. One of the most promising technologies is Software-Defined Networking (SDN) [The13] which is seen as an enabler to program the network behavior using a centralized controller. SDN promises several advantages such as flexibility in controlling the network behavior without downgrading forwarding performance, efficiency in routing optimization, facilitating implementation and administration, and cost reduction. Accordingly, inter-domain networks can leverage the advantages of SDN to issue new network services with respect to customers' demands. Further, recent innovations in data plane programming add more advantages like fast packet processing to the networking world.

Network services may span over multiple providers' network. Issuing and provisioning such a service over the network of different providers comes with several challenges due to the diversity of network devices and protocols. This

makes the network provisioning a difficult task which costs at least in terms of time and physical resources. Furthermore, the traffic of network services may cross several IXPs which is prone to revealing business agreements of members to a third party. We observe that the privacy of routing policy at IXPs is not guaranteed despite the rapid innovation in networking. The same issue exists for the IXPs that are built based on SDN concepts called Software-Defined eXchange (SDX) [GVS+15, GMB+16]. Despite SDX benefits, its architectures come with several shortcomings like privacy and scalability. SDX exploits the controller to perform route computations for the members which scales poorly when the number of members grows. The focus of this thesis is on the research problems that are resided within the context of inter-domain networks.

1.1 Contributions

This work focuses on the privacy, provisioning, and scalability of inter-domain networks which is resulted in five contributions aiming at improving them.

The contribution of this thesis starts with an overview of privacy challenges on IXPs and its architecture with Router Server (RS). Members of an IXP with RS can have one Border Gateway Protocol (BGP) multi-lateral peering instead of having several BGP bi-lateral peerings. However, the privacy of routing policies at IXPs is not preserved. This may reveal the Service Level Agreements (SLAs) or business relationships among the member of IXPs to a third-party. We propose a new RS implementation for RS-software at IXP which does not require to permanently store the routing polices at RS-machine. Additionally, we provide a mechanism for the members to attest the RS-software running on RS-machine. Such a mechanism allows the members to check the correctness of RS-software running on an RS-machine. Our simulation results showed that our architecture adds a negligible overhead to current IXP to improve the privacy.

The second contribution of this work is called SDNetkit which is an emulator to experiment SDN in multi-domain or inter-domain networks. This emulator is based on widely used Netkit emulator which empowers it with SDN-related software like Ryu [ryu17] framework and OpenVswitch [ovs17]. It removes the limitations imposed by current SDN simulators and emulators in creating an inter-domain network scenarios. SDNetkit creates a Linux-based Virtual Machine (VM) for each network device. Each VM of SDNetkit has all required routing suits and daemons to run an inter-domain network. We show several use-cases for SDNetkit.

The third contribution starts with overview of challenges for offering a new network service like Virtual Private Network (VPN) in the federated networks and how we can overcome these challenges by leveraging the concept of SDN. Issuing new network services like VPN service spanning on more than two networks demands several challenges like long setup time and complex configurations. These issues are referred to provisioning. Best practices to reach such goals demand long setup time, i.e., 5 days. We leverage SDN to overcome such challenges in SDN-enabled federated networks. We propose a framework to allow the customers of a federated network to create a federated VPN service by using the current backbone network. To do so, a configuration language with three simple yet effective primitives is proposed. The members of each customer exploit these primitives to join or leave a federated VPN service. The members of a federated VPN service exploit the Domain Name System (DNS) to quickly set a federated network. The framework uses the available backbone network to issue federated VPN service to save cost. The SDN-controllers for each member of a federated VPN service setups the required configuration for connecting the members.

The fourth contribution of this work relies on proposing a decentralized architecture for a Software-Defined eXchange (SDX) or SDN-based IXP to improve the privacy of routing policies and scalability. The current SDX solutions suffer from privacy and scalability issues. We propose a policy language for the members of SDX to define fine-grained routing policies for their traffic. The language takes high level user-defined routing polices and generates low-level forwarding rules to install on top of SDN-enables switches of each member at IXP. The language is able to find the dependency among the forwarding rules to allow the traffic of members to be forwarded to right destination. The solution is able to leverage non-BGP best paths to balance the traffic of participants. Furthermore, the proposed architecture provides backward compatibility for current IXPs meaning that it does not modify the IXP fabric.

The final contribution of this work is the implementation of Activity-based Congestion management (ABC) in the data plane programming language, e.g., P4. The object of this work is to introduce ABC as a congestion management for future networks as well as a use-case for P4. The ABC does not require any signaling in core nodes of the network and it can handle the congestion in the network without storing per-packet or per-flow information. Users can maximize their throughput by sending traffic at their fair share. ABC prefers to drop the packets of heavy users over light user in the case of congestion.

1.2 Outline

This thesis is structured as follows. Chapter 2 contains general concepts in inter-domain networks including its definition, related technologies, and protocols. Chapter 3 introduces the basic building blocks of an IXP and explains its privacy issues. We propose an architecture to preserve the privacy of routing policies at IXP. Chapter 4 overview the shortcoming of SDN-based simulators and emulators to perform an SDN-based experiment on multi-domain networks. Then, it introduces SDNetkit in ch as a promising solution to overcome the limitations of current de-facto emulator for SDN, i.e. Mininet. Chapter 5 proposes an SDN-based framework to issue Virtual Private Network (VPN) services in federated networks. We show how SDN can decrease the required time to offer a VPN service spanning over multiple providers' network. Chapter 6 introduces a decentralized architecture for an SDN-based exchange point. We first give an overview of current SDX architecture and then we explain the privacy and scalability issues of these architectures. Lately, we introduce DeSI as a solution to overcome the privacy and scalability issues of SDXs. The investigations of physical resource consumption revel that DeSI requires less physical resources to handle the IXPs' operations. We provide the P4 implementation of ABC in chapter 7. We briefly introduce the basic concepts of SDN and P4 in this chapter. Then, we explain how ABC can be implemented in the data plane. The simulation results on a P4-enabled switch depict that ABC can protect traffic of light users over heavy users. Chapter 8 concludes this work.

Chapter 2

Background

In this chapter, we give an overview of the basic concepts and technologies in inter-domain networks. We explain *inter-domain* routing concept and Route Server. We briefly describe the Border Gateway Protocol (BGP) and its decision process since it is the only inter-domain routing protocol. Then, we state congestion control and network emulation. Finlay, we describe the concept of Software-Defined Networking (SDN) and its related technologies and protocols.

2.1 Inter-domain Networks

The Internet consists of thousands of interconnected networks called domains. Each domain consists a set of network devices, e.g., routers, and hosts which are managed by an independent organization called Autonomous System (AS) [Bon11]. Google, cisco, and level3 are some example domains to state a few. The communications among the devices of these networks are packet based and routing protocols are in charge of computing paths for each network. Each domain network implements its own routing policies on the Internet.

2.2 Inter-domain Routing

The inter-domain routing is in charge of computing and propagating routes among different domain networks. However, the protocols that are leveraged to compute and distribute the routes inside a domain are called intra-domain routing protocols. In this work we focus on inter-domain routing. Each domain network consists of several routers which are responsible to maintain the routing

information. Border routers are used to connect different domain networks on the Internet.

The connection between two border routers can be performed in two different ways. First, the two border routers can be directly connected through a dedicated link called private inter-domain link or private peering link [Bon11]. Fig. 2.1 shows a private peering link among domains A and B via their border routers, namely, R1 and R2. This type of connection is useful when an organization or a university connects its network to an ISP. Second, a domain network may connect to several domains. Having many private peering links results in increasing the cost for the domain. A cheap solution for this problem is connecting to an Internet eXchange Point (IXP). An IXP is a neutral physical place that many providers connect their network to exchange their traffic at a shared-cost and better connectivity. Fig. 2.2 depicts an IXP with three domain networks. Each border router belonging to a domain is connected to the IXP via a physical link. Each magenta dotted link illustrates a peering link between two different domains.

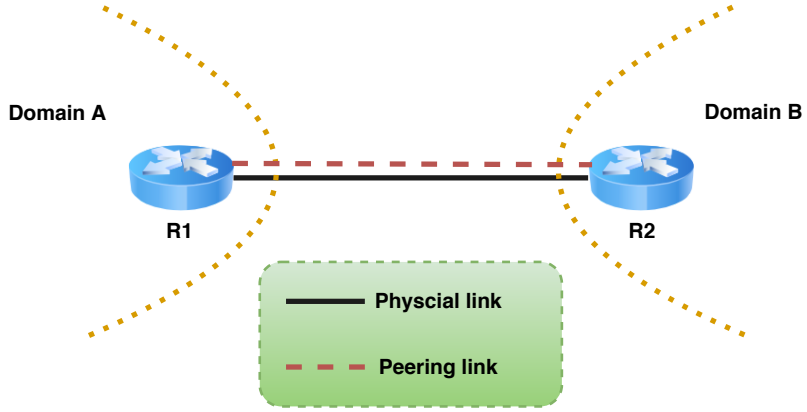


Figure 2.1: The interconnection of two domain networks via a private peering link.

When a domain is willing to peer with an IXP, it brings its own router to IXP and connects its border router to the switch fabric of an IXP. An IXP consists of a Local Area Network (LAN) that is composed of routers of different IXP's participants. When a domain wishes to exchange traffic with other domains it uses leverages this LAN. A packet-based mechanism is

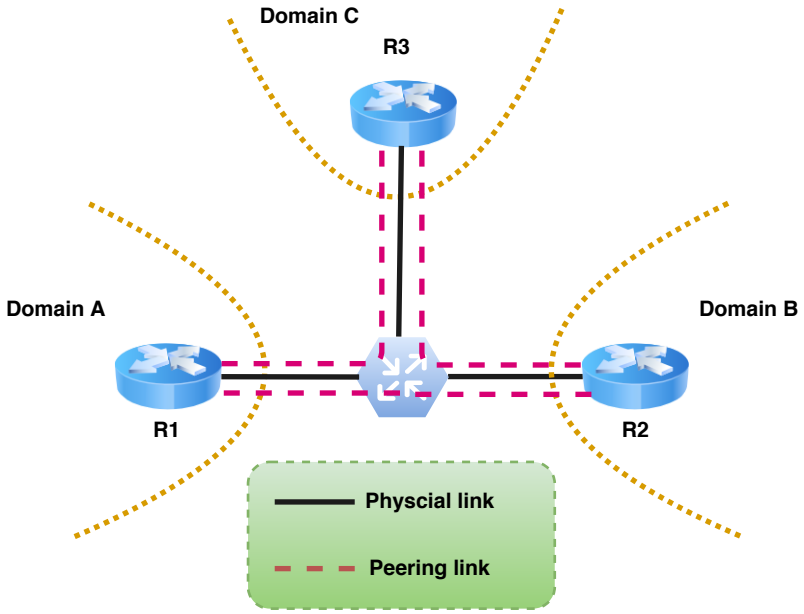


Figure 2.2: Interconnection of domains at an Internet eXchange Point (IXP).

exploited for exchanging the routing information.

Inter-domain routing requires to take care of economical agreements or Service Level Agreements (SLAs) among the domains and the cost of a route is important than other quality-related metrics like bandwidth and delay. These economic relationships or SLAs are converted to peering relationships among the domains in inter-domain routing. A detailed discussion for different types of relationships can be found in [Bon11]. The relationships among various domains in inter-domain routing are defined by inter-domain routing policies [Bon11]. A routing policy consists of three main items, namely, import, export, and ranking algorithm. An import filter specifies the accepted routes by a domain while an export filter determines the routes that can be advertised by a domain. The ranking algorithm determines the best route to a destination prefix based on the available routes to that prefix within a domain.

2.3 Route Server

Members of the same domain in inter domain routing require to establish a full mesh connectivity to exchange traffic with other domains. This direct peering allows the member to acquire the information that are reachable by adjacent routers and select optimum path to reach a destination. Unfortunately, maintaining a full mesh routing connectivity information on the switch with large number of routers is impractical [Has95]. To alleviate the full mesh problem RFC 1863 proposes to user route server. Typically, IXPs offer a service called Route Server (RS). An RS allows the member to easily exchange their traffic with other members by establishing a peering session with RS instead of having several peering sessions with each other.

Fig. 2.3 depicts the architecture of an IXP with three members and a RS. Each dotted line presents a BGP peering session between the member router and the RS. However, RFC 7947 [JHRB16b] describes the implementation of RS within an IXP.

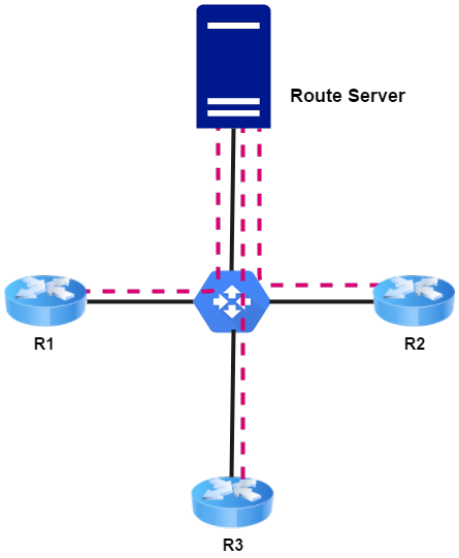


Figure 2.3: IXP architecture with a route server.

The routing information of members of an IXP are stored in suitable data structures which are used for path computations by the RS. The RS performs

path computations based on received paths from the neighbors and their routing policies. Then, it performs the path selection and finally propagates the best path to the neighbors.

Each IXP member can specify the *export policy* for each destination IP prefix, i.e., the set of IXP members that are allowed to receive it announcements. The export policy of each member must be revealed to the IXP which is supposed confidential due to business reasons. Due to privacy concerns some network operators are reluctant to connect to the IXPs [CDC+17].

2.4 Border Gatewy Protocol (BGP)

The Internet relies on a single inter-domain routing protocol which is called Border Gateway Protocol (BGP) [Bon11]. The current version BGP is defined in RFC4271 [RLH06]. The routing information in BGP is exchanged via establishing a BGP session with a border router of an AS which is called BGP speaker. The end-points of a BGP session are called BGP peers. A BGP session is a Transport Control Protocol (TCP) connection on port 179 of two BGP speakers. The BGP error notification mechanism assumes that TCP supports graceful close, i.e., all the outstanding data will be successfully delivered to the destination before the connection is close [RLH06].

A BGP speaker advertises a route toward a prefix, more specifically, it announces the IP prefix and the inter-domain path to reach that prefix. The inter-domain path is called AS-path in BGP context. When a BGP speaker advertises an IP prefix it will receive traffic from that IP prefix.

Each BGP speaker belongs to a unique AS which has an integer number called Autonomous System Number (ASN). After establishing a BGP session, two BGP speakers can exchange BGP messages which contain reachability information. The reachability information contains the IP address of systems that are reachable by a BGP speaker. The Network Layer Reachability Information (NLRI) field of a BGP update message carries this information.

Each BGP speaker contains three different data structures to store the Routing Information Bases (RIBs), namely,

- Adj-RIBs-In. The Adj-RIBs-In contains the incoming routes from BGP peers. These routes are available as input for the BGP decision process.

- Local Routing Information Base (Loc-RIB). The Loc-RIB stores all acceptable routes for this router.

- Adj-RIB-Out. The Adj-RIB-Out stores the BGP routes that have been advertised to each BGP peers.

The routing information that BGP speakers use to forward packets or construct the forwarding tables to forward packet are stored in routing tables. These tables accumulate by the routes that are: 1) directly connected networks, 2) static routes, 3) and routes learned from Interior Gateway Protocol (IGP) and BGP protocols. The routing information can be exchanged by other BGP speakers via BGP update messages. Here, we explain just the basic concepts of BGP that are useful for this thesis. More detail about the BGP, its features, it decision process can be found in [Bon11, RLH06].

The BGP Decision Process

The decision process selects the routes by applying the routing policies. The output of this process is a set routes that will be advertised to peers. A key different between BGP and inter-domain routing is that each domain can select its own preferred route to a destination prefix based on learned routes from toward that prefix [Bon11]. Import and export filters and ranking algorithm has impact on route selection in the BGP decision process.

There are several attributes that are used to rank the BGP routes like *local-pref* and *AS-path* to mention a few for the context of this work. The *local-pref* is an attribute that specifies an integer number. This number is used to prefer a route over other routes. The *AS-path* is an attribute that determines the ASes that announced a prefix and corresponds to a route to a destination. More information about the attributes for BGP route selection can be found in [RLH06, Bon11]. Among all available paths to a specific destination, the BGP decision process selects one path as the best BGP path to that prefix and announces the path to peers.

2.5 Congestion Control

When the number of packets in the network is higher than the capacity of the network it causes packet delay and loss resulting in performance degradation. This kind of situation called congestion [TW96]. The duration of congestion can vary from milliseconds to hours. Consequently, the impact of congestion on the applications depends on the duration and severity of congestion and the application design to handle congestion [G+13]. If the duration of congestion is short enough or the application can tolerate the congestion the user may not

experience any performance degradation. Therefore, congestion is a problem when its duration is long.

Fig. 2.4 depicts the onset of a congestion. When the number of sent packets is equal to or less than the carrying capacity of the network, the packet delivery is proportional to the number of sent packets. However, when the traffic load approaches the network carrying capacity, burst of traffic fills up the routers' buffer and some packets are lost. The lost packets consume the network capacity, therefore the number of delivered packets is less than ideal. In this circumstance the network is congested.

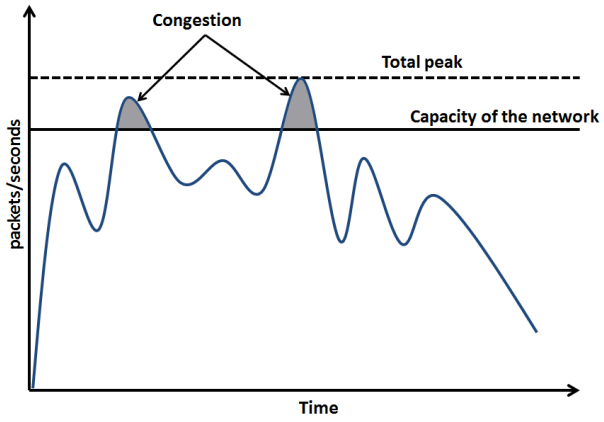


Figure 2.4: Congestion occurs when the traffic exceeds the network capacity.

The network and transport layers are responsible for handling congestion in the network. Since the circumstance happens in the network, the network layer experiences it and should decide what to do with the excess packets. However, the most effective way to control the congestion is reducing the traffic load. This requires the collaboration of network and transport layers. The principles of congestion control are defined in RFC 2914 [Flo00]. More detailed discussion on various approaches to control congestion can be found in [TW96, Bon11]. However, all transport protocols in Internet should address congestion control.

The Transmission Control Protocol (TCP) is one of the widely used transport protocols on the Internet which is a highly reliable host-to-host protocol in packet-switched computer communication networks [APB09, Pos81]. TCP defines several different congestion control algorithms. RFC 5681 describes

each of TCP congestion control algorithms in more detail [APB09].

2.6 Network Emulation

Testing network protocols and applications has been a difficult task for years and current advances in software and hardware makes it even more complicated. The network emulation rapidly gains the attention of researchers, teachers, and network administrators in simulating the behavior of a real network due to ease of use and low cost [PR08]. With network emulators, the administrators can quickly set up a network configuration to check its functionality based on the expected behavior. The researchers can validate their theoretical models in practical network environments. The teachers can educate the students how to configure a network on a Personal Computer (PC) for education purposes.

A network emulator is different from simulator, which allows the users to compute the evolution of a network. More specifically, the emulator allows the user to produce the behavior of real network devices [PR08]. It consists of a set of software/hardware platform which enables running of the same software on the real network devices. Typically, in an emulator the network testing is performed by exchanging a set of packets between devices in such a way that they occur on real networks.

2.7 Software-Defined Networking (SDN)

Software-Defined Networking (SDN) is a networking paradigm that changes the way of controlling the network devices. SDN refers to the ability of software applications to dynamically program the forwarding behavior of network devices [KRV+15b]. Each network device, e.g., a router or a switch, has two main planes, namely, control plane and data plane. The forwarding decisions are made by the control plane while data forwarding is performed using the data plane. In traditional (IP) networks these plane were coupled and it was difficult to mange the network. SDN separates them. The control plane is controlled via a centralized unit, which is called *controller*, via Application Programming Interfaces (APIs). Afterward, the network devices act based on installed forwarding rules by the controller. The OpenFlow [MAB+08] is the most notable API.

Each SDN-enabled network with OpenFlow protocol has three main components [Ope18]; *flow table,secure channel*, and *OpenFlow protocol*. Each device in OpenFlow context, i.e., datapath, has one or more *flow tables*. Each flow

table performs packet lookup and forwarding. The OpenFlow switch communicates with the controller via one or more *secure channels*. The controller manages each *OpenFlow* switch via the secure channel. Each flow table contains a set of flow entries. Each flow entry has three main fields; *match fields*, *counters*, and a set of *instructions* to match the packets. If a match occurs with an incoming flow, the associated instructions to that match will be executed. If the incoming flow does not match with a flow entry of a flow table, the outcome depends on the configuration of table-miss flow entry. Therefore, the outgoing flow depends on the matching field and its action.

SDN controller in such networks has the following tasks [The13]. 1) Translating of SDN application layer requirements to SDN datapath. 2) Providing a global view of the network to the SDN applications.

Controlling the network with a centralized unit brings several advantages like easily programming the network applications or leveraging the network information by all applications to take decisions [KRV+15b]. Initially, it was supposed that there is a single controller to control the forwarding behavior of the whole network's devices. An SDN network may have more than a single controller for scalability, performance, or robustness reasons [WZHW17].

Data Plane Programmability

Programming the network through the controller facilitates the network management, but it limits the flexibility of the controller for future protocols to support. To program a network through a controller with new features the following steps are required. First, the features should be supported by the hardware. Second, a suitable API should be provided for the controller to program hardware. These may result in long production delay. Furthermore, the communication links between the controllers and devices may result in packet processing delay. These limitations can be removed by data plane programming.

Data plane programming has been recently introduced to facilitates innovation in networking [BDG+14]. It enables the definitions of new protocols and their behavior without waiting for the vendors to offer them.

Programming Protocol-Independent Packet Processing (P4) [BDG+14] is a data plane programming language which is introduced for these definitions. Programming the network in P4 is not limited to application-specific integrated circuit (ASIC) any more. The language is hardware agnostic aiming at fast innovation in networking protocols and fast packet processing. The features provided by P4 have the potential to shape the future of networking.

Chapter 3

Privacy of Routing Policies at IXPs*

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

*Part of contexts in this chapter is based on the following publication: Chiesa, M., di Lallo, R., Lospoto, G., Mostafaei, H., Rimondini, M. and Di Battista, G., 2017, May. PrIXP: Preserving the privacy of routing policies at Internet eXchange points. In Integrated Network and Service Management (IM), 2017 IFIP/IEEE Symposium on (pp. 435-441). IEEE.

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.

3.1 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). 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.

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. [AI16,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 [AI16]), 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. 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 chapter, we present PrIXP, an 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 this chapter is organized as follows. In Sec. 3.2, we provide an overview of a common real-world architecture of a route server deployed inside IXPs. In Sec. 3.3, 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. 3.4, 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. 3.5, we evaluate our system by using a real-world trace of BGP updates from one of the largest IXP worldwide. In Sec. 6.2, we review the most relevant contributions related to Internet routing privacy and security. Finally, we draw conclusions and future work in Sec. 5.9.

3.2 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+14]). 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 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 an RS-software is shown in Fig. 3.1. 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 an 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, an 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

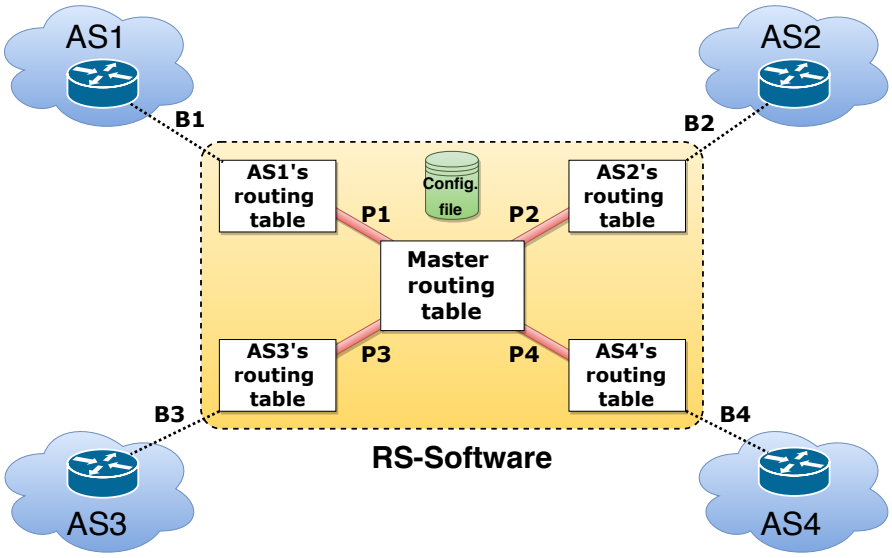


Figure 3.1: Reference architecture of route server.

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. 3.1). 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. 3.1). A pipe is a bidirectional communication channel connecting two tables, such that every route added to one table is immediately inserted into the other (or "peer") table, and vice versa. This applies to all routes, regardless of whether they are marked as best or not. Although in BIRD pipes can be used to connect an arbitrary pair of tables, the best practice is usually to just link each client-specific table with the master table [RSF+14].

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. BIRD supports the definition of both import and export filters: for the case of the `bgp` protocol, import filters are applied to BGP announcements received by the RS from the clients, while for the case of the `pipe` protocol import filters are applied to BGP routes when they are copied from a client-specific table to the master table. Export filters are applied in the opposite direction in both cases. Of course, each IXP member may have custom inbound and outbound BGP policies set up in its own routing software. 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 $(0 : \mathsf{IXP\_id})$, 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 $(\mathsf{IXP\_id} : \mathsf{IXP\_id})$ 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 ,combined with the fact that pipes are established between each client-specific table and the master table, allows IXP operators to overcome the well-know problem known as *path hiding*, which arises whenever filters are applied [JHRB16a, RSF+14]. 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].

3.3 Enforcing Privacy of Routing Policies

In this section, we describe how PrIXP improves the level of privacy for the members' routing policies within an 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. 3.2) 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 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. 3.2. 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

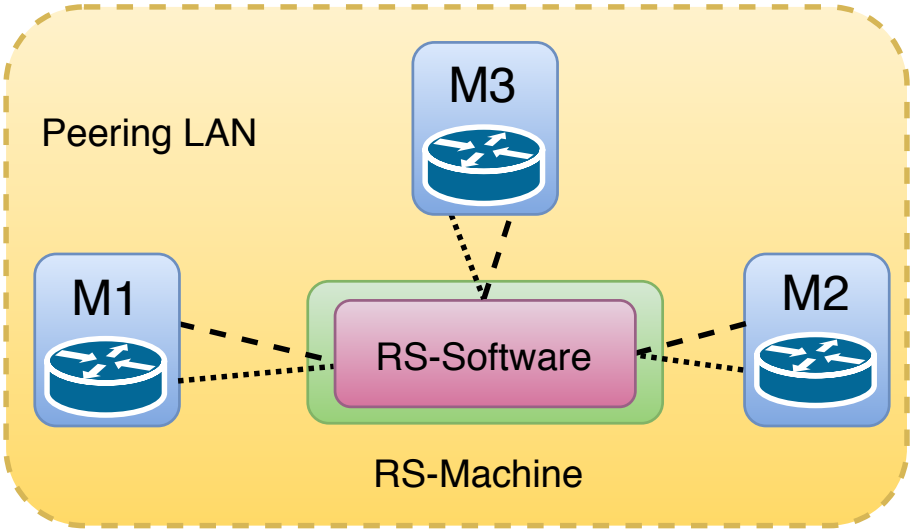


Figure 3.2: The architecture of an IXP infrastructure.

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.

3.4 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. 3.3 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 an RS security architecture, depicted in Fig. 3.3, 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 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

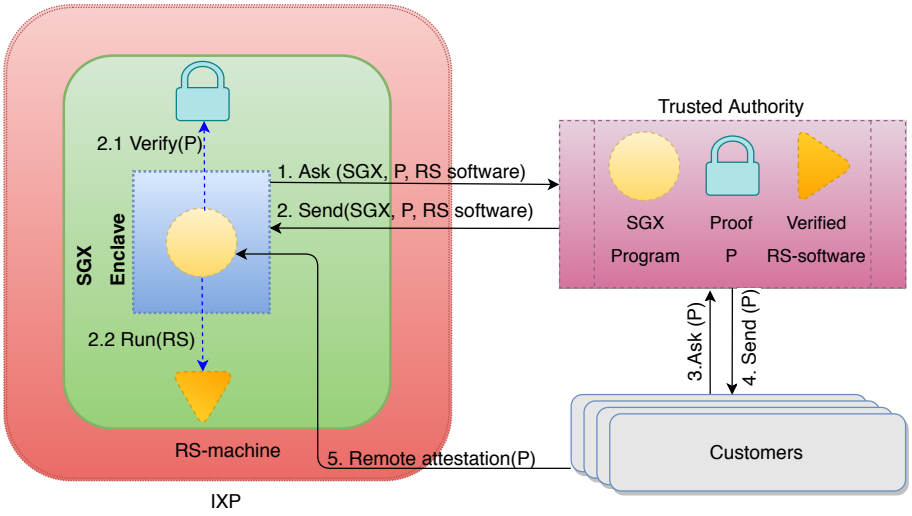


Figure 3.3: Architecture for checking the integrity of the RS software.

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. 3.3.

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 chapter, 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.

3.5 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 chapter. 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. 3.4, including a decision process acting according to the route dispatching mechanism described in Sec. 3.3. 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: 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

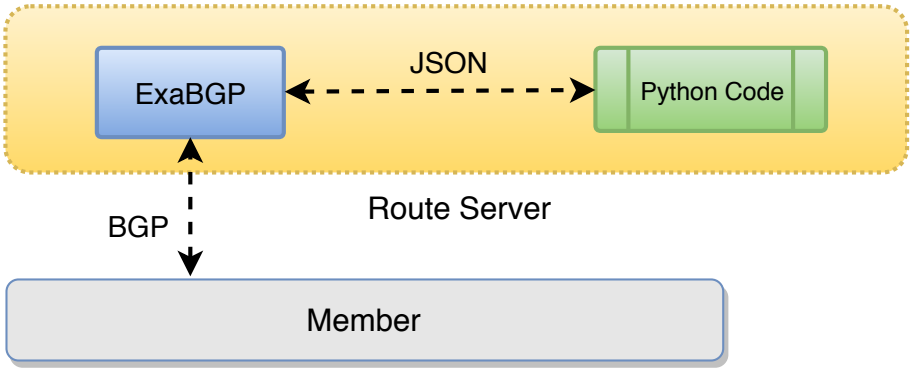


Figure 3.4: Architecture of our RS-software prototype implementation.

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. 3.5. 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 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. 3.6. 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* ([JDS+16, Ope16]), an open source implementation of SGX. This experiment aims at verifying that the remote attestation mechanism described in Sec. 3.4 behaves as expected. We produced a hash value of the PrIXP implementation.

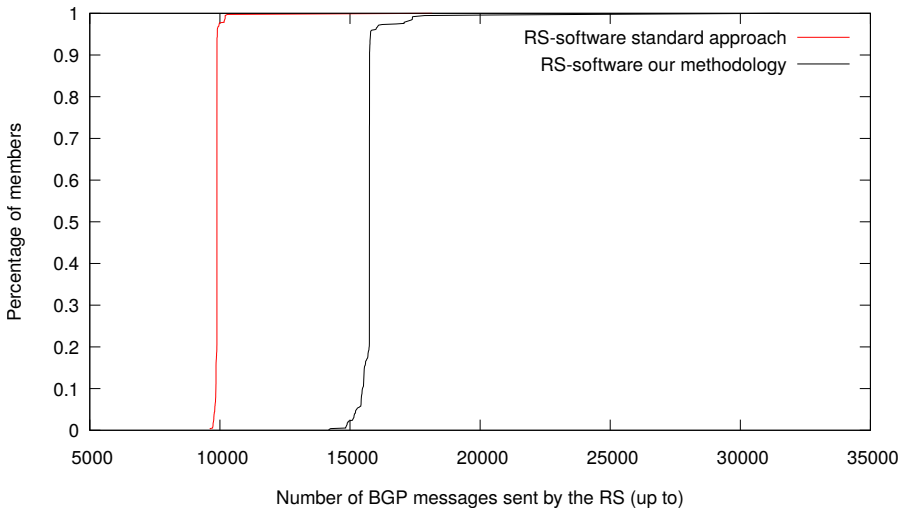


Figure 3.5: CDF of the number of messages issued by the RS-software.

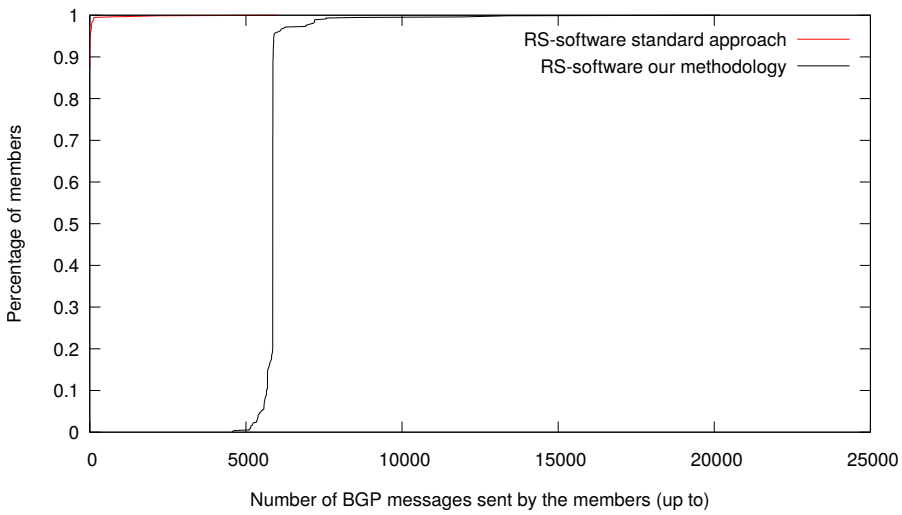


Figure 3.6: CDF of the number of messages issued by members.

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. 3.3. 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.

Performance overhead and threats of SGX. We are aware that leveraging SGX may add some overhead to our RS-software. This overhead comes when SGX executes the enclave code [ATG+16, TSB18]. The SGX performance overhead can be categorized as follows. 1) Since the privileged instructions cannot be executed within the enclave and it requires system calls. 2) Enclave pays to write in memory and cache misses because memory encryption engine (MEE) encrypts and decrypts cache lines. 3) Application that require extra protected physical memory, which is called the enclave page cache (EPC), should swamp pages between EPC and unprotected DRAM. This requires further encryption and decryption. Furthermore, SGX is vulnerable to several attacks like cache timing and page table side-channel attacks and several works tried to protect it like [OTK+18].

3.6 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 Multi-Party 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.

3.7 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.

We designed PrIXP, an 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.

Chapter 4

Experimenting SDN in Inter-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 chapter 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.

*Part of contexts in this chapter is based on the following publication: Mostafaei, H., Lospoto G., di Lallo R., Rimondini M., Di Battista G., SDNetkit: A Testbed for Experimenting SDN in Multi-Domain Networks, 2017 IEEE Conference on Network Softwarization (NetSoft), Bologna, 2017, pp. 1-6.

4.1 Introduction

Researchers and practitioners interested in SDN need systems to perform experiments with OpenFlow [MAB+08] and, over the years, Mininet [LHM10] 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 NS-3 [Jur13], Estinet [WCY13], distributed mininet [LO15], Maxinet [WDS+14], Mininet CE [AS13], 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 [ovs17] 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, i.e., directly connected, with each device in the network.

We argue that these two weaknesses pose severe limitations in experimenting interesting scenarios, such as *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 chapter, 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).

The rest of the chapter is organized as follows. In Sec. 6.2 we review the most relevant state of the art about network simulators and emulators for SDN and OpenFlow, pointing out which are the differences among them and SDNetkit. In Sec. 4.3, we present basic concepts on Netkit, highlighting how we improved it in order to create SDNetkit. We also briefly describe the SDNetkit architecture. In Sec. 5.6 we show two use cases aiming at presenting scenarios to test interoperability and network control, involving multiple providers simultaneously offering services in the network. In Sec. 4.5 we discuss issues we experienced in setting up specific configurations for network control. Finally, in Sec. 5.9 we report future improvements to SDNetkit we are interested in pursuing.

Platform	Type	OpenFlow Support
Mininet [LHM10]	Centralized	1.0, 1.3
Mininet CE [AS13]	Centralized	1.0, 1.3
Maxinet [WDS+14]	Distributed	1.0, 1.3
Distributed Mininet [LO15]	Distributed	1.0, 1.3
NS-3 [Jur13]	Centralized	1.0, 1.3 [1]
Estinet [WCY13]	Centralized	1.0, 1.3
MiniNExT [min17]	Centralized	1.0, 1.3
Ofelia [ofe17]	Remote testebed	1.0
ToMaTo [tom17]	Remote testbed	1.0

Table 4.1: A comparison among simulation systems in terms of type (e.g. centralized, distributed, or remote testbed) and supported OpenFlow versions.

4.2 Related Work

In this section, we review the most relevant state of the art with respect to SDN and OpenFlow simulators and emulators, pointing out the differences among them and SDNetkit.

Table 4.1 shows an overview of the available tools. We group them into three categories: (1) centralized platforms, namely systems that run on a local machine; (2) distributed platforms, namely systems that run on distributed environments (e.g. clusters); and (3) remote testbeds, namely physical networks that offer to the users the possibility of experimenting SDN by providing virtual overlay networks built on top of the physical infrastructure. In the rest of the section we discuss each category, giving an overview of the simulation systems.

Centralized Platforms - Mininet is the most used system allowing users to experiment with SDN, more specifically OpenFlow, capable networks. It uses a lightweight virtualization technique to create and manage OpenVSwitch instances and provides an extensible CLI and API for rapid prototyping. Unfortunately, it does not provide any support for experimenting with interoperability scenarios, namely networks in which OpenFlow and standard routing protocols (e.g. OSPF) simultaneously run.

MiniNExT is an extended version of Mininet which enables the possibility

[1] *The support to OpenFlow 1.3 is carried out by a thirdy party module* [ns317]

of experimenting networks also running traditional routing protocols. It introduces the support for standard routing suite (e.g. Quagga). Unfortunately, the system is not easily extensible and it is no longer supported.

Despite the name, Mininet CE (Cluster Edition) is a centralized system allowing the creation of multiple Mininet instances over a single machine. These tools have in charge the task of coordinating those instances in such a way that they collaborate. As for Mininet, it does not allow network devices to run traditional routing protocols.

NS-3 (Network Simulator) allows the users to run SDN networks with Direct Code Execution (DCE) module. The tool just supports OpenFlow 1.0 specifications. Additionally, it allows the network topologies to have interactions with the real-time network components. A performance comparison of NS-3 with other tools like Mininet can be found in [IYZR16].

EstiNet is a network simulator and emulator which supports OpenFlow based networks. It provides the capability of using well known controllers, like NOX/POX or Floodlight. Those controllers run on a host which is created by Estinet. As for Mininet, and all platform based on it, Estinet does not support the usage of standard routing protocols. In addition, it is not freely available.

Distributed Platforms - Distributed Mininet [LO15] has been introduced in order to support the Mininet execution over a distributed environment. The basic idea of that system is to create several cooperating Mininet instances and span them over a cluster of physical machines, for scalability purposes. Nevertheless, it is able to just run OpenFlow devices.

MaxiNet [WDS+14] is another extension of Mininet to span a large network on several physical machines instead of using a single one. This feature gives to MaxiNet the possibility of emulating networks with the size of a data center with several thousands of the nodes and servers. The general idea behind Maxinet is to create several workers which run the original Mininet. Then, by exploiting a centralized API, the tool can have access to the clusters of Mininet instances. A set of suitable APIs are provided for Maxinet to interconnect and control the Mininet instances. This tool also inherits the limitations of Mininet to experiment mixed scenarios.

Remote testbeds - Systems falling in this category are meant as services offering overlay networks relying on a physical infrastructure. Examples are OFELIA and ToMaTo. They offer the possibility of creating custom networks in which OpenFlow devices run. The level of OpenFlow support of this platform is poor (just OpenFlow 1.0 is supported) and they still have problems with respect to interoperability.

With respect to every system discussed so far, SDNetkit aims at overcoming those limitations, providing an emulator for experimenting interoperability and network control scenarios, which are not feasible by using Mininet and systems based on it. Also, emulating networks with SDNetkit is not more laborious than experimenting with standard networks (e.g. in terms of effort required to configure devices and set up the network itself), in contrast to what happens with other tools (e.g. NS-3). In addition, SDNetkit is scalable and extensible, in terms of software (e.g. standard tools or SDN software) that can be installed inside that system. It also provides a simple interface to the final users allowing them to easily create and run custom topologies.

4.3 From Netkit to SDNetkit

In this section, we give a brief overview about Netkit, showing its architecture and presenting several basic concepts. Finally, we describe how Netkit has been enriched in order to provide SDN functionalities, resulting in SDNetkit.

Netkit [net17] is a network emulator allowing users to run complex network topologies and to experiment the behavior of standard routing protocols (e.g. OSPF). Netkit has two strengths: first, it does not require administrative privileges to run. Second, it does not require many resources to be executed.

The emulation approach used by Netkit is very simple: every network device is a virtual machine. Resources owned by each virtual machine are mapped to portions of the corresponding resources on the host which Netkit is running on. Each virtual machine is equipped with a disk, a memory, and one or more virtual network interfaces. The virtual machine disk is a file in the host machine, containing all software that can be executed inside the virtual machine itself (e.g. routing protocols daemons). The memory is shared with the host and it can be set independently for each virtual machine. Finally, the network interfaces are connected among them exploiting virtual hubs. By using those virtual hubs, each virtual machine is also able to access the Internet. The network functionalities of each virtual machine depend on the software that is running. That software can be run starting from the virtual file system that is owned by each virtual machine. For instance, a device can act as a standard switch whether standard ethernet bridge administration software is executed (e.g. by using the `brctl` suite) or as a router by properly populating the IP routing tables (e.g. by running routing protocols).

Netkit and SDNetkit share exactly the same architecture. Indeed, SDNetkit has been built on top of Netkit by adding SDN software to the virtual file system

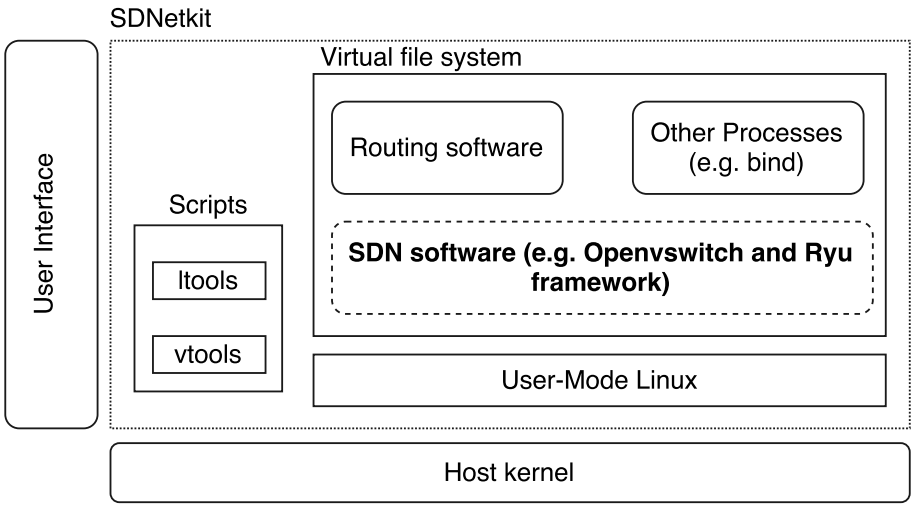


Figure 4.1: Overview of the SDNetkit architecture.

of each virtual machine. In the rest of the section we focus on the description of the SDNetkit architecture. The SDNetkit architecture is depicted in Fig. 4.1. SDNetkit, represented by the dotted square in the figure, runs on top of the host kernel by exploiting User-Mode Linux (UML) [uml17], a software allowing a kernel to run as an userspace process. Each SDNetkit virtual machine has its own virtual file system containing all software that can be executed in the virtual machine itself. Examples of software are routing protocols suite (e.g. Quagga) and software for specific purposes (e.g. BIND for DNS functionalities). The interaction between users and SDNetkit is made by a collection of scripts (`vtools` and `ltools` in the picture) that allow a user to run a single virtual machine or more cooperating virtual machines, respectively. In the SDNetkit terminology, a *lab* is a network composed of many virtual machines connected among them. A lab is described by a folder containing a configuration file in which the user specifies how virtual machines are connected to each other. In that file it is possible to tune specific parameters for each virtual machine (e.g. the amount of memory). Also, for each virtual machine, there is a *startup* file containing specific configurations (e.g. network interface configurations) and a folder containing configurations used by software running inside the virtual machine itself (e.g. Quagga configuration files).

In order to introduce SDN functionalities in Netkit, we equipped it with *ad-hoc* software. More specifically, we added the support to the OpenFlow protocol, the most used protocol enabling SDN capabilities, by including in the virtual file system OpenVSwitch as the OpenFlow-enabled switch and the Ryu framework as SDN controller. SDNetkit now supports up to OpenFlow version 1.3 [Ope14]. We point out that other SDN software can be added to Netkit without any restrictions, by simply adding it to the virtual file system. It can be selectively done on each virtual machine or by building a new file system that is used as starting file system during the boot phase of each virtual machine. For example, we can simply upgrade the OpenFlow version from 1.3 to one of the latest versions like 1.5. More detailed information can be found here [net17]. Such an architecture guarantees that SDN software can run together with standard one on each virtual machine. This opens the possibility to test scenarios in which standard devices running standard protocols (e.g. OSPF) can interact with OpenFlow switches running OpenVSwitch instances. Also, by using SDNetkit it is possible to simultaneously run those software on the same device. To the best of our knowledge, SDNetkit is the first emulator enabling such an interoperability mode.

4.4 A Simple Example and Success Stories

In this section, we present two use cases, in order to show how mixed scenarios can be realized and tested. The first use case shows how OpenFlow and OSPF simultaneously run on the same device. We call such a use case *Hybrid node*. The second aims at presenting a network in which OpenFlow nodes interact with standard ones. We name such a use case *Hybrid topology*.

Hybrid Node Use Case

The hybrid node use case is implemented by using a simple topology in which at least one device simultaneously runs both OpenFlow and a standard routing protocol. We chose to run OSPF as routing protocol on the hybrid node. We point out that other routing protocols (e.g. RIP) can be executed without any restrictions. The reference topology is depicted in Fig. 4.2.

Controller is an SDNetkit virtual machine running an instance of the Ryu framework. The goal of such a machine is to correctly manage the hybrid node. Routers R1 and R2 are IP-speaking nodes running OSPF. The role of those devices is to show that it is possible to handle OpenFlow devices by using a

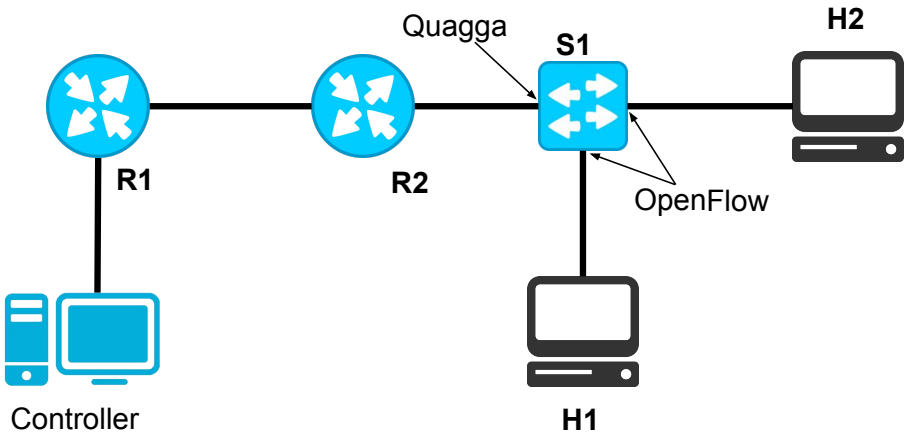


Figure 4.2: Topology for the Hybrid node. S1 is the device in which OpenFlow and OSPF simultaneously run.

dedicated management network, in contrast to what happens in Mininet, where this scenario cannot be carried out. H1 and H2 are virtual machines acting as standard hosts. S1 is the hybrid node and arrows in the figure summarize which protocol runs on which network interface for that node. On that virtual machine, both OpenVSwitch and Quagga, more specifically OSPF, are running. The S1's virtual network interfaces connecting it to the two hosts are OpenFlow ports, namely they have been assigned to the OpenVSwitch software process by using the `ovs-vsctl` suite, whereas the port connecting S1 to R2 has been assigned to the Quagga routing protocol suite by writing specific statements in the OSPF configuration file.

We set up an SDNetkit lab implementing this topology and run it in order to check whether Controller and S1 properly communicate. To do that, we wrote a very simple piece of software on top the Ryu framework showing information about the hybrid node once it successfully performed the handshake with the controller. We observed that both Controller and S1 can communicate. After that, we added to our Ryu-based software a piece of software aiming at instructing S1 to send to Controller machine all ARP traffic. We verified that the OpenFlow message carrying such a rule, i.e., the rule to send all ARP traffic to Controller, reaches S1 and it is correctly installed in its flow table. Once this check passed, we perform a ping from H1 to H2 and we verified that all ARP requests issued by H1 matched the rule previously installed on S1 and they

finally reached the Controller.

With this very simple use case, we showed that by using SDNetkit it is possible to create dedicated management networks and those networks can be arbitrarily complex.

Hybrid Topology Use Case

The hybrid topology use case is implemented starting from the reference scenario we used in [MLB+17a] and it is depicted in Fig. 4.3. In that topology, there are three different domains, ISP1, ISP2, and ISP3, each running its routing protocols. In this case, there are no hybrid nodes: each device runs either OpenFlow protocol or legacy routing protocols, like OSPF. The goal of this use case is to show how pure OpenFlow devices interoperate with IP-speaking ones. To accomplish this task, we performed a test in which H1 issued a DNS request in order to resolve the H2's domain name. With this example, we are able to validate the interoperability among OpenFlow-enabled and legacy devices, verifying whether the name resolution is successfully carried out.

Such a scenario is more complex with respect to the hybrid node one. Routers from R1 to R9 are IP-speaking devices running OSPF. Since such a scenario is multi-domain, routers R4, R5, and R6 also run an instance of the Border Gateway Protocol (BGP) in order to establish BGP peering among them. Switches S1 and S2 are pure OpenFlow-enabled devices, namely, on those virtual machines an instance of OpenVSwitch is running. Each of them is handled by a controller that is directly connected to the switch. We remark that even if this is not a reasonable assumption (in general such a back-to-back connection might not be set up in production networks), we showed in the previous use case that they can communicate by using an arbitrarily complex management network.

We placed a set of name servers in the network, realizing a DNS hierarchy. In particular, there is a *root* name server in the domain ISP3, as well as an *authoritative* name server for the domain which ISP1 and ISP2 belong to. We assume that this name server is authority for `.it`. Other name servers are ISP1-NS and ISP2-NS that are authority for ISP1 and ISP2, respectively. Finally, there are two customers, Customer1 and Customer2, that are connected to ISP1 and ISP2 respectively. Each of them has its own public IP subnet and comes with a local name server (NS1 for Customer1 and NS2 for Customer2). All traffic issued by those customers pass through the OpenFlow-enabled switches S1 and S2. By doing so, we are able to carry out our experiments. We recall that in this use case, we focus on DNS traffic.

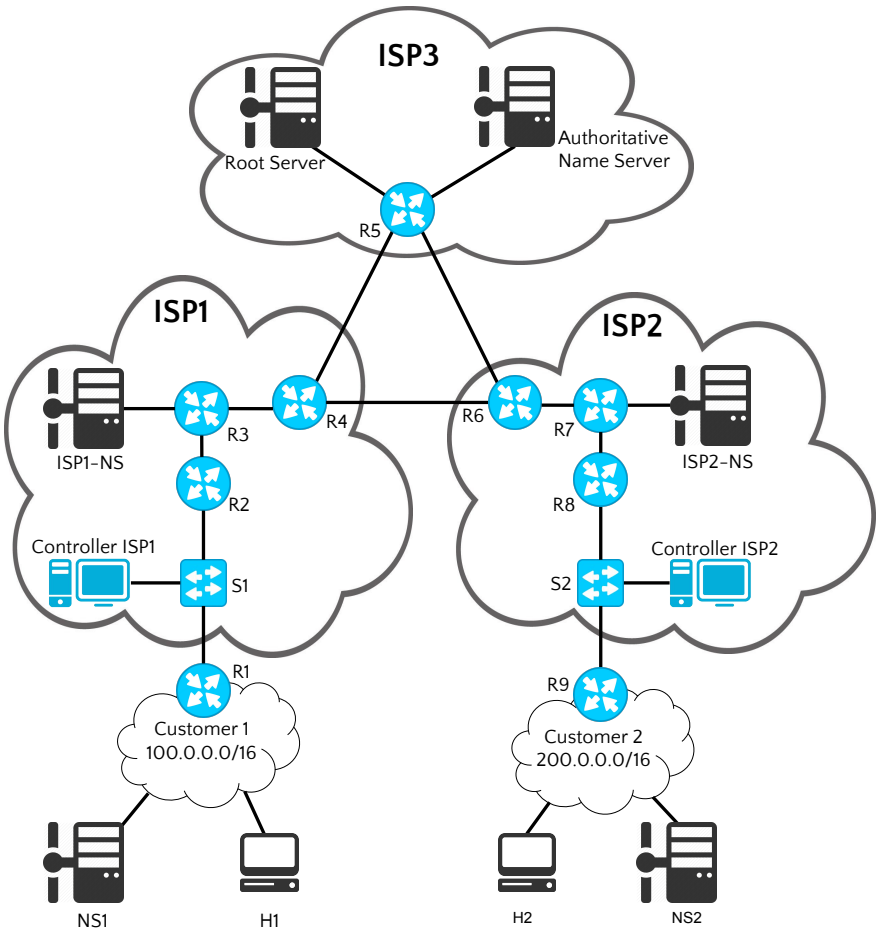


Figure 4.3: Topology for the Hybrid topology. S1 and S2 are pure OpenFlow-enabled switches, whereas all other devices are IP-speaking nodes.

First of all, we wrote an SDN controller that is able to properly handle: (1) ARP traffic; (2) ICMP traffic; and (3) DNS traffic. We need to handle ARP traffic since the interfaces of S1, as well as those of S2, are on two different collision domains, namely each of them has an IP address belonging to different subnets. Hence, we need to intercept all ARP traffic coming from R1 and produce suitable ARP request towards R2. Basically, our SDN controller instructs S1 to act as a standard router. Obviously, the same happens for S2. To accomplish such an operational mode, we install on S1's flow table a rule to send all ARP traffic to the SDN controller. Once that traffic reaches the controller, it properly reacts by producing suitable ARP packets and it sends them to the OpenFlow-enabled device using specific OpenFlow messages (e.g. PacketOut). We successfully verified that ARP traffic has been handled according to what we expected.

In order to handle DNS packets, our controller simply installs a pair of rules to allow traffic coming from and directed to Customer1 to be forwarded to the next router on the path (R2 and R1, respectively). To issue DNS packets, we executed standard commands for DNS lookup, like `dig` (with the option `+trace` in order to verify the interactions between all name servers in the network) and `host`. We checked that the rules have been correctly installed on the OpenFlow-enabled devices and they have been matched by DNS packets issued by H1.

Finally, we performed an experiment in order to verify whether ICMP traffic is correctly managed. Our controller installs a pair of rules in the S1's flow table in order to forward traffic generated from and directed to Customer1. Our controller does the same on S2 to forward Customer2 traffic. After that, we executed a ping from H1 towards H2 (and vice-versa) and we verified that ICMP traffic has been correctly forwarded.

With this use case, we showed that SDNetkit is able to run topologies in which pure OpenFlow devices cooperate with standard ones. SDNetkit also allows users to create topologies that are arbitrarily complex: the topology we used in this chapter is just an example aiming at showing that having such an interoperability feature is feasible. On the other hand, we used a very simple controller with respect to that used in [MLB+17a]. SDNetkit does not introduce any limitations in terms of what a controller can do: it is possible to implement SDN controllers that are complex at will.

4.5 Configuration Considerations

In this section we report a collection of experienced issues in producing specific configurations, aiming at providing a more complete view of what SDNetkit allows users to do. Surely, having the possibility to use a dedicate management network, as shown in Sec. 5.6, is a plus to experiment with SDN testbeds. The best solution might be to use the same network interfaces of a virtual device to forward both control (e.g. OpenFlow messages) and standard traffic (e.g. packets exchanged between source and destination). We tried to do that, unfortunately experiencing several problems that we summarize in the following. We argue that the issues which we dealt with cannot be directly ascribed to SDNetkit.

We refer to the topology depicted in Fig. 4.4 in order to achieve the goal of using the same network interfaces of a device to forward both traffic generated by end hosts and traffic generated by protocols running on the devices (e.g. OpenFlow, OSPF, BGP, ...). Arrows in the figure show a summary of how network interfaces are assigned to Quagga and/or OpenVSwitch processes for this scenario. We now explain the details of such choices.

In that network, R1, R2, and R3 are hybrid nodes. H1 and H2 are hosts and Controller is the SDN controller. The network interface connecting R1 with Controller has been assigned to Quagga process. OSPF is running on that interface. On the other hand, the network interface of R2 connecting it to H1, as well as the network interface connecting R3 and H2, have been assigned to the OpenVSwitch process. Since we are exploiting the same network for different types of traffic, each router runs both OSPF and OpenFlow. Differently from the use cases reported in Sec. 5.6, in this use case some network interfaces of a device are simultaneously assigned to different processes. The network interfaces of each hybrid node connecting it with each other have been assigned to both Quagga and OpenVSwitch processes, since OSPF is used to compute a routing allowing each hybrid node to reach the controller and OpenFlow is used to install rules on the network devices in order to forward standard traffic. Those configurations are possible and have been accomplished without any problems.

Then, we have to configure routers so that they can interact with the controller. The first step consists in allowing the OSPF packets to reach each router in order to compute a routing that is used to forward OpenFlow messages. To allow that traffic to traverse OpenFlow-enabled devices, a set of rules must be pre-installed on them. Those rules are statically installed, since they are used to allow each OpenFlow-enabled device to reach the SDN controller.

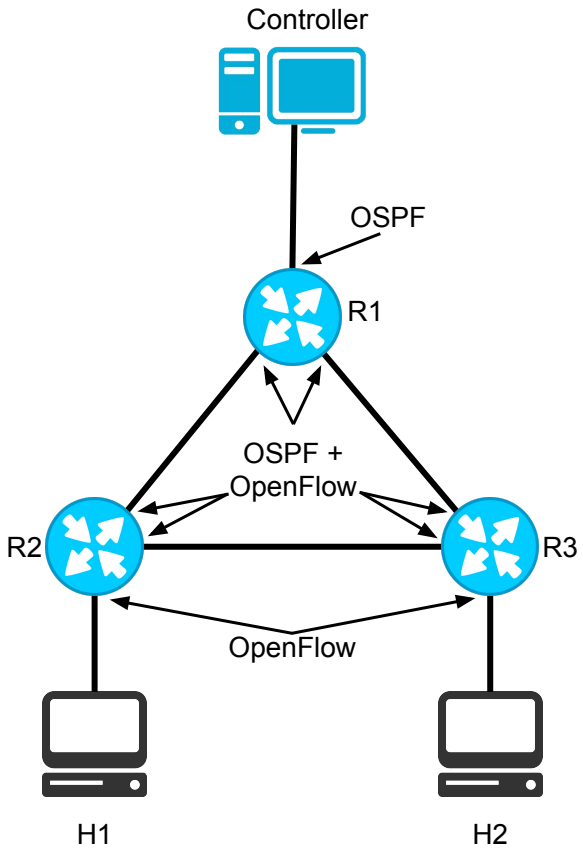


Figure 4.4: Topology used to experiment a single network for forwarding both control and standard traffic.

By doing so, it is not convenient to determine which is the network interface to use to send out such a traffic, since in case of a failure there are no routing protocols being able to compute a new routing and to automatically change the value of the out port in the OpenFlow rule. OpenFlow provides a suitable action to accomplish this task: the `NORMAL` action, that we used for this purpose. The semantics of this action is to handle all traffic matched by the rule using the standard protocol stack. However, since OpenVSwitch is a layer 2 switch with OpenFlow capabilities, what we observed is that the traffic is sent broadcast on every network interface, that is the usual operational mode of a switch. This is an undesirable behavior for our goal. Indeed, we expected that the traffic was processed by Quagga in order to allow OpenFlow messages to reach the SDN controller. Such a behavior results in the impossibility for OpenFlow devices that are not directly connected to the controller to establish a connection with the controller itself.

We also faced with another problem, that is a direct effect of the previous one. The control channel of an OpenFlow-enabled is managed by the controller and if an OpenFlow-enabled device is not able to reach the controller, we cannot install any rule on the device. However, the OpenFlow-enabled device can handle the flows based on the installed rules. We became aware of such an issue by performing the following experiment. We manually installed on `R2` and `R3` rules (by using the `ovs-ofctl` command) to handle traffic generated by the hosts. More specifically, those rules matched ARP and ICMP traffic, forwarding those packets on suitable output ports. We observed that those rules have never been matched.

We believe that OpenVSwitch is not suitable to realize such a use case. We also argue that replacing it with an implementation that also supports layer 3 capabilities (e.g. OSHI [SVL+16]) might be enough to set up such a scenario.

Consider that the limitations we put in evidence can be easily overcame by configuring a pair of VLANs on each of the connection between routers and by using one VLAN for the control traffic and the other VLAN for standard traffic.

4.6 Limitations of Network Emulators

Since SDNetkit is a network emulator it comes with some performance limitations. SDNetkit runs on a single machine and like other tools it comes with some limitations. The limitations of network emulator can be described as follows [Ram13, AOC+10]. 1) The reliability of the results is difficult to validate

because it depends on several factors. The results of network emulators can be useful if they are comparable with the results of a real network. 2) The scalability of an emulator relies on at least two factors; the computation time and the memory usage. Each component of an emulated network requires memory space which is bounded with the available memory of the system. The execution time of each event depends on the processing power of the system and can vary.

4.7 Conclusions and Future Work

We present SDNetkit, a network emulator with SDN capabilities built on top of Netkit. We show several use cases discussing scenarios also involving multi-domain networks, like interoperability and network control, that cannot be realized with current systems for experimenting with SDN technologies. We also discuss specific configuration settings, pointing out issues that prevent us to test other scenarios. Also, by using SDNetkit is possible to run any application on every virtual machine, opening to the possibility of running scenarios that are application software oriented.

As future research direction, we plan to create more use cases, aiming at carrying out interesting interoperability scenarios for multi-domain networks. Moreover, we intend to further enhance SDNekit, by adding new SDN software that can be used to overcome the limitations we dealt with, since it also guarantees the interaction with the application level, as shown in Sec. 5.6. We are also planning to release a container-based version of SDNetkit.

Chapter 5

Multi-Provider VPNs in Software-Defined Federated Networks*

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 adopted best practices 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

*Part of contexts in this chapter is based on the following publication: Mostafaei, H., Lospoto G., di Lallo R., Rimondini M., Di Battista G., Sdns: Exploiting sdn and the dns to exchange traffic in a federated network. In Network Softwarization (NetSoft), 2017 IEEE Conference on, pages 1-5. IEEE, 2017. and another part is on the following publiction: Mostafaei, H., Lospoto G., di Lallo R., Rimondini M., Di Battista G., A Framework for Multi-Provider Virtual Private Networks in Software-Defined Federated Networks, under review.

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.

5.1 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 and 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 *quickly deliver* it: "5 days are needed". Thus, our question is: Is there a way to reduce such a provisioning time?

In this chapter 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' networks, 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+17a].

In [dLRB16], we propose several mechanisms to support SDN-based end-to-end connectivity in federated networks by means of source and destination IP addresses translation. In that chapter, we also introduce the operations that the SDN-controllers have to carry out in order to keep the customer information updated. In [MLB+17a], we illustrate how the SDN-controllers interact with the DNS and show how this interaction enables the SDN-controller to get information about the IP addresses that are replaced by applying the mechanisms described in [dLRB16].

The main contributions of this chapter can be summarized in: 1) drastically reducing the time needed to provision the federated VPNs and 2) relying on the SDN architecture to set up the federated network and to provide the federated services.

We reach those goals by providing a configuration language that allows each federated ISP to easily define federated networks as well as quickly configure and provision federated VPNs. We provide a set of primitives that allow customers to join or leave from federated VPNs *on-demand*, thus reducing both explicit configuration actions of the ISPs and the time required to set up or down the federated VPNs. Also, in order to allow ISPs to keep unchanged their whole network architecture, we introduce in the network the smallest number of SDN-enabled devices. Finally, we exploit the SDN architecture to interact with the application level (e.g. the DNS system), allowing each ISP to keep unchanged any IP address plan previously assigned to its customers. The interaction with the DNS is crucial. Indeed, inside a federated network, there are no guarantees about the fact that the IP addresses of each customer participating to a VPN do not overlap. Since this requirement is mandatory, such an interaction allows providers to not change the IP address plan of the 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 add-value of our framework is the following: *customers sharing the same IP address plan can be part of the same federated VPNs, without any limitation.* In the evaluation, we focus on two coordinates: control- and data-plane impact. Our experiments show that the number of control plane messages linearly grow with respect to the number of DNS queries in the network, regardless of the amount of traffic injected in the network. On the other hand, the impact on the data-plane, measured in terms of SDN rules installed at the SDN-enabled devices linearly depend on the number of customers per ISPs that join the federation.

The rest of the chapter is organized as follows. In Sec. 6.2 we review the state of the art. In Sec. 5.3 we discuss today's best practices for federated networks. In Sec. 5.4 we show our framework and how it can be used to tackle the most common problems in today's federated networks. In Sec. 5.5 we present our configuration language and our primitives, explaining how they allow a fast delivery of the service. In Sec. 5.6 we show a complete example, detailing the interaction between SDN and the DNS. In Sec. 5.7 we summarize the benefits of our framework. In Sec. 6.9 we discuss the results of our experiments. Finally, in Sec. 5.9 we draw conclusions and future research perspectives.

5.2 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, the 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 networks. 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 adopt LISP.

All the proposals that rely on standard technologies either require a non-negligible amount of time to make the federated services available to the users or impose assumptions that are not applicable in real networks. In both cases, the provisioning is highly impacted. In contrast, our framework aims at simplifying the provisioning of the federated services, demanding to the SDN-controllers the management of all the aspects related to the federation. Thus, both the configuration effort and the amount of time to set up the federated services are reduced.

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+17a] we built on top of that paper a mechanism based on the DNS to simplify the communication among end-hosts. In this chapter, we extend [dLRB16] and [MLB+17a] 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, addressing the today's federated network issues.

With respect to the application of SDN to the interdomain routing, Software Defined eXchange point (SDX) [GVS+15] represents the first and the most significant proposal in that direction. SDX introduces SDN in the layer 2 switching fabric allowing the interconnection among the members of the Internet eXchange Point (IXP). To overcome some specific limitations, an industrial version of SDX has been also proposed [GMB+16]. To further improve the sca-

lability, Endeavour [ACC+17] proposes a specific architecture for the layer 2 fabric switch network of the IXPs. In Software Defined Inter-domain routing (SDI) [WBL+16], the authors propose a new SDN-based protocol in order to improve several aspects (e.g. multipath interdomain routing). Nevertheless, our framework does not impose specific IXPs' architecture, since we consider IXPs as interconnection points, regardless specific technologies.

5.3 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 to 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 ISPs to reduce the amount of time spent in coordination activities as well as technological and management issues. Also, several costs can be reduced. Indeed, being present in a PoP represents a cost, often recurrent, for the provider. Examples of those costs are equipment (e.g. switches), network infrastructures (e.g. fibers, possibly considering multiple physical circuits for the interconnection), and housing (e.g. rent of space in the rack for the network devices). On the other hand, a provider has to consider costs for the SDN-enabled devices, if its devices are not SDN-enabled.

5.4 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. 5.3 (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

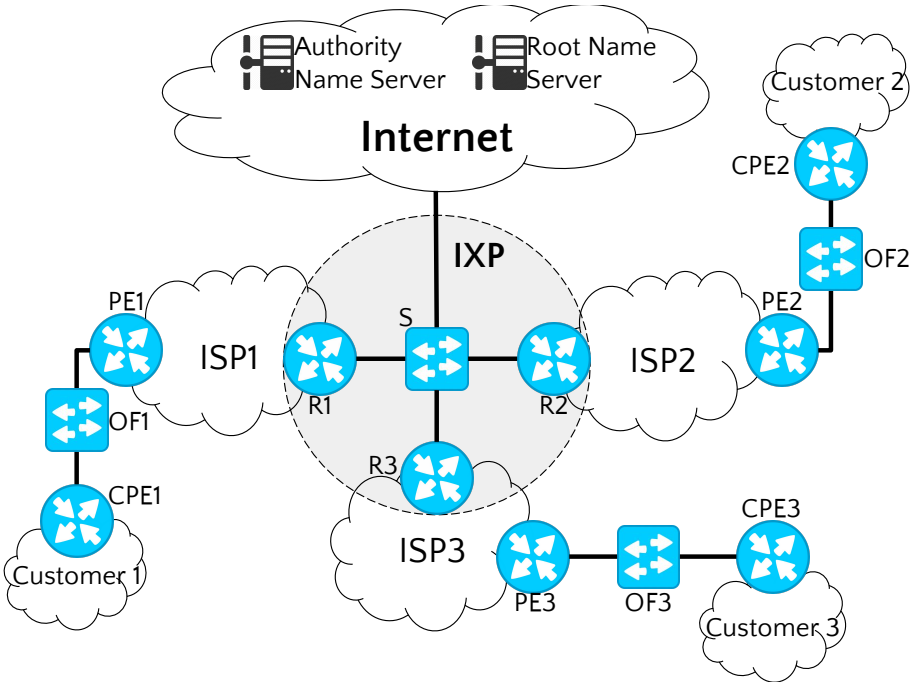
value-added services, like VPNs spanned over two or more ISPs' networks. Nevertheless, federated networks based on traditional technologies lead to many challenges [FBP+10].

We choose to rely on the SDN architecture since it brings flexibility in issuing services and it 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. 5.3, 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 ISPs in terms of expenses and configurations (e.g. buying and managing devices used in the federated PoP). We argue that having a BGP session with an ISP or being connected to an IXP is enough to create a federation and this requirement is easily satisfied by ISPs. Also, each ISP can be connected to several IXPs simultaneously as well as it can have multiple bilateral peerings with no restrictions.

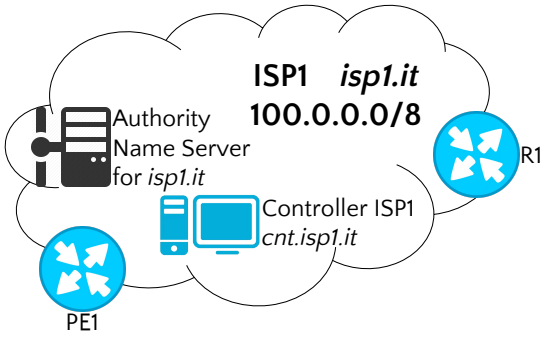
Our framework relies on this observation. By doing so, we preserve all the benefits of a federated PoP (e.g. clear identification of responsibilities) and save addictive costs due to new hardware. Removing the idea of having a federated PoP, combined with the centralized approach offered by SDN, allows 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. 5.1. 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. 5.1(a) has exactly three partners whose networks are called ISP1, ISP2, and ISP3, respectively. Such networks run IP-based routing protocols 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.

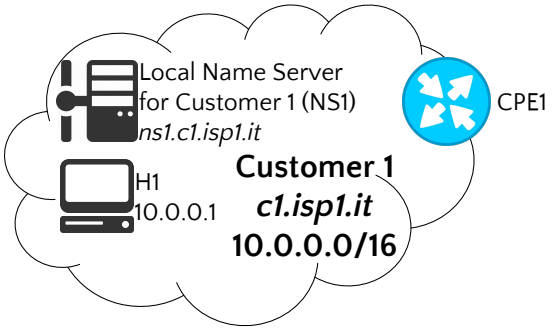
More generally, a federated network can have several partners. We assume that each of them has a border router peering with the border routers of the other partners. As we already said in this section, Internet eXchange Points (IXPs) are a natural place for establishing such peerings, therefore IXPs 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



(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 5.1: Reference scenario for SDNS.

itself, each ISP can also reach Internet. We assume that somewhere in Internet there are (at least) two name servers: 1) a root 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. 5.1(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. 5.1(b) and 5.1(c) depict the internal architecture of an ISP and of a customer, respectively. Referring to Fig. 5.1(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 chapter 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.

With respect to the internal architecture of a customer (Fig. 5.1(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 those servers behind the SDN-enabled switch allows us to take the control over the DNS traffic generated by the customers.

Referring to Fig. 5.1, 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 reported in [FBP+10] and related to management, 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. 5.1), 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 there 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).

5.5 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

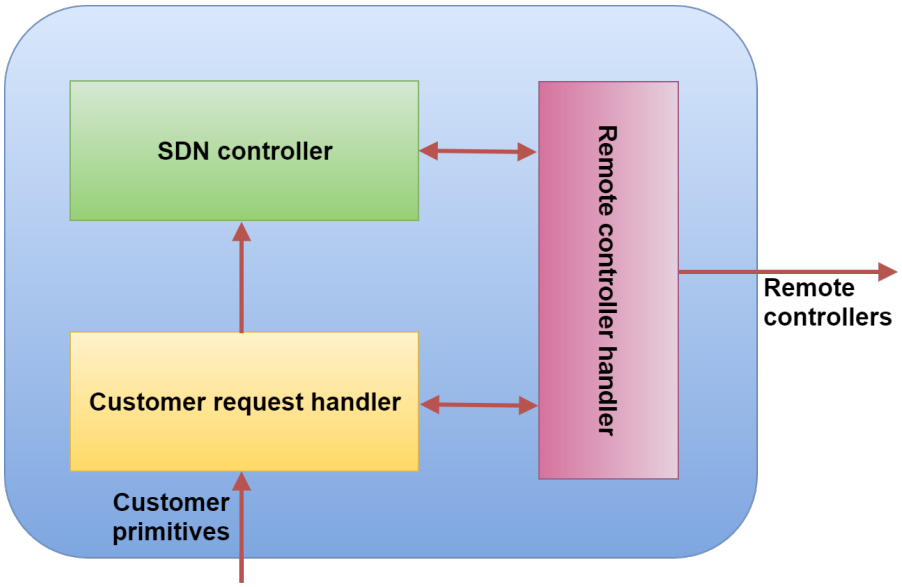


Figure 5.2: Logical architecture of our SDN-controller, consisting of a set of components each devoted to a specific task.

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 requirements, we internally design our SDN-controller as depicted in Fig. 5.2. Incoming and outgoing arrows mean that those components can be publicly accessed by customers and remote controllers, respectively (e.g. they have a public IP address). Note that the SDN-controller is not publicly accessible, avoiding to expose it to possible security issue. We divide our SDN-controller into three logical components: 1) the Customer Request Handler, which handles the requests coming from the customers to join or leave from a federated VPN; 2) the Remote Controller Handler, which takes care of keeping

the information about the federation and the federated VPNs updated; and 3) the SDN controller, which manages the SDN switches.

Security Considerations – With respect to the communication between SDN-controllers and SDN-switches, we rely on the guidelines suggested by the OpenFlow standard [Ope14]. We also extend the adoption of Secure Socket Layer (SSL) or Transport Layer Security (TLS) technologies to the communication between customers and providers (e.g. exchange of primitives), as well as to the communication among SDN-controllers. Since those phases need to be publicly accessed, we made several design decisions, in such a way to not expose the SDN-controller on the Internet, preventing it to be a possible target.

The components which really need to be accessed by the Internet are the Customer Request Handler and the Remote Controller Handler. Indeed, the first one must be reached by the customers that want to access the federated VPN service, whereas the latter one is used for the communication between the SDN-controllers. By doing so, the SDN-controller is not publicly exposed, hence the functionality of the SDN switches is not compromised by possible attacks. To further reduce the risk of cyber attacks, those components might run on different machines, exchanging information by means of private networks (e.g. standard point-to-point VPNs). Regardless of the specific setting, suitable interoperability mechanisms are used (e.g. JSON files or databases) to allow the communication among the components, so that the SDN-switches can be instructed with proper OpenFlow rules.

We highlight that the proposed architecture has the goal of preventing malicious users to take the control of the SDN-controllers, as well as, avoiding denial of service attacks against the SDN-controllers themselves. We argue that the components that can be publicly accessible (from the Internet or by a possible malicious customer) are the Remote Controller Handler and the Customer Request Handler, namely those components that do not act the SND-enabled switches, preventing a direct interaction with the SDN controller. We state that such an architecture is crucial. Indeed, if an SDN-controller was hacked, an attacker could install in the SDN switches arbitrary forwarding rules, so taking the control of the traffic issued by the customers. On the other, if one or more SDN-controllers was made unavailable, the SDN-switches would not be able to properly handle the requests coming from the customers, making the federated service unusable. In this chapter we do not address problems related to robustness in case of controller failures.

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.

We argue that our configuration language and our primitives, together with the SDN architecture, represent our solution for solving problems discussed in Sec. 5.4.

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. We explain the structure of our configuration language relying on the following example (see code in listing 5.1).

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. Each federated network has a globally unique name.

The <federation> element represents a federated network. In this example, ISP1 and ISP2 belong to a federated network called $\mathsf{ISP1-ISP2-net}$. The <myself> element contains all information about the SDN-controller of the ISP's network in which this configuration is deployed. The most relevant information are the URL and the IP address if the SDN-controller (e.g. cnt.isp1.it and 100.100.100.1, respectively). Additionally, a fake_ip_subnet is used by SDN-controllers in scenarios where hosts sharing the same IP address need to exchange traffic, as reported in Fig. 5.1. 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

```
<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>
    <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="10.0.0.0/16"/>
              <ns domain="n2.c2.isp2.it" ip_address="
    ↪ 10.0.0.3"/>
            </site>
            </customer>
        </isp>
      </vpn>
    </vpns>
  </federation>
</federations>
```

Listing 5.1: A configuration example for the language.

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. The <isps> element contains the list of all the other SDN-controllers belonging to the federated network. Even in this case, the relevan information are represented by the URL and the IP address of each controller. However, the element <nat> inside <isps> 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.

The <vpns> element contains both the list of all federated VPNs defined inside the federated network and the information about the customers belonging to each VPN. In this example, there is only a federated VPN, called $C1 - C2 - vpn$.

This configuration allows customers Customer1 and Customer2 to exchange traffic in the federated VPN. Customer1 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. Customer2 is a remote customer (we recall that this is the piece of configuration deployed at the ISP1's SDN-controller). In this case information about the SDN-enabled switch are not provided, since the ISP1's SDN-controller does not manage that switch. All the other information are taken as for Customer1.

We highlight that the information inside the configuration of each SDN-controller represents the minimum set of information to be specified to allow two (or more) customers to establish a federated VPN. Also, the configuration is simplified: it is now centralized and no more distributed over multiple devices, making the troubleshooting easier. As a consequence, the provisioning time of the federated services is reduced.

It is important to note the information enclosed inside <myself> and <isps> subtrees are static and manually configured by each ISP after reaching agreements with other ISPs on creating a federated network. While 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.

Finally, all the information provided in the configuration are commonly known by a provider. Indeed, providers know the port (physical of virtual) at the PE which each customer is connected to, as well as which are the IP addresses used to interconnect each CPE to the PE.

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 Customer Request Handler, that performs, cooperating with the other components, checks in order to allow (or deny) a customer to join the service. The customers do not need to know the presence of the SDN controllers in the ISP's network. Indeed, such primitives can be provided to customers by means of easy-to-use user interfaces, like a web portal, allowing the usage of the federated VPN intuitive and flexible.

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. There are two types of checks. The first one prevents a *malicious* customer to establish federated VPNs with other customers that are not interested in exchange traffic with it. The second type allows customers to receive traffic from and to send traffic to specific end systems. We deeply illustrate them.

Suppose that there are two customers, C1 and C2, and C1 is a malicious customer trying to set up a VPN with C2. Note that the VPN can be established if and only if both C1 and C2 issue the *Insert*. C1 sends an *Insert* to the Request Customer Handler of its provider. Once that component receives that *Insert*, it transfers suitable information to the SDN-controller, so that it can contact the remote SDN-controller. On the other side, C2 should send an *Insert* to the Request Customer Handler of its provider to establish the VPN. Since C2 is not interested in setting up such a VPN, it does not send any *Insert* to that component. Hence, the SDN-controller of the C2's provider replies to the SDN-controller of C1's provider that the federated VPN cannot be set up, because it did not receive any *Insert* from C2. This is the first type of checks.

The second type is the following. Suppose that two large customers, C3

and C4, want to establish a VPN, but they only allow a small subset of their end systems to exchange traffic. Suppose that ES3 is the set of allowed end systems of C3, whereas ES4 is the set of allowed end systems of C4. After the VPN is established , the two customers start to exchange traffic. Both the SDN-controllers issue forwarding rules to enable the communication among the end systems belonging to ES3 and ES4, so that if one of the customer tries to send traffic to or receives traffic from one end system that does not fall in the declared sets, the communication is forbidden. So, the SDN-controllers are acting as firewalls. We now describe the semantics of the 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 (e.g. the in_port can be inferred by the PacketIn message) and relying on utility functions made available by the underlying SDN framework (e.g. the Ryu framework offers functions to get auxiliary information about the OpenFlow switches, like their IP address); 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. 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. 5.1, an example of the primitive *Insert* sent by Customer1 to the Customer Request Handler of the ISP1's SDN-controller is Fig. 5.3 (we recall that Customer2 has to send a similar primitive to its provider):
Upon receiving an *Insert*, the Customer Request Handler 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 it creates a new <vpn> element assigning to it an id automatically generated and the name reported in the primitive, namely $C1 - C2 - vpn$. Then, it sends the primitive to the Remote Controller Handler, so that the latter forwards the information to the

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**

Figure 5.3: An example of insert primitive.

Remote Controller Handler of the remote provider.

Thus, the Customer Request Handler of the two providers are able to create the element <isp id="1" name="isp1.it"> as child of the element <vpn id="1" name="C1-C2-vpn"> for the Customer1 and Customer2 configurations, respectively. 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, information about the <datapath> element is added to the configuration. 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 the information is already in the configuration.

After performing those checks, the Remote Controller Handler of the ISP1's SDN-controller sends the primitive received by Customer1 to the Remote Controller Handler of the 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. 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, as well as to provision a service, is completely delegated to the Request Customer Handler, the SDN-controllers and the Remote Controller Handler.
Update - By using this primitive, a customer can modify what declared in the *Insert*. Indeed, *Update* takes as input the same parameters of *Insert*. Also, this primitive is used to keep the information of the customers updated.
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*. This information is propagated among the providers having customers in the declared VPN, so that the information in the federated network is consistent.

5.6 A Complete Example

In this section we provide a complete example of our framework, showing the whole interaction between two end-hosts, sharing *the same* IP address and belonging to two different customers connected to different ISPs' network. Referring to Fig. 5.1, 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 the customers; 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. 5.5, 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 an *Insert* to the Request Customer Handler, that is handled accordingly to the description in Sec. 5.5. After this step, each SDN-controller owns a 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

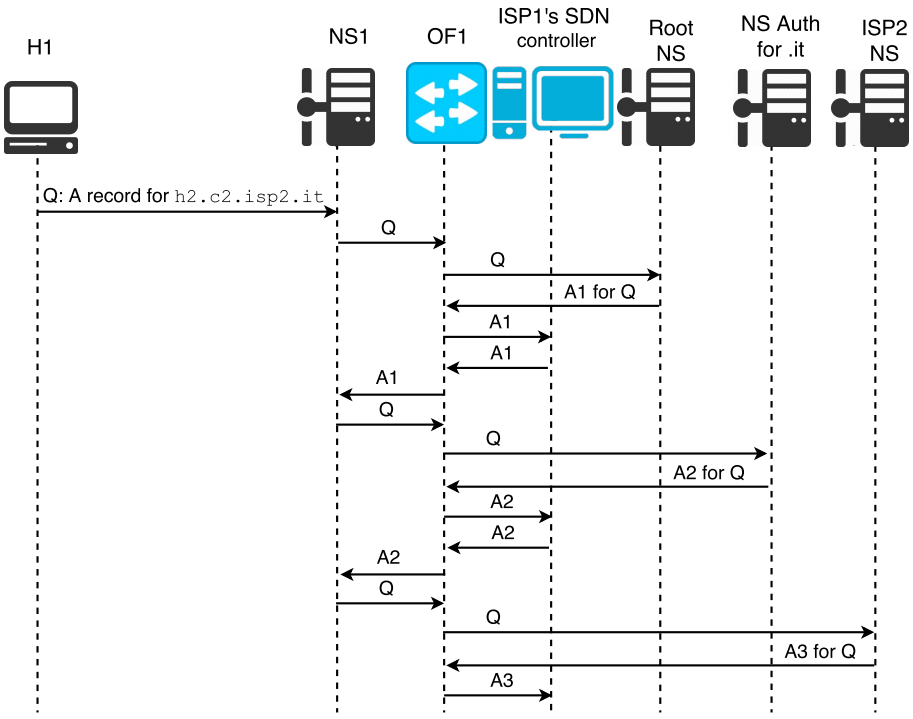


Figure 5.4: Interaction among OpenFlow switch, name servers, and SDN-controller in case of *Partial configuration* scenario.

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. 5.5. 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. 5.5. We call these scenarios *Full* and *Partial configuration*, respectively. In this chapter, we focus on the *Partial* scenario, since the *Full* is described in [MLB+17a].

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. 5.4). In Sec. 5.5, 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 that customers in different IPS's networks might share the same IP subnet.)

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.

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 communica-

tion 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, 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.

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. 5.5. 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 strate-

gies 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. 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: 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 (CAPS and LAPS strategies can be found in [dLRB16]) and we refer to specific OpenFlow messages defined in [Ope14].

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. 5.5. 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. 5.5. 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 des-

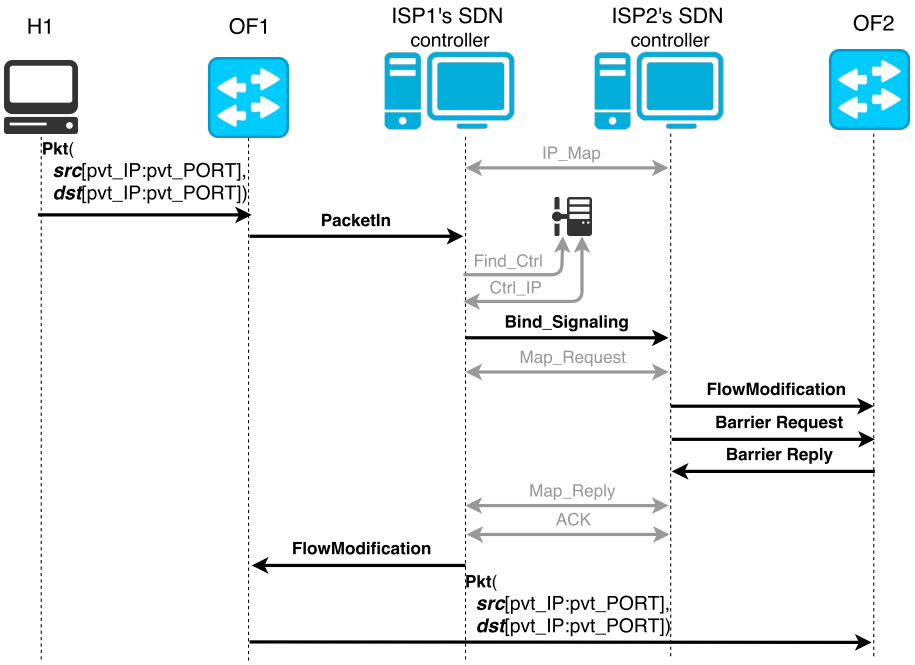


Figure 5.5: Messages exchanged to support communication between H1 and H2, for different translation strategies.

tination 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]→ **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]→ **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.

5.7 Takeaway

By relying on the SDN architecture and providing 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. Also, the traffic isolation typically provided by standard VPNs is still guaranteed to the extent that a provider adopts MPLS in its backbone. Indeed, our SDN-switches are placed outside the MPLS domain since they interconnect the CPE with the PE. Since the SDN-controllers of our framework instruct the SDN-switches to issue standard IP packets, they can be encapsulated into MPLS packets once they reach the PE routers, as in standard VPN fashion. Recalling the main challenges of Sec. 5.3, we now summarize how we solve those issues.

Management Problems – Avoiding additional interconnection points, each federated provider 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 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.

Drawbacks of our framework – We recognize a major drawback in our framework. Although we do not impose any constraints with respect to the protocol adopted by each provider inside the backbone, we assume that each federated ISP has (at least) an SDN controller plus an SDN-enabled switch collecting the traffic of its federated customers. On one hand, such assumptions do not impact the network architecture of the ISP; on the other hand, the SDN equipment might represent a cost for the provider. We argue that a reasonable trade-off for choosing our solution is the size of the federation (in terms of both ISPs and customers) and the flexibility degree that the federated partners want to achieve.

5.8 Evaluation

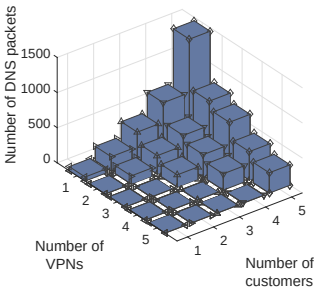
To validate the effectiveness of our framework, we implemented a prototype SDN-controller by relying on the Ryu framework [ryu17] and OpenFlow 1.3 [Ope14]. We focus our evaluation activity on both control and data plane, analyzing

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 do not evaluate monitoring and fast setup aspects. Indeed, we do not provide any contributions in terms of how to monitor the network: we just claim that we propose a solution for monitoring issues described in [FBP+10]. On the other hand, fast setup is achieved by the SDN-controller, in contrast to [gea17a].

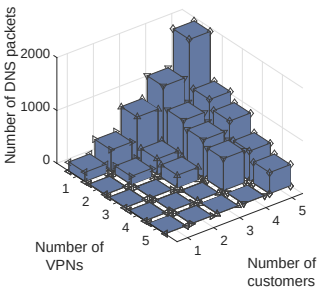
We run our experiments on SDNetkit [ML+17], an SDN-enabled enhancement of Netkit [net17], a widely used network emulator. Within SDNetkit, we used BIND [bin17] to implement name server functionalities and OpenVSwitch [ovs17] to implement OpenFlow devices. We run our SDN-controller on topologies reflecting the scenario in Sec. 5.4. Our implementation is composed by three main components: 1) *Primitive Handler*, to handle the primitives; 2) *DNS Handler*, to handle DNS messages; and 3) *Routing Handler*, to compute the routing.

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 $[1, 5]$. Then, we set up a number of VPNs varying in the range $[1, 5]$. 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.

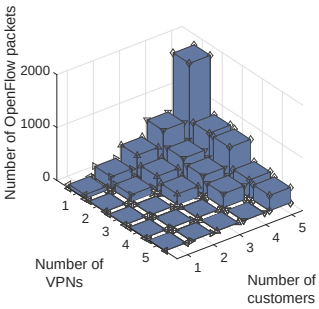
Control plane impact – To evaluate the impact of our implementation on the control plane, we measure the amount of DNS and OpenFlow packets exchanged in the network in order to allow a source to exchange traffic with a destination. We count the number of DNS and OpenFlow packets on each interface of 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



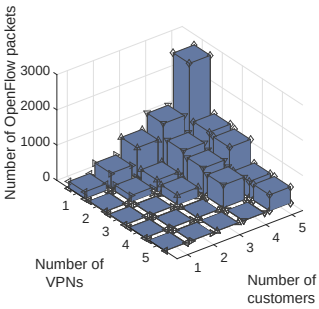
(a) Number of DNS packets exchange in the *Full configuration* scenario.



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

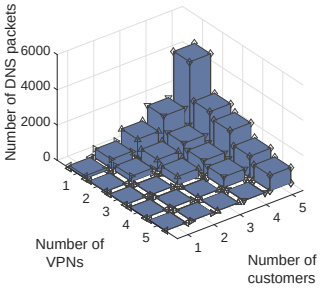


(c) Number of OpenFlow packets exchange in the *Full configuration* scenario.

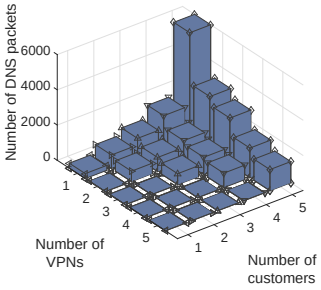


(d) Number of OpenFlow packets exchange in the *Partial configuration* scenario.

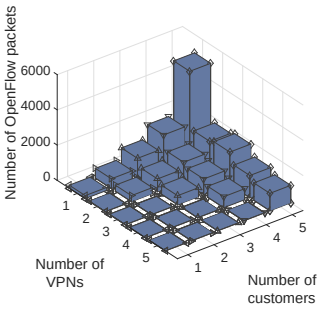
Figure 5.6: Control plane impact in a federated network consisting of two ISPs.



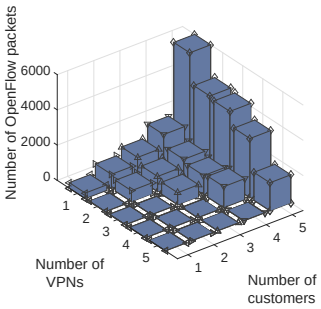
(a) Number of DNS packets exchange in the *Full configuration* scenario.



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

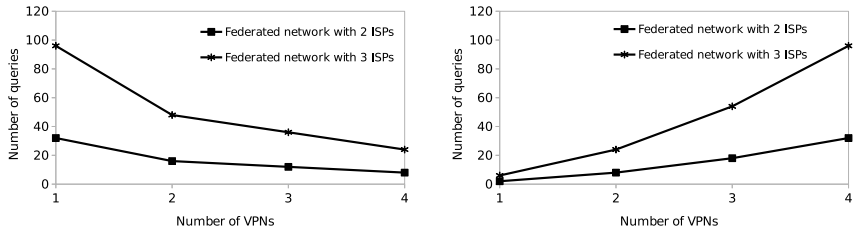


(c) Number of OpenFlow packets exchange in the *Full configuration* scenario.



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

Figure 5.7: 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 5.8: Number of queries in the federated network.

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. 5.6.

Fig. 5.6 shows the total number of DNS (Figg. 5.6(a) and 5.6(b)) and OpenFlow (Figg. 5.6(c) and 5.6(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. 5.8. Indeed, we observe that the number of queries grows with respect to the number of customers (Fig. 5.8(b)), while decreases with respect to the number of VPNs (Fig. 5.8(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. 5.7. Of course, in this case we observe a greater number of messages (almost 2000). Such an increase with respect to Fig. 5.6 has to be ascribed to the fact that adding a new ISP in the federa-

ted 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. 5.8. 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, retrieving that information from the DNS resolution process is a plus from a configuration point of view (see Sec. 5.6), but it represents a cost considering the number of exchanged messages, that is – in any cases – the same of any DNS service.

We also analyzed the correlation among the number of queries and the number of DNS messages. Pearson's correlation coefficient was used for this measurement. The correlation between the number of queries and the number of DNS messages in *Full* configuration scenario is 0.99 while for the *Partial* configuration scenario, this coefficient has a value of 0.90. This correlation for OpenFlow messages of *Full* configuration scenario is 0.99 and for *Partial* one is 0.90.

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).

We also provide an analytical model to count the number of required rules to install on the SDN-enabled switches. Assume that C is the number of customers per ISP. First, we count the number of required rules for *Full* configuration scenario. As foregone, we need one rule to handle ARP packets. To handle DNS packets, we need $2 \times C$ rules and for handling IP packets, the controller needs to install $4 \times C + 2$ rules. The number of OpenFlow rules in the *Full* configuration scenario (OF_{FC}) is

$$OF_{FC} = 1 + (2 \times C) + (4 \times C + 2) = 6 \times C + 3 \tag{5.1}$$

While for the *Partial* configuration scenario, we have the same number of rules

for ARP and $2 \times C + 2$ rules for handling DNS packets. Finally, we need $4 \times C + 2$ rules for IP packets. The overall number of OpenFlow rules for the *Partial* configuration scenario (OF_{PC}) can be computed as follows;

$$OF_{PC} = 1 + (2 \times C + 2) + (4 \times C + 2) = 6 \times C + 5 \tag{5.2}$$

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. 5.5. 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. 5.6. 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.

Finally, the evaluation shows that our system is able to manage federated networks with a small number of VPNs. Of course, having a single SDN-controller might be a limitation under different point of views (e.g. scalability and robustness). By increasing the number of SDN-controllers per ISP, our framework is able to handle a growing VPN demand.

5.9 Conclusions and Future Work

In this chapter, 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 6

A Decentralized SDN Architecture for IXPs*

Applications of Software-Defined Networking (SDN) to the Internet Routing come with great promise for supporting the ever-growing performance requirements posed by Internet applications. Yet, deployment in production of the most promising applications of SDN at the Internet eXchange Points (IXPs) like iSDX has so far been an elusive goal. We argue that the inherent centralization of these SDN approaches hinders real-world deployment for the following reasons: privacy (i.e., operators are reluctant to share private routing information), separation of responsibilities (i.e., the IXP running the centralized controller is involved in the routing and forwarding at too many levels), and scalability (i.e., the number of rules installed by the SDN controller is too large).

In this chapter, we take a new approach to applying SDN at IXPs, called DeSI. In our design, operators join the IXP with their own SDN equipment while requiring no modifications to the IXP fabric; thus, supporting separation of responsibilities. The members of the IXP use the SDN controller to exchange BGP messages, coordinate with other members, and install the forwarding state into the SDN switches connected to the IXP. We present two mechanisms to install this forwarding state that strike a different tradeoff between forwarding

*Part of contexts in this chapter is based on the following publication: Kumar D., Lospoto G., Mostafaei, H., Chiesa M., Di Battista G., DeSI: A Decentralized Software-Defined Network Architecture for Internet eXchange Points, under review in International Journal of Network Management.

space and processing time. Policies are never shared with unintended parties, thus satisfying privacy. To spur adoption, we introduce an expressive, yet simple, language to configure the routing policies of the members. We evaluate the practical feasibility of our system on Kathará, a system emulator for SDN and legacy networks, in order to test the resource consumption of our proposal.

6.1 Introduction

Modern-day Internet applications pose ever-growing performance requirements on the Internet. Such services require heterogeneous support for performance from the underlying network, including high-bandwidth (e.g., Cloud backup [MBHM13]) and low-latency (e.g., video streaming [SGH14]). Yet, the underlying network protocol used to determine the Internet paths through which domains send Internet traffic, *i.e., the Border Gateway Protocol* (BGP), is alarmingly oblivious to such performance metrics, ultimately hindering performance. Unfortunately, modifying BGP "overnight" has proven to be an elusive goal as it requires to reach some sort of wide consensus among independent network entities. Researchers and operators have therefore concentrated efforts to improve the status-quo at the emerging crossroads of Internet traffic, i.e., Internet eXchange Points (IXPs), where hundreds of organizations connect to exchange traffic at a reduced cost.

IXPs have traditionally acted as mere layer-2 interconnects that transit packets among BGP-speaking networks. The *Software-Defined-eXchange* (SDX) [GVS+15] is a revolutionary IXP architecture that brings the high programmability of Software-Defined Networking (SDN) [KRV+15a] to the IXP ecosystem. Both IXP operators and IXP members program the IXP fabric through a well-defined interface (e.g., OpenFlow [MAB+08]) to implement their routing policies. The potential impact of SDXes is huge: a recent work [CDA+16] showed the high benefits of improved Traffic-Engineering, security, traffic monitoring, network management, and more. Yet, the most notable SDX architecture, i.e., iSDX [GMB+16], has so far failed to be deployed in production for many reasons. First, several SDX architectures collect all the members' routing policies in a central controller owned by a third, possibly untrusted, entity. These policies dictate how packets should be routed at the IXP and therefore reveal potentially business information that is deemed confidential [CDC+17]. Second, SDXes solutions that install the forwarding policies of different IXP members on the same physical device (e.g., iSDX) may exacerbate any dispute regarding the separation of responsibility in case of failures in delivering traffic.

In fact, traditional Layer-2 IXPs clearly separates the responsibility of IXPs, i.e., transporting traffic between two statically configured MAC addresses, from selecting the routes through which sending traffic, which is left to the operators and does not involve any computation on a third party entity (e.g., the SDX controller). Third, solutions that install forwarding state in the IXP fabric tend to scale poorly in the number of members and configured policies. A recent study from a large IXP operator [HVSC16] showed that the forwarding state of recently proposed SDXes quickly explodes in the number of policies, ultimately refraining IXPs from deploying SDN solutions at IXPs.

We argue that any SDX architecture, e.g., iSDX, must satisfy the following requirements: *privacy*, i.e., the routing policies of the IXP members should not be disclosed to any unintended third party, possibly including the IXP itself, *separation of responsibility*, i.e., identifying who is responsible of what in case of outages should be easy, *forwarding state scalability*, i.e., the IXP fabric should scale to the size of the largest IXPs (and beyond), thus limiting the amount of forwarding state required to support the SDX architecture, and *expressive policy language*, i.e., IXP members need to forward packets according to their business- and performance-driven requirements.

In this chapter, we present our envisioned architecture for SDXes, called DeSI, that satisfies all of the above requirements. First, we argue that IXPs should not be involved in the route computation among members. Instead, in DeSI, IXP members connect to the IXP with their own SDN-enabled equipment, including an SDN switch to be connected to the SDX fabric as well as an SDN controller to configure the switch and coordinate with other IXP members. This fundamentally different approach to the design of SDX architectures comes with huge benefits in terms of privacy (policies are stored locally), separation of responsibilities (IXPs are not involved in the route computation), and forwarding state scalability (each IXP member only stores its forwarding state). In DeSI, IXP members use two complementary mechanisms for the installation of the forwarding state to scale the forwarding state. Through a *proactive* approach, an IXP member installs the whole forwarding state regardless of whether some rules are never matched by actual data packets. This approach has the benefit of quickly handling the incoming traffic but may result in overly large forwarding tables whose size may not be supported by the underlying SDN hardware [1] Through a *reactive* approach, the

[1] SDX routing policies rely on wildcard matching, thus requiring TCAM support from the underlying SDN switch. TCAM space is often limited due to being a power-hungry and expensive resource.

DeSI controller installs a forwarding rule only after a packet matching that rule is received by the SDN switch. With this approach, an operator limits the amount of forwarding state needed to support the defined forwarding rules but introduce additional latency and controller load overheads for each packet. DeSI relies on a combination of these two approaches to achieve forwarding state scalability. Finally, we introduce an expressive policy language, inspired by Pyretic [RMF+13], that can be used by the IXP members to define their routing policies.

We evaluated our system to assess its practical feasibility in term of physical resource consumptions. We observed that the most critical resource of DeSI, i.e., the member controller, scales well in the number of policies and BGP announcements being installed.

The reminder of the chapter is structured as follows. Sec. 6.2 reviews the most relevant contributions for the application of SDN to the inter-domain routing and IXPs. Sec. 6.3 briefly illustrates the current SDX-based architectures. Sec. 6.4 presents the architecture of DeSI. Sec. 6.5 shows our routing policy model, introducing basic concepts of our policy language. Sec. 6.6 introduces the reactive and proactive approaches, explaining the main differences between them. Sec. 6.7 explains the architecture of our SDN-controller and Sec. 6.8 states several applicability considerations. Sec. 6.9 shows the results we collected during the evaluation of our SDN-controller. Finally, Sec. 6.10 draws the conclusions and describes the research perspectives opened by our system.

6.2 Related work

In this section, we review the most relevant literature related to the application of SDN to IXPs. We describe the application of SDN to inter-domain routing, limiting the scope to those that are more related to ISPs. Table 6.1 gives an overview of the state-of-the-art, putting in evidence the differences among them.

An IXP can be seen as a LAN which each IXP member is connected to, in order to establish BGP peering with other members (ISPs). Typically, the switching fabric consists of a set of standard layer-2 switches.

Software-Defined Internet eXchange Point (SDX) [GVS+15] is the first attempt to apply SDN to the inter-domain routing inside IXPs. In SDX, standard BGP outbound policies are overridden by an SDN controller, improving the flexibility of the BGP protocol. The most relevant SDX contribution consists

Table 6.1: Comparison of SDN-based solutions for inter-domain routing.

Name	Number of Controllers	Switch Fabric	Forwarding Rules handled by	Dependency check	Path computed by
SDX	1	one switch	controller	not-known	controller and BGP
iSDX	1 + one per member	one switch	iSDX controller	not-known	controller and BGP
Endeavor	1 + one per member	several switches	Endeavor controller	not-known	Endeavor controller and BGP
SDI	one per member	not-known	ISP controller	not-known	controller
Edge Fabric	1	several switches	controller	not-known	controller and overriding BGP
Espresso	hierarchy of controllers	not-known	local controllers	not-known	global controller and BGP
DESI	one per member	IXP dependent (one or more switch)	ISP controller	yes	controller and BGP

in replacing the IXP switching fabric with an SDN-capable switch handled by a centralized controller which collects the policies of each member.

SDX suffers of scalability problems [GMB+16], in terms of control plane computation time and the number of generated forwarding rules. Those issues are addressed by an improved version called industrial-SDX (iSDX) [GMB+16]. In iSDX the control plane computation time is reduced by introducing, in addition to the IXP SDN controller, another SDN controller on the ISP side, so that the control plane computation is distributed along multiple controllers, while the number of forwarding rules is reduced by introducing compression mechanisms. The iSDX architecture imposes two strong limitations for the ISPs: 1) each of them must be equipped with an SDN controller and 2) they must share their policies in the centralized SDN controller inside the IXP. Also, identifying responsibilities becomes difficult in case of faults.

Herman et al. [HVSC16] report that currently employed switch platforms are incapable of supporting iSDX because of the lack of flow tables. Also, the policy compression mechanism of iSDX is too heavy for the current hardware.

Endeavour [ACC+17] reduces the number of installed rules on the SDN-enabled switch of an IXP switch fabric (70% less than those of SDX and iSDX). Endeavour is built on top of SDX [GVS+15], iSDX [GMB+16], and Umbrella [Bru16]. It proposes a new architecture for an IXP switch fabric which is composed of edge and core switches. The rules are installed on edge switches, while the core switches are in charge of forwarding traffic to its designated egress points. This architecture helps to improve the scalability of IXP fabric even if it adds duplication in forwarding state while installing the inbound and outbound routing policies of the participants. The proposed architecture introduces a mechanism to check (possible) dependencies among the forwarding rules.

In contrast to SDX, iSDX, and Endeavor, our architecture does not replace the IXP switching fabric with as an SDN-based one. Indeed, we introduce SDN only on the ISP side, without forcing other members to be equipped with an SDN controller. Finally, we preserve backward compatibility with providers that are not interested in using SDN on their side.

Software Defined Inter-domain routing (SDI) [WBL+16] provides the flexibility for routing policies based on the header fields of IP packets. SDI can check for the possible dependencies among the forwarding rules but fails to perform the BGP peering among the participants. It relies on SDI peering sessions among the members without leveraging the IXP peering LAN.

Edge Fabric [SKC+17] is an SDN-based Facebook solution to improve the capacity of the network to better steer Facebook traffic through the continents.

It aims at avoiding congestions in near real-time. Espresso [YMR+17] is a Google solution for the Google network. Espresso cares more about traffic engineering issues than generating rules to install on switching fabric. The idea behind the Espresso architecture is to scale cost-efficiently to Internet peering and allow application-aware routing for Google network. Edge Fabric and Espresso have been devised for Facebook and Google networks and it is not clear how and if they are applicable for general-purpose IXPs networks.

6.3 SDX based IXP Architectures

This section describes the typical architecture of an SDN-based IXP (we mainly concentrate on the latest version of SDX, which is called iSDX [GMB+16]) discussing the main components and explaining their functionalities.

Components

The main components of the architecture are the following.

1) **An SDN-enabled switch.** To program the switching fabric of an IXP, there is the need of at least one SDN-enabled switch. It is the collector of the policies of all the ISPs that participate to the IXP.

2) **A BGP route server.** Currently, the most important IXPs offer a *route server*. An ISP can substitute its bi-lateral peerings with just one peering with the route server. The route server computes the best routes to reach the target prefixes and redistributes such routes to the ISPs. The iSDX route server is implemented with ExaBGP [EN16].

3) **An IXP controller.** The controller cooperates with the route server to integrate the BGP policies with custom outbound and inbound policies.

4) **The Members' SDN-controllers.** Each participant to the IXP can have its own SDN-controller that shares part of the computations performed by the IXP controller. This improves the scalability of the architecture.

5) **The Members' border routers.** Each member runs (at least) one border router to exchange BGP messages with the router server. The route server can check the BGP reachability information of each member by checking the BGP update messages that come from these devices.

SDX and iSDX Architectures

SDX was the first attempt to bring SDN to inter-domain routing, but it does not scale for the following reasons: 1) it generates many SDN rules to handle traffic and currently available TCAM size for SDN-enabled switches is not able to maintain them [HVSC16] and 2) the computation time to generate low-level forwarding table entries from high-level forwarding policies, which may change the forwarding behavior of BGP [GMB+16], is high.

It is worth stating that SDX and iSDX inherit the same approach in rule generation because they used the same mechanism with different compression method to generate the initial rules from SDN-policies of participants.

There are two key design improvements in iSDX with respect to the original SDX proposal. First, the control plane computations of iSDX are partitioned among the participants' controllers which ensures that the routing policies of a participant remain independent from the others. Second, the BGP and the SDN policies are kept separate. This avoids the recomputations that can be triggered when new updates are received. Once the control plane computation is carried out, a forwarding equivalence class (FEC) for each member is created, in order to allow the forwarding of the traffic. For this goal, the reachability information is encoded inside a tag which is stored inside the destination MAC address field of the packets' header. To do so, the multiple match-action tables feature of an OpenFlow-enabled switch is leveraged. iSDX uses one table for inbound and one for outbound policies of each participant. A virtual MAC address is used to encode the reachability information. We remark that a MAC address consists of 48 bits. Considering a very large IXP with 1024 participants, iSDX needs 10 bits to encode the next-hop ASes. The remaining 37 can be exploited for encoding the received prefix from participants. According to iSDX [GMB+16] each unique prefix is announced at most by 27 participants which requires one bit per member to encode them. By using 6 bits for the bitmask, the iSDX could encode the reachability information of each announced prefix. This encoding is performed in the inbound table of a participant as the virtual MAC (VMAC) and will be replaced by the real MAC of destination in the outbound table of the receiver. This way of encoding the reachability information is called hierarchal encoding. To keep the VMAC updated, many gratuitous ARP messages are injected into the IXP's LAN, increasing the overhead of the whole network.

We leverage the same mechanism for encoding the reachability information for each prefix, but we rely on the OpenFlow metadata registry.

Limits of the Existing SDX Architectures

Although the proposed architectures are the result of a deep and sophisticated research work, up to now very few IXPs adopted SDX or iSDX technologies (as far as we know, just one IXP is based on SDX [ACC+17]). This, in our opinion, depends on the following main aspects.

The first issue is the privacy of routing policies. Current architectures do not offer a guarantee on the privacy of the policies of the participants. Both the IXP SDN-controller and the Route Server have shared equipment. Anybody that is allowed to enter such machines can access information that can unveil (totally or partially) the policies of the members [CdLL+17].

A second issue is that the proposed architectures do not allow to clearly separate and identify who is responsible for what in case of outages. Namely, in a traditional IXP the center of the architecture is a basic layer-2 switch, with limited intelligence and limited capabilities (in large IXPs this is substituted by more complex layer-2 switch fabric; however, its overall behaviour is the one of a simple switch). This allows, in case of problems, to clearly separate the responsibility of members and the responsibility of the IXP. In SDX and iSDX the central SDN-enabled switch and the IXP controller are sophisticated machines where the policies of all participants are mixed into a unique container. This does not allow to have a clear boundary between the ISPs and the IXP.

A third issue is the scalability. Integrating into a unique switch the policies of a large-size IXP can be unfeasible [HVSC16].

Finally, the current architectures do not provide any service for checking the consistency of members' policies. Namely, SDX and iSDX lack a component that is devoted to check whether one or more policies might compromise the effectiveness of other policies, resulting in an undesired traffic forwarding.

6.4 A New SDN Architecture for Internet eXchange Points

In this section, we describe our architecture for an SDN-based Internet eXchange Point. We point out the main differences between SDX and DeSI, in order to show how we overcome the limitations imposed by the SDX architecture.

Our architecture is depicted in Fig. 6.1. Each provider joins the IXP with its own SDN-enabled switch and its own SDN-controller. In the figure, there are three providers, whose names are ISP1, ISP2, and ISP3. They are connected to the IXP peering LAN by means of SDN-enabled switches, which are called

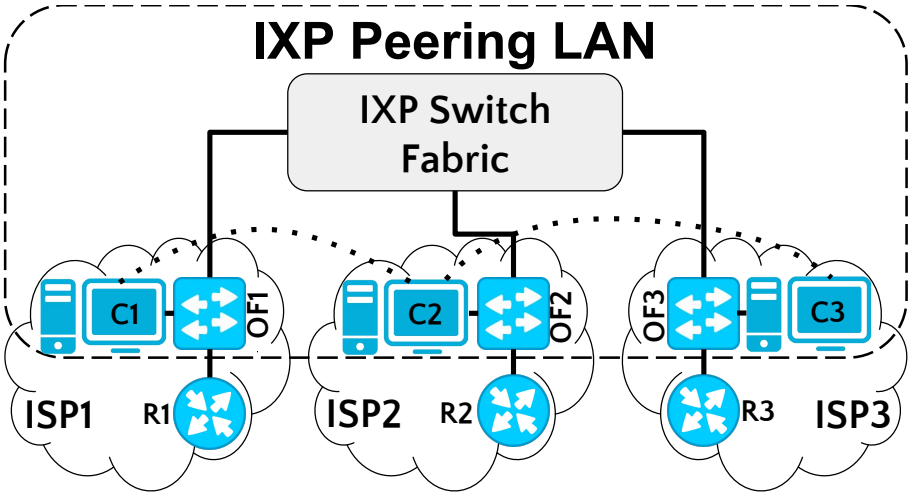


Figure 6.1: Our architecture in which SDN is moved in the provider's side in order to avoid policies sharing and to easily identify responsibilities.

OF1, OF2, and OF3. Each SDN-enabled switch is handled by a specific SDN-controller, namely C1, C2, and C3. Such an architecture does not impose any limitations on the possibility for a provider to be interconnected to multiple IXPs. We provide more details about this scenario in Sec. 6.8.

In Fig. 6.1, routers R1, R2, and R3 are IP-speaking devices directly connected, on a specific port, to the SDN-enabled devices and they represent the whole network of each provider. Note that the IXP switch fabric is not SDN-based. We decided to move SDN capabilities inside the network of each provider. Such a choice has two advantages: first, policies are not stored anymore in a centralized place; second, each provider independently acts on its own SDN-enabled device, still having the flexibility offered by SDN, but avoiding the possibility of compromising the policies of any other IXP member connected to the IXP.

Still referring to Fig. 6.1, while the bold lines are physical connections, the dotted ones represent BGP peerings. This is another change we introduce. In order to allow each provider to be as flexible as possible, we assume that the peerings are established between SDN-controllers. T The most valuable benefit is the possibility of easily interacting with the BGP control plane table which enables the chance of using multiple paths for the inter-domain routing.

Another consideration regarding the choice of publicly exposing the controller on the peering LAN. We argue that such a situation is not dangerous for IXP members, for two reasons. First, the peering LAN is typically assumed to be trusted; second, only the BGP speaker component of the SDN-controller is publicly exposed on that LAN.

6.5 A Routing Policy Model

In this section we describe our routing policy model, which is based on a language allowing each provider to forward traffic along multiple paths. Also, we discuss the semantic of our language, highlighting its main properties.

Our language does not replace the BGP configuration of members for outbound and inbound policy specification. Rather, it can be used in conjunction with the BGP policy specification language in order to extend the standard BGP capabilities. So, the backward compatibility with standard BGP speakers is preserved. Also, we do not exploit Pyretic [RMF+13], since it is based on the POX controller and it imposes constraints on the controller to use, while our proposal is more general. We now introduce the model that represents the building block on which we build our language.

We model the IXP as the set of the ISPs connected to the IXP itself. Let $\mathcal{I}$ be such a set. Also, let $BGP \subseteq \mathcal{I} \times \mathcal{I}$ be the set of all the BGP peerings established at the IXP. Given any two providers $i_1, i_2 \in \mathcal{I}$, we say that i_1 and i_2 establish a BGP peering if and only if $(i_1, i_2) \in BGP$. All the BGP neighbors of an ISP are modeled as a set: $\forall i, j \in \mathcal{I}$, if $(i, j) \in BGP$, then $j \in \mathcal{N}_i \subset \mathcal{I}$ and $i \in \mathcal{N}_j \subset \mathcal{I}$, namely j belongs to the set of all the BGP neighbors of i and i belongs to the set of all the BGP neighbors of j, respectively. Finally, given two ISPs i, j such that $j \in \mathcal{N}_i$, we define $\mathcal{P}_i^j$ as the set of all the IP prefixes announced by j to i.

Each provider specifies a set of policies to route traffic through the Internet. Given a provider $i \in \mathcal{I}$, P_i is the policy set of i. We define a *policy* $p \in P_i$ as a pair $p = \langle match \rightarrow neighbors \rangle$. The *match* is a (possibly empty) *expression*. The operators of the expression are the logical operators AND ($\wedge$) and OR ($\vee$). In our language, expressions including the AND operator are evaluated before of those including the OR operator. The *atomic elements* of the expression are relational conditions in the form $atom = value$. Each *atom* is an element of the quadruple $\langle srcip, dstip, srcport, dstport \rangle$, where: 1) *srcip* is a source IPv4 or IPv6 address; 2) *dstip* is a destination IPv4 or IPv6 address; 3) *srcport* is a source TCP or UDP port; and 4) *dstport* is a destination TCP or UDP port.

None of the elements is mandatory: a policy without any match condition means *all the traffic*. We define M_p as the set of all the atoms in the match conditions of p. Note that M_p can be extended by including any matchable field defined in the OpenFlow specification [MAB+08].

The *neighbors* part of the policy $p \in P_i$ is a list of neighbors $\mathcal{N}_p \subseteq \mathcal{N}_i$, that are candidates to receive the packets that match the policy. Hence, we assume that a single type of action exists within each policy whose semantic is: *(potentially) forward to a neighbor*. Such an assumption is not restrictive, since our language does not replace the BGP policy specification.

Consider the following example. Referring to Fig. 6.1, suppose that ISP1 has two BGP peerings, one with ISP2 and another one with ISP3 (this is made possible by the interconnection with the standard IXP fabric switch), and suppose that it wants to send a portion of its outgoing traffic to ISP2 and another portion to ISP3, according to some field of the packet header. By using a standard BGP policy specification language, this is not feasible, since BGP computes a single best path and all the traffic is forwarded along that path.

Given $ISP1, ISP2, ISP3 \in \mathcal{I}$ and $ISP2, ISP3 \in \mathcal{N}_{ISP1}$, consider the following policies $p_1, p_2 \in P_{ISP1}$:
$p_1 = \langle dstip = 20.1.2.0/24 \wedge dstport = 80 \rightarrow (\text{ISP2,ISP3}) \rangle$
$p_2 = \langle dstport = 21 \vee dstport = 22 \rightarrow (\text{ISP3}) \rangle$
Also, consider a prefix $\pi = 20.1.2.0/24$ such that $\pi \in \mathcal{P}_{ISP1}^{ISP2}$ and $\pi \in \mathcal{P}_{ISP1}^{ISP3}$. The semantic of p_1 is: *send all the traffic whose destination IP address falls in the subnet 20.1.2.0/24 and whose destination port has value 80 to ISP2. Else, (either ISP2 does not announce that prefix or it is not reachable for temporary connectivity problems), send that traffic to neighbor ISP3.* The semantic of policy p_2 is: *send all the traffic whose destination port has value either 21 or 22 to neighbor ISP3.* Observe that p_1 and p_2 use a subset of the available atoms. If not specified in the policy, an atom is considered as a wildcard. Also, the BGP routing must support the traffic forwarding through the neighbors specified in the *actions* part of the matched policy.

Note that policy p_1 allows the traffic to be forwarded to ISP2 even if ISP2 is not the best choice for BGP. In order to send that traffic to ISP2 it is enough that ISP2 announces prefix 20.1.2.0/24. We augment the semantic of a policy by implicitly stating that if none of the neighbors specified in the action announces the prefix mentioned in the match, then the traffic is forwarded according to the BGP computation, even if the neighbor to which the traffic is being forwarded is not mentioned in the *actions* part of the policy.

Finally, our language allows for a double level of priority level. Indeed, the first level is expressed inside the policy when multiple neighbors are defined in

the *actions* list, as reported in policy p_1. In that case, *forwarding traffic to neighbor ISP2* has higher priority than *forwarding traffic to neighbor ISP3*. The second priority level is among policies. In the example, policy p_1 has higher priority than policy p_2. This means that policy p_1 must be checked always before policy p_2 and the latter one can be considered by the SDN controller if and only if traffic cannot be forwarded according to policy p_1. Hence, the policy priority levels are defined by the order of the policies themselves. Such an ordering might lead to a problem that we call *Covering Problem*.

The Covering Problem

As we just said, the order of the policies defines their priority. It is important to note that such an order could lead to the situation in which a policy will be never selected, even if it should. This circumstance might happen due to human error. This situation can happen either in proactive or reactive approaches. Before the formal definition of such a problem, we show this with an example. Referring to Fig. 6.1, suppose that ISP1 wants to forward traffic with $dstport = 80$ to neighbor ISP2 and traffic with $dstport = 80$ and the source IP address falling in the subnet 2.0.0.0/8 to the neighbor ISP3. The ISP1's network administrator might write the following two policies:
$p_1 = \langle dstport = 80 \rightarrow (\mathsf{ISP2}) \rangle$
$p_2 = \langle dstport = 80 \wedge srcip = 2.0.0.0/8 \rightarrow (\mathsf{ISP3}) \rangle$
Also, suppose that both ISP2 and ISP3 send BGP announcements for the same IP prefix π. Now, suppose that two flows must be forwarded according to those policies. In particular, the flows have the following (portion of the) header:
$f_1 = \langle srcip = 1.0.0.1, dstip = 3.0.0.1, srcport = 10, dstport = 80 \rangle$
$f_2 = \langle srcip = 2.0.0.1, dstip = 3.0.0.1, srcport = 11, dstport = 80 \rangle$
and the IP address 3.0.0.1 belongs to the announced IP prefix π, so that it can be reached through ISP2 or through ISP3.

In the intention of the ISP1's network administrator, flow f_1 must be forwarded according to policy p_1, whereas the flow f_2 must be forwarded according to policy p_2. Suppose that the first packets that arrive belong to f_1. Then, the SDN controller selects policy p_1, installing into the open-flow switch the corresponding OpenFlow rule. Upon receiving the flow f_2, the SDN-enabled switch already has a rule to use. Hence, that flow is forwarded according to the OpenFlow rule installed after the selection of policy p_1, resulting in a policy mis-usage. We call such a problem *Covering Problem*, since policy p_1 *covers* policy p_2, preventing its selection. In contrast to [KARW16], we do not aim at finding dependencies for performance purposes. Indeed, we aim at guaran-

teeing that the forwarding is performed according to what an ISP wants to achieve.

Before formally showing the *Covering Problem*, we define the *priority* of a policy: given $i \in \mathcal{I}$ and $p \in P_i$, $pr(p)$ is the priority value of p.

Definition. *Given $i_1, i_2, i_3 \in \mathcal{I}$ and $p_1, p_2 \in P_{i_1}$ such that $pr(p_1) > pr(p_2)$, we say that p_1 **covers** p_2 if the following two conditions are satisfied: 1) $\forall n \in \mathcal{N}(p_1)$ and $\forall m \in \mathcal{N}(p_2)$, $\mathcal{P}_{i_1}^{i_2} \bigcap \mathcal{P}_{i_1}^{i_3} \neq \emptyset$ and 2) $M_{p_2} \subset M_{p_1}$.*

Roughly speaking, the *Covering problem* states that, given any two policies of the same provider, the policy with the higher priority value must be the policy with the largest match condition set, if one of the two policies has the match condition set that fully includes the other. Note that the *Covering problem* does not occur if the match condition sets of any two policies partially overlap.

To overcome the covering problem, it is enough to give higher priority to the covered policy (p_2 in the example). This results in writing the policies (p_1 and p_2) in the reverse order. Hence, given $i \in \mathcal{I}$, we say that the set $P_i^{cf} = \{p_2, p_1\}$ is the set of cover-free policies, where:
$p_2 = \langle dstport = 80 \wedge srcip = 2.0.0.0/8 \rightarrow (\mathsf{ISP3})\rangle$ and
$p_1 = \langle dstport = 80 \rightarrow (\mathsf{ISP2})\rangle$
Note that: 1) the policies set P_i^{cf} is not affected by the covering problem, and 2) such a new policies order makes $pr(p_2) > pr(p_1)$. If the order of the policies induces a covering problem, DeSI must arise a notification, without undertaking any specific action (e.g. by executing any re-ordering algorithm for the policies). This means that this problem does not depend on the BGP announcements, since it is only a static check of the policies.

6.6 From Policies to Forwarding Rules

After a computational process inside the SDN controller, a policy is translated into one or more suitable forwarding *rules* to be installed inside each SDN-enabled switch (e.g. OpenFlow rules). Such rules allow the device to forward the traffic to the proper neighbors. In the rest of the section we describe such a process, which consists of a sequence of steps.

Before explaining the process of generating forwarding rules from policies, we clarify the difference between policies and forwarding rules. While a policy represents a high-level way to declare how traffic must be routed in the network, a forwarding rule is the translation of that policy, resulting in suitable data

structures which are deployable into the SDN-enabled switches. In our case, each policy is translated into one or more OpenFlow rules. More details are given in Sec. 6.9.

We present two approaches, that we call *Reactive Approach* and *Proactive Approach*. The first one performs the translation from policies to rules when the traffic reaches the switch, whereas the latter one computes such a translation before any packets reach the device. Of course, each approach comes with its benefits and drawbacks. Before going deep in the description, we discuss several differences between them.

The reactive approach performs the translation from policies to rules once the traffic reaches the SDN-enable switch. Such an approach allows the SDN-controller to be fast during the start-up phase, namely then the controller is started, whereas it might be slower when the traffic reaches the device, forcing the packets to be buffered in the SDN-enabled switch waiting for a forwarding rule. Indeed, computing the translation can be time-consuming due to the amount of control plane information to handle.

On the other hand, the proactive approach is complementary. It reduces the amount of time that the packets must wait in the device's buffers before being forwarded. Indeed, the most benefit of this approach consists in allowing the traffic to be immediately forwarded as soon as it reaches the device, since the rules are already deployed. On the other hand, the start-up phase might need a lot of time, especially if the full routing table is announced by some neighbors and needs to be processed in order to perform the translation from policies to forwarding rules.

For those reasons, we argue that the reactive approach is a suitable solution for those providers which handle few amount of entries in their routing tables (e.g. small providers with few upstreams which announce them the default route). The proactive approach can be adopted by big providers that directly handle the full routing table and need to be fast in forwarding a big amount of traffic (e.g. transit Autonomous Systems).

Thus, there are no limitations in adopting one of the two approaches under different conditions, provided that benefits and drawbacks of each approach are evaluated in advance.

We describe the details of those approaches in the following.

The Reactive Approach

In the reactive approach several conditions must be taken into account. First of all, there could be *dependencies* among policies. If such a situation happens,

the SDN-controller must be able to detect it and acts properly. To support that task, we define the *Dependency Graph*, which is a graph modeling specific relations among policies.

The Dependency Graph

The policies set P_{cf}, shown in Sec. 6.5, is not affected by the covering problem. However, such a policies set might be affected by another issue. Suppose that a traffic flow matching the policy p_2 (e.g. $dport = 80$) reaches the SDN-enabled switch and there is no suitable rule in the forwarding table of the switch to match that flow. According to the reactive approach, a packet of that flow is sent to the controller, so that it selects the suitable policy to apply. After selecting the policy p_2 and translating it into a forwarding rule, the SDN-enabled switch is able to route the packets.

Now, suppose that a new traffic flow arrives at the same SDN-enabled switch. That traffic still has in the header of the packets the value $dstport = 80$, but the source IP address now falls in the subnet 2.0.0.0/8. The traffic flow must be forwarded according to the policy p_1, but it is not, since that traffic flow matches the previously installed forwarding rule (e.g. the rule obtained from the policy p_2).

Through this very simple example, it is evident that two policies, or more, might depend on each other. In particular, this is true when a lower priority policy is matched before a higher one. To avoid this problem, we introduce the concept of *Dependency Graph*. The dependency graph is a directed graph $G = (V, E)$ modeling dependency relationships among the policies, in which: 1) V is the set of the vertices. Each vertex represents a policy. Hence, we say that $V = P$, and 2) E is the set of the edges. Each edge is a pair $\langle v_1, v_2 \rangle$ where $v_1, v_2 \in V$. Since the graph is oriented, the pair $\langle v_1, v_2 \rangle$ represents an edge from v_1 to v_2.

Given a set of policy P, the dependency graph for that set of policy is a graph $G = (P, E)$ where P is the set of the vertices, each of which models a policy, and $\forall p_i, p_j \in P$ where $i \neq j$, $\langle p_i, p_j \in E$ if and only if the two following conditions are satisfied: 1. $Pr(p_i) < Pr(p_j)$ and 2. $M(p_i) \cap M(p_j) \neq \emptyset$. It is easy to note that the graph $G = (P_{cf}, E)$ is the dependency graph for the set P_{cf}, where $P_{cf} = \{p_1, p_2\}$, and $E = \{\langle p_2, p_1 \rangle\}$. In fact, $Pr(p_2) < Pr(p_1)$ and $M(p_2) \cap M(p_1) = \langle dstport = 80 \rangle$.

We highlight that the covering problem and the dependency graph address two different problems, but complementary. In particular, the covering problem is the problem of a higher policy which prevents the selection of a lower one,

whereas the dependency graph is a data-structure aiming at avoiding to forward the traffic according to a lower priority policy in place of a higher one, if present. Alg. 1 builds the dependency graph.

Algorithm 1 Creating dependency graph among the policies

```
1:  Input The set of policies (P) for a controller
2:  Output The set of dependent policies.
3:  procedure CreateGraph
4:      state all policies as M(p)
5:      for each p ∈ P do
6:          create a vertex for each policy p
7:      end for
8:      M(p_i) ← match fields of vertex i
9:      M(p_j) ← match fields of vertex j
10:     i ← pr(p)                    ▷ start with a policy with a higher priority
11:     for i=1 to P do
12:         pick a policy p
13:         pick corresponding vertex for p
14:         for j=1 to j < i do
15:             pick the policy for vertex j
16:             if M(p_j) ⊂ M(p_i) then
17:                 add an edge from vertex i to vertex j
18:             else if (M(p_j) ⊉ M(p_i)) ∧ (M(p_j) ∩ M(p_i) ≠ ∅) then
19:                 add an edge from vertex i to vertex j
20:             end if
21:         end for
22:     end for
23: end procedure
```

At the beginning, the set of all the vertices of the graph is built (lines 4-7). After that, given any pair of policy, they are represented by means of their match part (lines 8 and 9). Each policy is now compared with each other: if a policy j has the match condition set that is a subset of the match set of a lower priority policy i (lines 16 and 17), then an edge from vertex i to j is added, as well as in lines 18 and 19. After building the dependency graph, the SDN-controller is now able to produce the suitable set of forwarding rules that allow the traffic to be forwarded without any mistakes. This step is called *Expansion Process* and we explain it in the following.

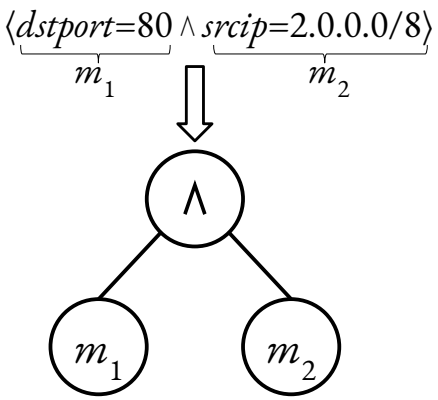


Figure 6.2: Graph representation of a policy only containing the AND operator.

The Expansion Process

In this section, we describe the policy expansion process, namely how the policies are translated into forwarding rules. We explained in Sec. 6.5 that our routing policy model uses two operators: AND and OR. Since the match part of an OpenFlow flow entry can be seen as a sequence of match conditions evaluated by using the AND operator (e.g. a packet matching ***all*** the fields specified in the match condition), we start our explanation by considering a policy whose match part consists of a set of matching conditions using the AND operator. After that, we show how policies including the OR operator are translated into an equivalent set of policies which only use the AND operator, making the translation process straightforward.

As the first step of the expansion process, we build a tree for each policy in which the parent node in the tree indicate the used operator and the leaves of the tree show the match fields. Fig 6.2 depicts an example of such a representation. After the tree is built, we run on it a Depth-First Search (DFS) algorithm on it in order to create the forwarding rule to install on the device. According to Sec. 6.5, we consider *wildcard* ($*$) for all the other match conditions which do not explicitly appear in the policy itself, and we assume them in AND with all the other match conditions.

Relying on this representation, we are now able to represent a policy containing the OR operator into a set of policies only containing the AND operator.

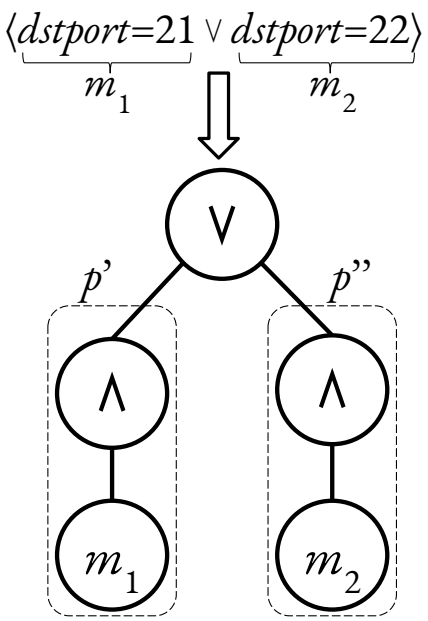


Figure 6.3: Tree representation of a policy containing the OR operator.

Consider the policy:

$p = \langle dstport = 21 \vee dstport = 22 \rightarrow (\mathsf{n}) \rangle$
According to the representation we just described, we build the tree shown in Fig. 6.3. We run the DFS algorithm on this tree. Each time a node containing an OR operator is visited, a new sub-policy is created. By doing so, the policy p leads to two policies, which we call p' and p'', that only contain the AND operator. In particular, those two policies are:
$p' = \langle dstport = 21 \rightarrow (\mathsf{n}) \rangle$
$p'' = \langle dstport = 22 \rightarrow (\mathsf{n}) \rangle$
To better clarify the presence of wildcards and operators, p' and p'' can be seen in the following way:
$p' = \langle srcip = * \wedge dstip = * \wedge srcport = * \wedge dstport = 21 \rightarrow (\mathsf{n}) \rangle$
$p'' = \langle srcip = * \wedge dstip = * \wedge srcport = * \wedge dstport = 22 \rightarrow (\mathsf{n}) \rangle$

As second step of the expansion process, we actually *expand* the policy.

By expanding, we mean to check whether the BGP routing allows the policy itself. To do that, we need to interact with the BGP Routing Information Base (RIB). Indeed, once a packet arrives in the switch, a policy is selected. Then, we check whether there is an entry in the BGP RIB allowing that packet to be forwarded according to the *action* part of the policy. If so, the policy is expanded. This means that, if no destination IP address is specified in the match part of the policy, that IP address is added, after a lookup in the RIB. We describe it in more detail.

Suppose that a packet matching policy p arrives at the SDN-enabled switch, whose destination IP address is π. Since no destination IP address is specified in p ($dip = *$), that value must be specified, in order to avoid possible mismatch with other forwarding rules. So, a lookup in the BGP RIP is carried out, checking whether $\pi \in \mathcal{P}(n)$. If it is the case, the policy p is *expanded* by setting $dip = \pi$. Hence p becomes:
$p = \langle (dstport = 21 \vee dstport = 22) \wedge dstip = \pi \rightarrow (\mathsf{n}) \rangle$.

The expansion process represents the last step, and it results in the creation of a set of forwarding rules. As anticipated in Sec. 6.4, we now describe the proactive approach.

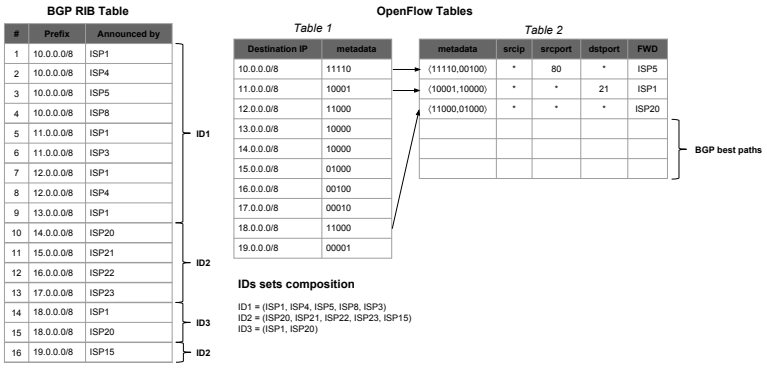
The Proactive Approach

In proactive approach all the policies are translated into forwarding rules before the traffic flows reach the SDN-enabled switch, namely during the startup phase of the SDN-controller.

As reported in Sec. 6.3, iSDX relies on the destination MAC address to encode all the reachability information, namely the set of neighbors which a flow can be sent to. We rely on the same encoding iSDX implements, but we exploit the metadata registry, an available data-structure in the OpenFlow specification [Ope18]. It has two advantages: first, we can encode more information, since the metadata consists of 64 bits, whereas the MAC address is 48 bits long; second, since we do not change the destination MAC address, we do not need to inject additional ARP traffic in the network, reducing the overhead of the whole IXP's peering LAN.

Now, we show how we exploit the OpenFlow metadata to encode network reachability information. We assume that:

1. the BGP RIB table is ordered based on the announced IP prefixes;
2. the metadata is a pair $\langle ID, mask \rangle$ and they have the same length. This assumption is natively supported by the OpenFlow specification [Ope18].



#	Prefix	Announced by
1	10.0.0.0/8	ISP1
2	10.0.0.0/8	ISP4
3	10.0.0.0/8	ISP5
4	10.0.0.0/8	ISP8
5	11.0.0.0/8	ISP1
6	11.0.0.0/8	ISP3
7	12.0.0.0/8	ISP1
8	12.0.0.0/8	ISP4
9	13.0.0.0/8	ISP1
10	14.0.0.0/8	ISP20
11	15.0.0.0/8	ISP21
12	16.0.0.0/8	ISP22
13	17.0.0.0/8	ISP23
14	18.0.0.0/8	ISP1
15	18.0.0.0/8	ISP20
16	19.0.0.0/8	ISP15

Destination IP	metadata
10.0.0.0/8	11110
11.0.0.0/8	10001
12.0.0.0/8	11000
13.0.0.0/8	10000
14.0.0.0/8	10000
15.0.0.0/8	01000
16.0.0.0/8	00100
17.0.0.0/8	00010
18.0.0.0/8	11000
19.0.0.0/8	00001

metadata	srcip	srcport	dstport	FWD
(11110,00100)	*	80	*	ISP5
(10001,10000)	*	*	21	ISP1
(11000,01000)	*	*	*	ISP20

Figure 6.4: Example of how to use metadata to encode network reachability information.

We use two OpenFlow tables to forward traffic according to the policies defined at the SDN-controller.

Fig. 6.4 shows an example of how we encode network reachability information in the OpenFlow metadata. To do that, we rely on several sets. Each element of these sets is an ID, namely a value that uniquely identifies a provider (e.g. the Autonomous System number). Two conditions are needed to create a new set. These conditions do not have to simultaneously occur. Each set is created when two conditions happen. The values for the metadata registry are built relying on those sets. For simplicity, we assume that the metadata is 5 bits length, so that each IDs set exactly contains 5 elements as well which will be used as mask. The proactive approach consists of three steps: 1) building the set of the IDs, that are used to populate the metadata registry. Since we assume that metadata is 5 bits length, each ID strictly contains only 5 elements as well; 2) filling the OpenFlow Table 1; and 3) filling the OpenFlow Table 2.

First, our controller builds the set of IDs. This step is accomplished in the following way. At the beginning, no IDs sets are present. Then, our controller starts to scan the BGP RIB table. It finds that provider ISP1 announces the prefix 10.0.0.0/8. So, the first set, called ID1, is created and the first element of that set is the IS ISP1. Still scanning the BGP RIB table, ISP4 announcing 10.0.0.0/8 is the second entry found. Since we have space in the set ID1 (4 bits are still available), ISP4 is included in that set. Such a process continues until the set ID1 is full. This condition happens when reaching the BGP RIP

entry number 9. Once the entry number 10 is being scanned, a new ISP is found, namely ISP20. Since the set ID1 is full, a new set, called ID2, is created and the bit in position one correspond to ISP2. This is the first condition that triggers the creation of a new IDs set. The second condition that triggers the creation of a new set is the following. If a prefix is announced by two or more providers that are already inside two or more IDs sets, then a new IDs set is needed. Such a condition applies when scanning lines 14 and 15 of the BGP RIP table. Indeed, the prefix 18.0.0.0/8 is announced by providers ISP1 and ISP20. Since ISP1 is in set ID1 and ISP2 is in set ID2 (because of prefixes 10.0.0.0/8 and 14.0.0.0/8, respectively), a new set, called ISP3, is needed. The first step is accomplished.

The second step, namely filling the OpenFlow Table 1, is carried out. The first OpenFlow table contains an entry for each prefix in the BGP RIB table. Each entry is associated with a metadata value that is built in the following way: there is a bit set to 1 for each provider that announces the prefix. The choice of which bit is set to 1 depends on the position of the ID in the set. As an example, consider the first entry of the OpenFlow Table 1. Prefix 10.0.0.0/8 is announced by ISP1, ISP4, ISP5, ISP8. Since those providers belong the set ID1, the metadata value for the considered prefix is 11110.

Finally, the third step, namely filling the OpenFlow Table 2, is accomplished. This step involves the policies defined at the controller. Indeed, there is an entry for each policy defined. Consider the policy:
$p_1 = \langle dstport = 80 \rightarrow (\mathsf{ISP5}) \rangle$
and a traffic flow:
$f_1 = \langle srcip = 1.0.0.1, dstip = 10.0.0.1, srcport = 10, dstport = 80 \rangle$
then an entry is built in the following way: since the destination falls in the subnet 10.0.0.0/8 and this subnet is announced by providers ISP1, ISP4, ISP5, and ISP8, the metadata 11110 must be used. Since the policy p_1 must be taken into account and the metadata value does not give any information about which is the neighbor which the traffic must be forwarded to, a mask is needed. Since policy p_1 has ISP5 in the action atom, the mask 00100 must be used to forward the traffic. This process results in the creation of the first entry for the OpenFlow Table 2 in Fig. 6.4. After the last forwarding rule is installed because of a policy, this table contains the rules for forwarding the traffic simply according to the BGP best paths. We recall that this is the forwarding rule we apply whether the incoming traffic does not match any policies or whether the BGP routing does not allow the traffic to cross the neighbors specified in the action part of the policies.

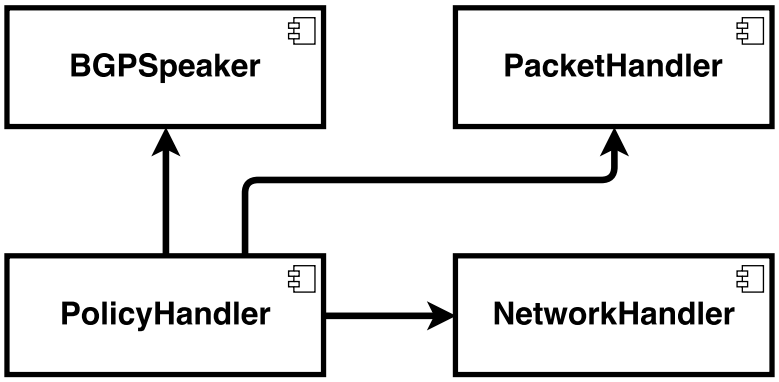


Figure 6.5: High level view of the internal architecture of our SDN-controller.

6.7 The Architecture of our SDN-controller

In this section we illustrate the internal architecture of our controller. We show the main components of our system and how they cooperate with each other in order to allow the traffic to be forwarded according to the policies defined by the user.

The internal architecture of our SDN-controller is shown in Fig. 6.5. It consists of four main components, called: 1) BGPSpeaker, 2) PacketHandler, 3) Network Handler, and 4) PolicyHandler. We now discuss each component in detail.

The BGPSpeaker component implements a BGP speaker being able to establish BGP peering with other speakers, to receive and to announce the BGP packets and to maintain a full BGP RIB. This component is crucial for two main reasons: it guarantees backward compatibility with the standard IP-speaking routers running the BGP protocols and it allows the policies to be expanded, according to the process widely described in Sec. 6.6.

The PacketHandler component offers basic functionalities for parsing and creating standard packets used by the controller to accomplish its tasks. For example, our SDN-controller relies on this component to handle the ARP traffic exchanged on the peering LAN of the IXP.

The NetworkHandler component allows the SDN-controller to interact with each SDN-enabled switch. This component implements most of the functionalities described in Sec. 6.8.

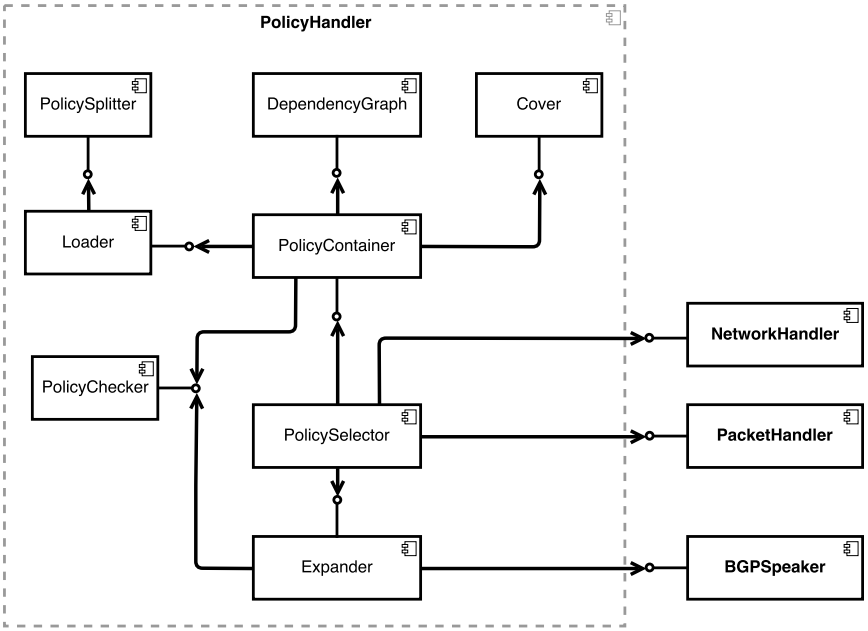


Figure 6.6: A detailed view of the internal architecture of the PolicyHandler.

Finally, the PolicyHandler component is the most important one. It implements all the algorithms described in this paper. It represents the core of our SDN-controller and carries out most of the functionalities. We now deeply illustrate this component and how it works.

The internal representation of the PolicyHandler is depicted in Fig. 6.6. It consists of a set of sub-components, each of which performs a specific task. The Loader component simply loads the policies (e.g. from a file) and builds the suitable data-structures representing them. It cooperates with the Splitter component in order to build policies only containing the *AND* operator, according to Sec. 6.6. After the policies have been loaded, they are stored in the PolicyContainer component, which exploits the DependencyGraph component to build the graph of policies dependencies described in Sec. 6.6. Also, by interacting with the Cover component, the PolicyContainer is able to raise a warning in case of a problem of covering among policies is happening. Finally, it interacts with the PolicyChecker to check whether policies are syntactically

correct. Now, the policies are available to be selected and then translated into forwarding rules according to the approaches described in Sec. 6.6.

The PolicySelector component selects a policy from the set of policies. In case of the *Reactive* approach, this component is triggered once a packet reaches the network devices; if the *Proactive* approach is running, then it is triggered in advance (e.g. during the start-up phase of the controller). The interaction with the network devices explains why it exploits the NetworkHandler component, whereas the interaction with the PacketHandler is justified by the need of interacting with the traffic. Finally, it also interacts with the Expander to carry out the expansion process described in Sec. 6.6. To perform such a step, the Expander needs to exploit the BGPSpeaker component, which provides a simple way to access the information contained in the BGP RIB. We recall that such an interface can be made available since that component implements a fully standard BGP speaker capable of establishing BGP peering.

6.8 Applicability Considerations

In this section, we discuss several aspects related to the applicability of DeSI. We focus on considerations about specific scenarios and backward compatibility with standard (or legacy) solutions (e.g., interconnection with IP-speaking nodes running the BGP protocol).

It is very common that a provider is interconnected to many IXPs. This choice is typically motivated by either resilience or performance reasons. In the first case, a provider typically implements the primary-backup strategies over the peering, whereas in the second scenario, load-balancing policies are applied. In every case, there might be the need of having multiple controllers. On one hand, more controllers represent a valid fault-tolerance strategy. On the other hand, performance increases whether each device is handled by its own controller, especially in the case of processing the full routing table.

Many techniques can be adopted to design solutions using multiple controllers. The first solution is built according to the master-slave architecture, consisting of a pair of controllers. A controller of that pair (called *master*) is actually managing the SDN-enabled devices and the second one (called *slave*) starts to act when the other fails. In case robustness is very crucial, more than two controllers can be used, resulting in a cluster. In this case, we assume that both master and slave controllers handle all the SDN-enabled devices. Sometimes, such an architecture is natively supported by the OpenFlow devices. Indeed, it is possible to set two (or even more) controllers during the device

configuration: one acting as master and the others acting as the slave. In this scenario, the OpenFlow device typically sends the same information to all the controllers, allowing them to have the internal knowledge of the network perfectly aligned. Relying on keep-alive messages, the switch is able to verify if the master controller is running or not. In case of failure, the device immediately changes the controller, giving to the slave the role of master, being ready to change once again as soon as the master becomes again reachable.

If this operational way is not supported by the device, multiple controllers can still be used. Nevertheless, the synchronization among the controllers is demanded to the controller themselves, that now have to exchange information about the SDN-enabled devices autonomously, without any kind of support from the device. This synchronization is carried out according to standard strategies used in distributed systems (e.g., cold or warm approaches), so that the slave can replace the master without any lack of information. Surely, other solutions can be implemented, provided that the internal state of the controllers is aligned when the master fails and the slave replaces it.

Another scenario involving multiple controllers is the following. A provider might choose to have a single controller for each IXP it is connected to. The main difference with the previous scenario is that in this case each controller handles a single device, or in general a subset of the whole devices the provider has in different IXPs, whereas in the master-slave approach each controller handles all the devices. Even in this approach, controllers must synchronize their internal states. As a solution, iBGP peering among the controllers can be set as in standard architectures. Also, route reflectors strategies can be applied for increasing the scalability.

Backward Compatibility

Another consideration in terms of applicability is referred to the backward compatibility. Indeed, our architecture is fully compliant with standard (or legacy) ones. There are no limitations in establishing BGP peering with other providers which use IP-speaking node. Our solution does not force other providers in the IXP to have an SDN-controller. Our SDN-controller is able to establish BGP peering with either other SDN-controllers or standard IP-speaking routers without requiring any specific configuration on both sides.

Route Server

As final consideration, we discuss about route servers. It is very common that IXPs offer the possibility to each participant to establish BGP peering with one or more router servers. A route server is a collector of BGP announcements, allowing providers to have multiple logical interconnections by setting up a single BGP peering instead of multiple ones. Even in this case, our SDN-controller is able to establish a peering with the route server, with no technological limitations. There could be just a limitation in terms of the possibility to choose among different paths. Indeed, a route server typically computes a single best path and then only that path is announced along each BGP peering. This operational way reduces the number of available alternatives that each provider has. Apart from that, there are no restrictions.

Even if other considerations might be done, we argue that what we discussed in this section is a significant sample addressing the most important aspects related to the adoption of our proposal in production environments.

6.9 Evaluation

We implemented a prototype version of DeSI based on the Ryu framework [ryu17] to validate the practical applicability of our approach. The reason to choose Ryu is that it provides an implementation of a standard BGP speaker. We used Kathará [BILD18] emulator to create a network and to run all the SDN components of the testbed. We focus our measurements on scalability aspects, taking into account the impact of our proposal in terms of resource consumption and required time to perform its tasks.

Our simulations consisted of two parts. First, we built a small IXP to run our implementation run in order to test the functionality of our controller. Second, we focused on resource consumption on the machine hosting the controller and the time spent by DeSI to carry out its activities. The tests were carried out varying both BGP announcements and the number of policies for both the *Reactive* and the *Proactive* approaches.

We first illustrate our testbed and several preliminary functionality tests. Then, we show and discuss the results of the resource consumption tests.

The testbed. We run our experiments in an Ubuntu virtual machine equipped with 2 GB of RAM and two Intel Core i5 with 2.8 GHz.

Fig. 6.7 shows the topology used to run our functionality experiments. The network contains a simple IXP consisting of four members (AS10, AS20, AS30, and AS40), each equipped with an SDN-enabled switch (OF1, OF2, OF3, and

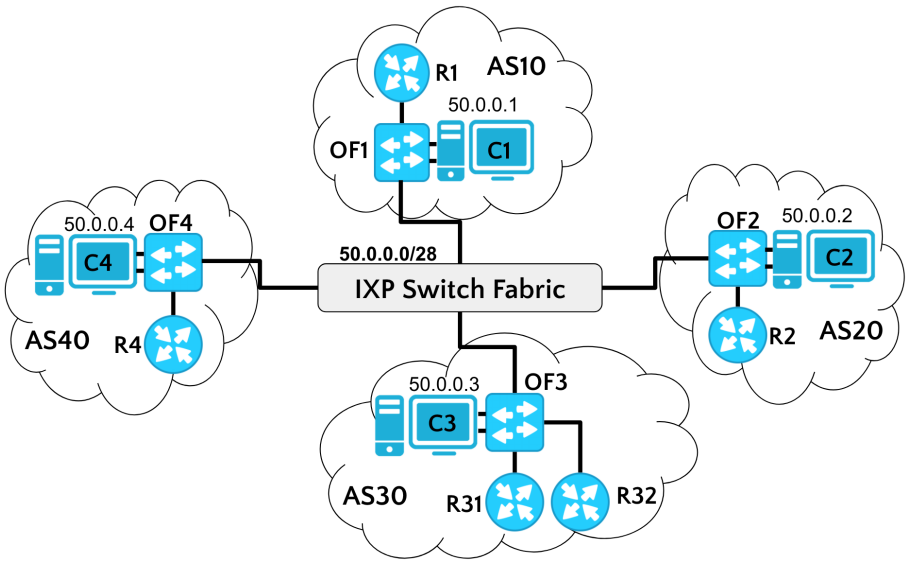


Figure 6.7: The topology used for the functionality tests. It simulates a simple IXP consisting of four members, each announcing just one prefix.

OF4) and with an SDN-controller (C1, C2, C3, and C4). Inside each provider's network we place a standard IP-speaking node (R1, R2, R31, R32, and R4) representing the whole network of the provider itself. In the case of member AS30, we use two nodes (R31 and R32) to reproduce the case in which a provider connects multiple border routers in the IXP. This is typically done for robustness or performance purposes, like primary/backup or load balancing strategies, respectively. The IXP switch fabric is a legacy layer 2 switch.

In the testbed, we assume that each controller is directly connected to its corresponding SDN-switch. This assumption is not restrictive. Indeed, since we do not introduce any constraints on the provider's backbone, we only need IP connectivity between SDN-controller and SDN-switch. Note that there are two connections between those components. These connections represent logical links: one link is used for the OpenFlow messages, whereas the second one is used for the BGP messages. We highlight that this interconnection is logical and not physical. Indeed, every technology that guarantees traffic isolation can be used (e.g. VLAN).

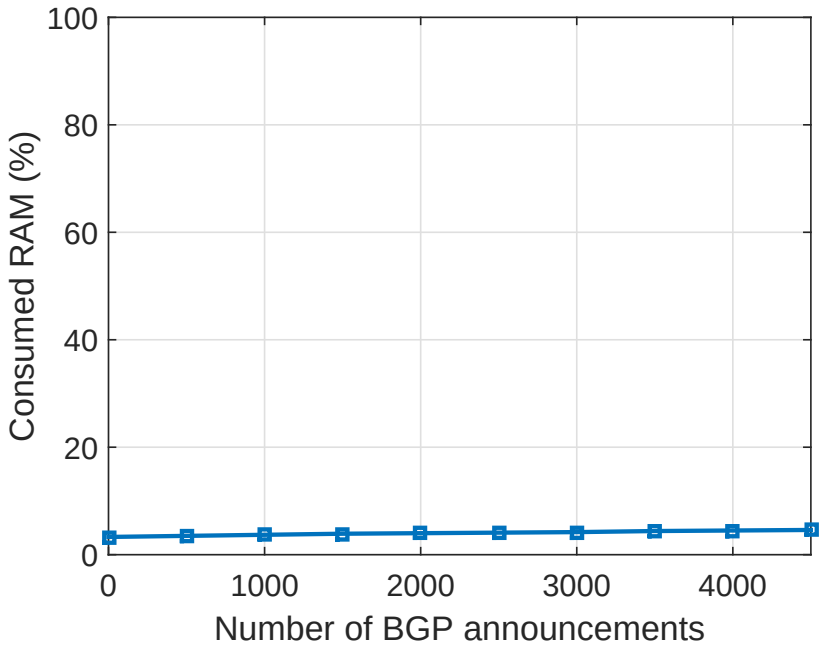


Figure 6.8: Percentage of consumed RAM in the proactive approach with a growing number of BGP announcements.

Each SDN-controller establishes a BGP peering with all the other controllers. Within those peerings, each provider announces a single prefix. Also, each SDN-controller gets its policies from a file. No restrictions are applied, namely, each controller does not filter anything, resulting in a full-mesh of peerings. For this experiment, we run both the reactive and the proactive approaches. We considered the following conditions: 1) We checked that each controller was able to successfully perform ARP requests over the peering LAN. This check is needed to allow the BGP messages to reach the right controller. 2) We checked that each BGP announcement was able to reach any other provider. We also checked that the announcements were successfully stored in the BGP RIB of each controller. 3) We checked that the traffic generated by each provider towards each known destination in the IXP was correctly forwarded according to the policies of each member. The above functionality experiments

were successfully carried out for each approach. Namely, we observed that our implementation works as expected.

We now show the results of resource consumption. For those experiments, we focus on a pair of SDN-controllers having a peering between them.

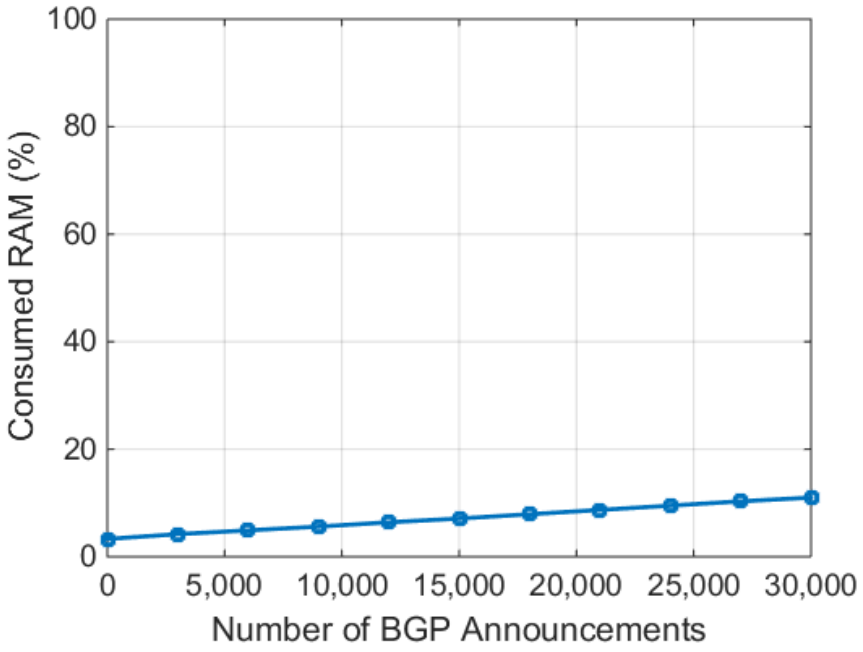


Figure 6.9: Percentage of consumed RAM in the reactive approach with a growing number of BGP announcements.

Resource consumption measurements varying the number of BGP announcements. The first test we perform refers to the amount of consumed resources by a controller which issues a growing number of BGP announcements. In this scenario, we measure the amount of consumed RAM. We perform experiments for reactive and proactive approaches.

Fig. 6.8 shows the consumed RAM by an SDN-controller issuing a growing number of BGP announcements. We perform this experiment with a number of BGP announcements in the range $[0, 4500]$. We observe that the percentage of RAM grows almost linearly with the number of BGP announcements. In

particular, during the experiments, our SDN controller consumes at most less than 5% of the available RAM.

Fig. 6.9 shows the results for the reactive approach. We increase the number of BGP announcements up to 30,000 and we still observe that the RAM grows linearly. Essentially, if no traffic reaches the SDN-enabled switch, both approaches have the same performance. For the test, no policies are given as input to the controller, so that the forwarding is accomplished according to the BGP control plane.

Resource consumption measurements varying the number of policies in presence of traffic. In the second experiment, we keep fixed the number of BGP announcements to 1,000 and we vary the number of the policies in the range [100, 1000]. For this experiment we use two controllers, C1 and C2. Controller C1 issues the BGP announcements, whereas C2 processes policies once it receives BGP messages from C1.

Fig. 6.10 depicts the amount of consumed RAM for the controller C2. Interestingly, C2 requires the same amount of RAM for both approaches. Even in this case, the amount of required RAM linearly grows with the number of policies. We ascribe this behavior to the fact that regardless the running approach, the controllers have all the needed information to issue the forwarding rules in its data-structure, so that it does not need additional resources (in terms of RAM) to perform the task of sending OpenFlow rules to the SDN-switch.

Resources consumption measurements varying the number of BGP announcements in presence of traffic. Finally, we perform an experiment similar to Test 2, in which we keep fixed the number of policies while we vary the number of BGP announcements. We still consider two controllers, C1 and C2, where C1 issues BGP announcements and C2 loads the policies and properly acts based on the implemented approach. We run this experiment giving 100 policies as input to C2.

Figs 6.11 and 6.12 show the amount of RAM consumed by C2. It is interesting to note that in the case of the reactive approach, the amount of consumed RAM is essentially constant, while in the case of proactive approach the amount of RAM grows linearly. We ascribe such a trend to the fact that in the proactive approach, C2 has to perform several operations in advance, regardless to the fact that the issued forwarding rules are used by some flows or not, while in the case of the reactive approach, C2 performs operations only if required. Those experiments are conducted by injecting in the network traffic that match several policies given as input to C2.

Time analysis. To assess the scalability of DeSI, we perform several experiments measuring the time spent by our system to: 1) handle BGP announ-

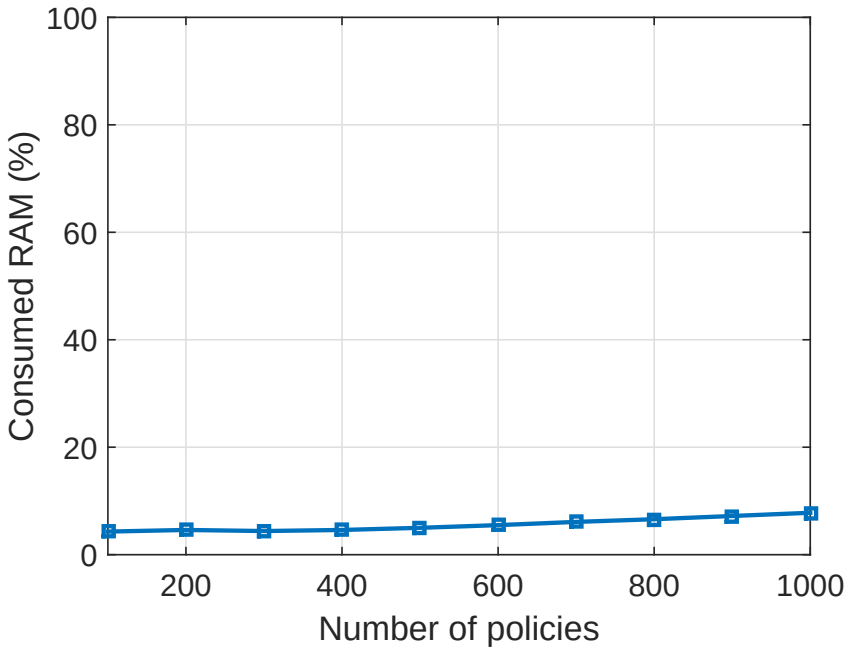


Figure 6.10: Percentage of consumed RAM at C2 in both approaches increasing the number of policies.

cements; 2) translate policies into OpenFlow rules; 3) install the OpenFlow rules. Fig. 6.13 shows the results of this experiments. In Fig. 6.13a, we show the time required to handle the BGP announcements received by a provider. We varied the number of BGP announcements in the range $[0, 20000]$ and the time spent to DeSI linearly grows with the number of BGP announcements.

Similarly, Fig. 6.13b and Fig. 6.13c show the time spent by DeSI to translate policies into OpenFlow rules and to install them in the OpenFlow switch, respectively. We varied the number of policies in the range $[0, 1000]$ and, even in this case, the time required to DeSI linearly grows with the number of policies. Note that, in the case of installing policies into OpenFlow rules, of course we are not considering the time induced by the network latency. By the way, that time can be considered negligible, since the management network typically used by providers induces a delay of few milliseconds (typically, less then 10).

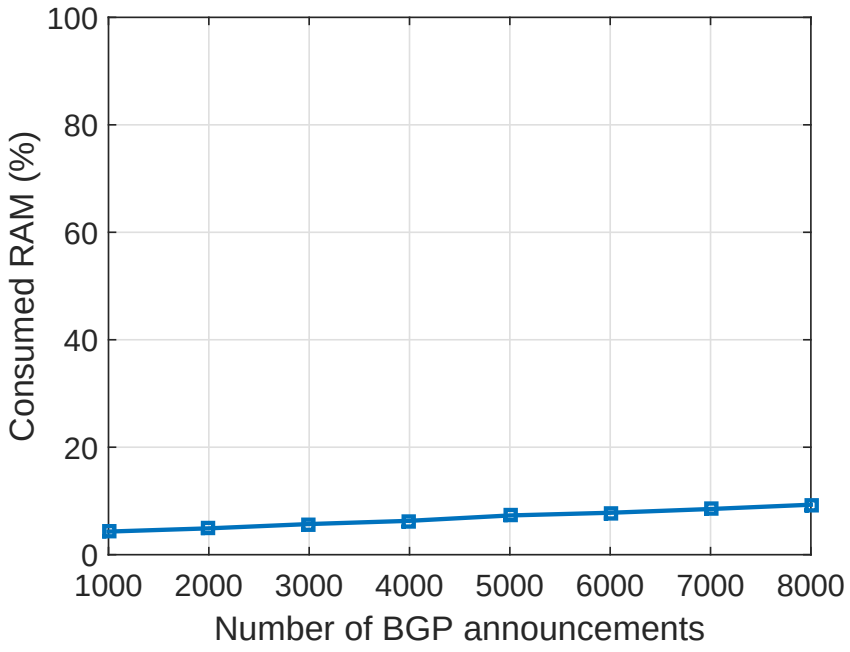


Figure 6.11: Percentage of consumed RAM in the proactive approach with a growing number of BGP announcements.

As a general consideration, these results show that our controller scales reasonably well.

6.10 Conclusion

We present DeSI, a decentralized SDN-based architecture for IXPs. DeSI guarantees the privacy of routing policies of ISPs , while opening to the possibility of overriding the standard traffic forwarding driven by BGP. Also, DeSI does not make the troubleshooting due to possible outages difficult, leaving the IXPs switch fabric as simple as possible. DeSI opens interesting research perspectives. Indeed, we are aware of possible deflection problems [BGFV17], namely possible forwarding loops induced by the choice of the non-best BGP paths. Ne-

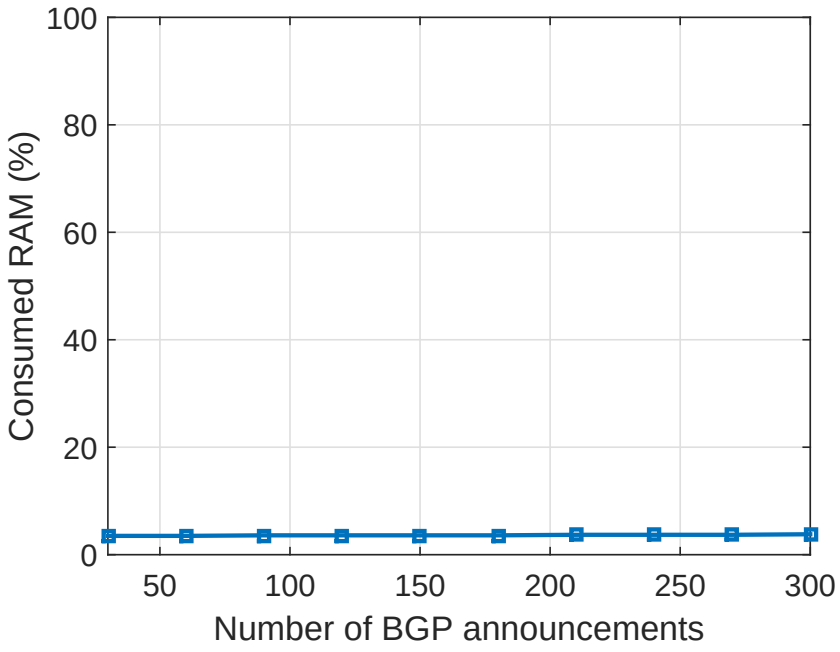
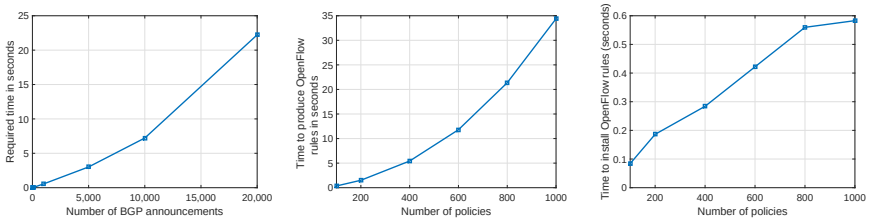


Figure 6.12: Percentage of consumed RAM in the reactive approach with a growing number of BGP announcements.

vertheless, we are also studying strategies and countermeasures, mainly based on the Gao-Rexford conditions [GR01], to avoid such a possible issue. Also, we are interested in exploring the possibility of extending DeSI with novel data-plane programmability paradigms, like P4 [BDG+14].



(a) Required time to handle BGP announcements.

(b) Required time to translate policies into OpenFlow rules.

(c) Required time to install OpenFlow rules generated by a set of policies.

Figure 6.13: Time analysis about the time spent by DeSI to perform its activity.

Chapter 7

Activity-based Congestion Management (ABC) In Programmable Networks*

Activity-based congestion management (ABC) is a stateless-core method for bandwidth sharing among users in packet-switched communication networks. ABC features a domain concept where ingress nodes record in packet headers the activity information which depends on the traffic volume recently sent by the user. Forwarding nodes rely only on this activity to take drop decisions in case of congestion, in particular, they do not require per-user or per-flow information. In this chapter, we report about an ABC prototype in P4 and show selected performance results that demonstrate the ability of this prototype to protect light users against heavy users. ABC provides an environment where users can maximize their throughput by sending at their fair share and which incentivizes the use of congestion control.

7.1 Introduction

Future mobile networks like fifth generation (5G) will consist of small cells that possibly issue large traffic rates with high fluctuations [JITT16]. As quality of service (QoS) is required, economic provisioning of the transport network is

*Part of contexts in this chapter is based on the following publication: Mostafaei, H., Merling, D., Menth M., Experience from a P4-Based Prototype for Activity-Based Congestion Management (ABC), under review in IEEE Communications Magazine.

a challenge. Datacenter networks and residential access networks of Internet service providers (ISPs) have uses cases with similar requirements.

To avoid QoS degradation, scalable bandwidth-sharing mechanisms for congestion management may be helpful, but they need to be simple and effective. That means, light users should be protected against overload caused by heavy users while avoiding per-user signaling and information within the transport network.

We discuss activity-based congestion management (ABC) for that purpose. It implements a domain concept where edge nodes run an activity meter that measures the traffic rates of users and add activity information to their packets. Forwarding nodes leverage activity-based active queue management (activity AQM) which leverages this information to preferentially drop packets from most active users in case of congestion. In [MZ18], ABC has been proposed in its current form and extensive simulation results have demonstrated that ABC can effectively protect light users from heavy users to such an extent that single TCP connection from a light user does not suffer in the presence of congestion caused by a large constant bitrate (CBR) traffic stream of a heavy user.

As ABC requires additional header information and new features in edge nodes and forwarding nodes, it cannot be configured on conventional networking gears. However, advances in network programmability support the definition of new headers and node behavior. The network programming language P4 is a notable example [BDG+14].

In this work, we report about P4-based prototype for ABC. It demonstrates the technical feasibility of ABC while pointing out difficulties that may occur when porting the implementation to other platforms. Furthermore, we present experimental results which confirm the simulative findings of [MZ18].

The reminder of the chapter is structured as follows. Section 7.2 briefly reviews related work. Section 7.3 explains the ABC concept in detail. Section 7.4 gives an introduction to SDN and P4. Section 7.5 describes the ABC implementation in P4. Section 7.6 presents our evaluation methodology. Section 7.7 reports experimental results. Finally, Section7.8 concludes this work.

7.2 Related Work

A comprehensive overview of congestion management techniques can be found in [Bro13]. Here, we discuss some methods which are similar to ABC.

Core-stateless fair queueing (CSFQ) [SSZ98] also features a domain concept. Edge nodes label meter the traffic rate of flows or users and record

them in packet headers. Forwarding nodes leverage this information together with online rate measurement to detect congestion and to determine suitable drop probabilities for packets. With ABC, forwarding nodes rather use queue lengths instead of rate measurement to detect congestion and activity-depending queue thresholds to drop packets. The performance of CSFQ and ABC has been compared in [MZ18] using simulation.

Fair dynamic priority assignment (FDPA) [CBB+17] is a fair bandwidth sharing method for TCP traffic. It assigns to the TCP traffic of light users a higher priority than to the TCP traffic of heavy users. FDPA is implemented in OpenFlow and P4. It assigns different priority to users' traffic based on their traffic rate. While FDPA is applicable only for responsive traffic, ABC works with responsive traffic, non-responsive traffic, and combinations thereof.

An approximate per-flow fair-queueing approach (AFQ) for reconfigurable switches has been proposed in [SLAK18]. It leverages the features provided by data plane programmability like per-packet state to dynamically determine a suitable egress port to a packet. AFQ leverage multiple queues per port and rotates their priority. However, the functionality of AFQ is not supported by all target switches. In contrast, ABC exploits the length of available queues to enforce bandwidth sharing.

7.3 Activity-Based Congestion Management (ABC)

We review the concept of ABC. We first describe the domain structure and its components. Then, we introduce the activity meter and the activity AQM in more detail and discuss properties of ABC.

ABC Overview

ABC features a domain concept which is shown in Figure 7.1. Ingress nodes leverage activity meters to measure the rate of traffic aggregates that enter the ABC domain. They derive an activity value for each packet and record that value in its header. Such an aggregate may be, e.g., the traffic of a single user or a user group. Thus, ingress nodes require traffic descriptors for any aggregate that should be tracked.

Forwarding nodes of an ABC domain, i.e., ingress nodes and core nodes, implement an activity AQM for each outgoing interface. They calculate a moving average A_{avg} of the activity of all forwarded packets. They leverage this value A_{avg} and the activity A of a received packet to determine the drop

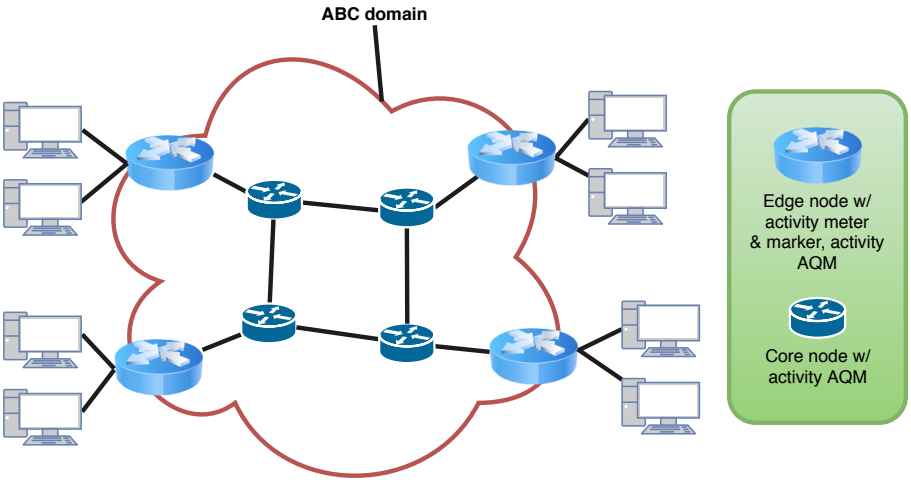


Figure 7.1: Activity metering and tagging is performed only by ingress nodes. Both ingress and core nodes apply activity AQM during packet forwarding. Figure taken from [MZ18].

threshold T_{drop}. If the current queue length exceeds that threshold, the packet is dropped. This enforces fair resource sharing among traffic aggregates within an ABC domain.

Egress nodes of an ABC just remove the activity information from packets leaving the ABC domain.

Activity Meter

Ingress nodes run an activity meter per monitored traffic aggregate. The activity meter measures a time-dependent traffic rate R_m. In [MH17], several algorithms have been documented and compared for that purpose. They all can be configured with a memory which essentially relates to the time over which the rate is computed. We denote the memory for the activity meter by M_{AM} which is a configurable value. An activity meter is additionally configured with a reference rate R_r and computes the activity of a packet by $A = \log_2(\frac{R_m}{R_r})$. The activity is written into the header of a packet before passing it to the ABC domain.

Activity AQM

Forwarding nodes run an activity AQM per outgoing interface. The activity AQM runs an activity averager that computes a moving average A_{avg} over the activity of all successfully forwarded packets. Thus, A_{avg} reflects mainly the activity values of recently forwarded packets. Algorithms for moving averages can also be found in [MH17]. They are configured with a memory, too, to control how fast old samples lose impact on the computed average value. We denote the memory of the activity averager by M_{AA}.

When a forwarding node receives a packet, the AQM calculates the drop threshold by $T_{drop}(A) = \max(Q_{min}, Q_{base} - \gamma(A - A_{avg}))$. If the current queue length Q of the egress port exceeds that threshold, the packet is dropped. Q_{base} is a baseline value around which packet dropping should start. The drop threshold T_{drop} is lower for packets with an activity larger than A_{avg} and it is larger for packets with an activity lower than A_{avg}. Therefore, packets with larger activity are preferentially dropped in the presence of congestion. The parameter γ controls that differentiation. The configurable value Q_{min} prevents packet dropping in the absence of congestion.

ABC Properties

Ingress nodes require state information per aggregate, i.e., a configurable traffic descriptor to map the traffic to its activity meter, a counter and a configurable memory value M_{AM} for rate metering, and a configurable reference rate R_r. The memory value M_{AM} should be the same for all aggregates within the ABC domain while the reference rate R_r may be used for service differentiation. That means, aggregates with larger reference rates can obtain a larger capacity share than aggregates with lower reference rates. This claim is backed by simulation results in [MZ18].

Forwarding nodes require state information only per egress port, i.e., a counter for the activity averager and the configurable parameters γ, Q_{min}, Q_{base}, and M_{AA}, whereby the latter should be the same for all egress ports within an ABC domain. Appropriate values depend on forwarding capacity and queue size. In [MZ18] their impact on resource sharing has been studied and reasonable values have been proposed for a link speed of 10 Mb/s and a queue size of 24 packets. Thus, only edge nodes of an ABC domain require per-aggregate state while core nodes require only state information per egress port. This makes ABC a scalable congestion management system for closed networking domains. As potential packet dropping depends on the activity

value contained in packet headers, activity meters and forwarding nodes should be trusted devices. Otherwise, malicious users may avoid ABC control by inserting low activity values and obtain unfairly large traffic rates.

In [MZ18], ABC has been extended to support service differentiation with different per-hop behaviors (PHBs). These extensions are simple but we omit them here for the sake of brevity.

7.4 SDN and Data Plane Programmability Using P4

We briefly explain the concept of software-defined networking (SDN) and introduce data plane programmability with the programming language P4 and its processing pipeline.

Software-Defined Networking

Conventional networking relies on distributed protocols that are run by switching devices to determine forwarding actions. E.g., self-learning bridges fill the forwarding tables in Ethernet networks and routing protocols compute forwarding tables for IP networks. In contrast, SDN leverages a network controller to populate forwarding tables of switching devices through a protocol which is mostly denoted as southbound interface. OpenFlow is a well-known example and appropriate switches implement its application programming interface (API). This concept of SDN is said to decouple data plane from control plane. Forwarding rules may be coarse- or fine-granular and they may be configured statically or on demand. Therefore, SDN increases network flexibility and simplifies network management [KRV+15b].

Data Plane Programmability Using P4

The controller-based approach in SDN makes the control plane more flexible. However, SDN approaches like OpenFlow can work only with existing headers and forwarding behavior that are configurable through the API. To enable new forwarding behavior on networking devices and to support new headers, networking devices require new forwarding logic, i.e., their data plane needs to be programmed instead of configured. P4 [Th] is a programming language for that purpose that may be applied on P4-capable switches. Reprogrammed switches offer an API for configuration purposes. It may be used to manually configure match action tables or it may be leveraged by a controller to configure forwarding devices automatically and on demand. Thereby, this form of data

plane programmability also implements the SDN concept. The definition of that controller is not part of P4.

A P4 program defines a pipeline for packet processing. It is installed and configured on the switches through the API. The pipeline of a P4 switch is visualized in Figure 7.2.

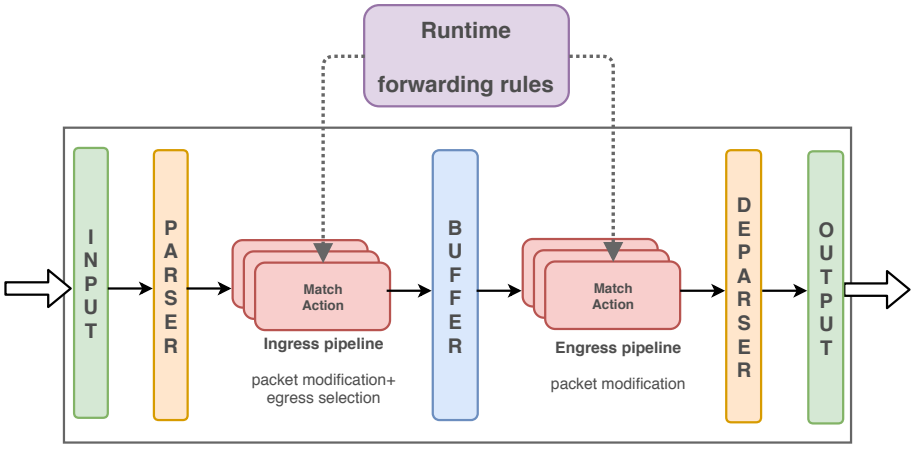


Figure 7.2: P4 processing pipeline.

When a packet is processed by the pipeline, first its is parsed. Throughout the whole pipeline, the parsed header fields are carried together with so-called standard metadata, e.g., the port on which the packet has been received. In addition, it is possible to define metadata fields (user-defined metadata) to store calculated values during processing, e.g. to implement flags for decisions later in the pipepline. The parser is followed by a programmable sequence of match action tables. The entries of a table are called rules and each consists of a match part, and a set of actions. Rules are installed, modified or deleted during runtime by a controller. When a table lookup is performed, a header or metadata field is matched onto a table and the field is compared with the match part of each rule until a matching rule is found. There are different kind of match types, e.g., longest prefix match, wildcard or exact match. If an entry matches, the corresponding set of actions, e.g., change header fields and/or update metadata counters for statistics, is executed and no further matching on this particular table is executed. To implement IP forwarding, a match action table would match on the destination IP address of the packet and the

corresponding action would be sending the packet to the appropriate egress port. It is possible to define a sequence of match action tables and to match on different tables depending on the match of a previous one. However, to prevent processing loops, it is not possible to match onto a table a second time. The first sequence of match action tables is called ingress pipeline. Normally it is used to adapt header fields and to determine the egress port of a packet. After the egress pipeline has concluded, the header fields and/or metadata are matched on a second sequence of match action tables: the egress pipeline. After processing, the packet is deparsed and sent if a port has been specified.

With P4 new headers can be defined and their fields can be used for matching rules. We leverage this feature for our ABC implementation since ingress nodes of the ABC domain record the activity of the corresponding aggregate in a new packet header.

The action sets of a match action table consist of pre-defined primitive actions like adding or removing a header, reading and writing header or metadata fields, adding or subtracting values, or dropping a packet. It is possible to call custom functions within an action set. Those functions are called externs and the set of available externs depends on the P4-capable switch. On some switches it is even possible to define new externs. Depending on available externs, it may be possible to perform floating point calculations or to encrypt and decrypt fields.

7.5 P4-Based Implementation of ABC

We now describe the implementation of ABC in P4. We first introduce the leveraged P4 features. Afterwards we explain the P4 packet processing pipeline for ABC.

Supported ABC Feature Set

An ingress node of an ABC domain activity-meters the traffic and records the activity in the packet header. A forwarding node performs activity AQM, i.e., it calculates the average activity A_{avg} of recently forwarded ABC packets and possibly drops packets depending on their activity and A_{avg} in case of congestion. An egress node removes the ABC header before the packet leaves the ABC domain. Our data plane implementation features ingress node, forwarding node, and egress node behavior. An ABC controller configures traffic descriptors, reference rates R_r, and the same memory M_{AM} for all aggregates

on ingress nodes, as well as Q_{base}, γ, Q_{min}, and the same memory M_{AA} on all forwarding nodes.

We only provide a data plane implementation for ABC. A controller implementation is straightforward as it statically configures ingress and core nodes. For testing purposes we utilize a python script that populates the match action tables with appropriate entries. For the sake of readability we omit some technical details about P4 programming, the P4 code and P4 syntax. For details we refer to the $P4_{14}$ specification [P4S17] and the $P4_{16}$ specification [Th].

Metadata

We leverage standard metadata in our P4 program to obtain packet information for activity metering and activity AQM. Since activity metering relies on time-dependent rate measurement, the packet arrival time is required. Additionally, the queue length of the egress port is needed to determine whether a packet should be dropped. Both values are available through standard metadata. To store calculated values throughout the pipeline, we rely on several user-defined metadata fields.

Header

ABC ingress nodes calculate aggregate activities and record them in packet headers. We define a header for that purpose which consists of a 32 bit field. ABC ingress nodes push this header onto packets entering the ABC domain. In P4 it is possible to access all headers of a packet and not just the top-most header. Thus, this very simple header is sufficient for a simple prototype as other header information is also available for forwarding decisions. Activity values are floating point numbers which are not supported by P4. Therefore, we multiply activity values with 10^6 and store these values as integers within the 32 bit field. This causes only negligible inaccuracy.

Externs

Activity metering and activity AQM require logarithmic and exponential operations. Since both are not supported natively by P4, we leverage extern functions to implement the necessary functionality. To that end, we added the desired C++ code for the necessary extern functions to the source code of the target switch which is the software switch BMv2 version 1.10.3. For more up to date versions with potentially changed structure and capabilities,

we recommend to copy the code of the extern functions and to manually insert them at the appropriate places in the source code. After recompilation of the software switch our user-defined extern functions were available for our P4 program. The use of such externs makes P4 programs target-specific. In particular hardware switches offer only very restricted possibilities to add custom extern functions.

In our P4 implementation we leverage extern functions to measure time-dependent traffic rates, calculate the packet and average activity, and take packet drop decisions. Those externs are required for the ABC algorithm itself and are called within the ingress control flow.

This requires access to the queue length of the egress port. As this information is not available in the ingress control flow but in the egress control flow, we developed the following workaround. The queue length of the egress port is read from standard metadata and written to a variable by another extern function which is called by the egress control flow whenever a packet is sent. This variable is accessed by another extern function in the ingress control flow to obtain the current queue length of the egress port. This workaround may cause slight inaccuracies as the ingress control flow utilizes not the current queue length but the queue length from the instant when the last packet was sent by the egress port.

Ingress Control Flow

To implement the behavior of an ABC edge node, the first match action table matches on the source IP address of the packet. This is necessary to calculate the user-specific activity by an extern function by the aid of the previously introduced metadata. The resulting value is then written into the packet header. This lookup implements activity metering in ABC edge nodes. Afterwards, the destination IP address is matched onto another table to determine the egress port. Additionally, activity AQM needs to be applied to the packet. The necessary operations namely, calculating the average acitivty, reading the egress queue size and making the drop decision are performed by additional extern functions at this point. At the end, the extern function that makes the drop decision sets an user-defined metadata field that acts as a drop flag. In the P4 processing pipeline the packet is passed to the egress control flow or dropped depending on the drop flag.

For our proof of concept the ABC control flow is only applied to traffic that is sent from the clients to the server. Traffic that goes from server to clients

does not pass the ABC domain and is forwarded simply by a match on the destination IP address.

Egress Control Flow

The egress control flow handles accepted packets. Since the egress port has already been determined in the ingress control flow, the egress control flow only needs to call an extern function, that reads the current egress queue size and writes it into the designated variable. Afterwards, the packet is sent.

7.6 Evaluation Methodology

In this section we describe the experimental setup of our evaluation. First, we explain the setup of our mininet environment leveraging a P4 software switch, followed by a description of our experiment design.

Network Design

We emulate a network leveraging mininet version 2.2.2. Two users are connected to a BMv2 P4 software switch via links with 100Mb/s capacity each. The switch itself is connected to a server via a 10Mb/s link and the buffer size for this connection is 24 packets. All links are configured in the same way with a delay of $5ms$.

Experiment Design

Scenarios

In each scenario two users try to reach a server. The access links of the two users have a higher capacity than the link that connects the switch and the server. Thus, it is possible for a user to turn the link between switch and server into a bottleneck by sending with a high traffic rate. We have two users. One user acts as the heavy user that tries to gain as much bandwidth as possible by relying on an aggressive sending behavior. The second user is a passive, light user. The light user always sends with a constant traffic rate or a constant number of TCP flows. In each run we configure the heavy user with an increase traffic rate or number of TCP flows. During the run, we measure the throughput of each user on the server. We do this with and without ABC and compare the throughput. We focus on three scenarios with different combinations of

non-responsive (UDP) and responsive (TCP) traffic. In the TCP scenario the light user establishes one TCP flow and we vary the number of TCP flows of the heavy user. In the UDP scenario we vary the traffic rate of heavy user while the light user sends with a constant traffic rate. In the last scenario we mix TCP and UDP traffic. We increase the UDP traffic rate for the heavy user and the light user establishes one TCP connection. For each link delay ($5ms$ or $50ms$) we measure the throughput for 300 seconds after running the experiments with 60 seconds. Each such run is done 10 times to average the resulting throughput.

Technical Setup

All experiments ran on a virtual machine with Ubuntu 14.04, 4 CPU cores with 3.5GHz and 8 GB of RAM. We leveraged version 3.1.7 of Iperf3 for TCP and UDP traffic generation.

ABC Parameter

In Section 7.3 we explained ABC. To keep the explanations simple we focused only on the concept and did not introduce the different parameters. They are explained in detail in [MZ18]. For the sake of reproducibility of our results and to make a comparison with [MZ18] we included our parameter set in this section. We take the values for these parameters from [MZ18]. $Q_{min} = 12$ packets, $Q_{base} = 20$ packets, $\gamma = 16$, $M_{AM} = 3s$, $M_{AA} = 0.3s$, $R_r = 10\frac{kb}{s}$.

7.7 Performance Evaluation

In the following we report experimental results with the P4-based ABC implementation. The detailed simulation results of ABC can be found in [MZ18]. We first describe our evaluation methodology and then we discuss performance results from three different experiment series.

Evaluation Methodology

The clients send CBR or TCP traffic via a switch to a server using Iperf3 version 3.1.7. We measure the throughput for 300 seconds. We repeat such experiments 10 times, average the obtained throughput for each user.

We run all experiments on a virtual machine with Ubuntu 14.04, 4 CPU cores with 3.5GHz, and 8 GB of RAM. To simplify comparison between ex-

perimental and simulation results, we run experiments with the same ABC parameter settings as in the simulation study in [MZ18]. These configuration parameters are: $M_{AM} = 3$ s, $R_r = 10$ Kb/s, $M_{AA} = 0.3$ s, $Q_{base} = 20$ packets, $\gamma = 16$, and $Q_{min} = 12$ packets.

Resource Sharing with CBR Traffic

In a first experiment series both clients send CBR traffic. Client 0 transmits at different rates $R_0 \in \{2.5, 5, 6, 7, 7.5, 8, 9, 10, 12.5, 15\}$ Mb/s while Client 1 sends at at $R_1 = 7.5$ Mb/s. Figure 7.3 shows the obtained throughput for both clients.

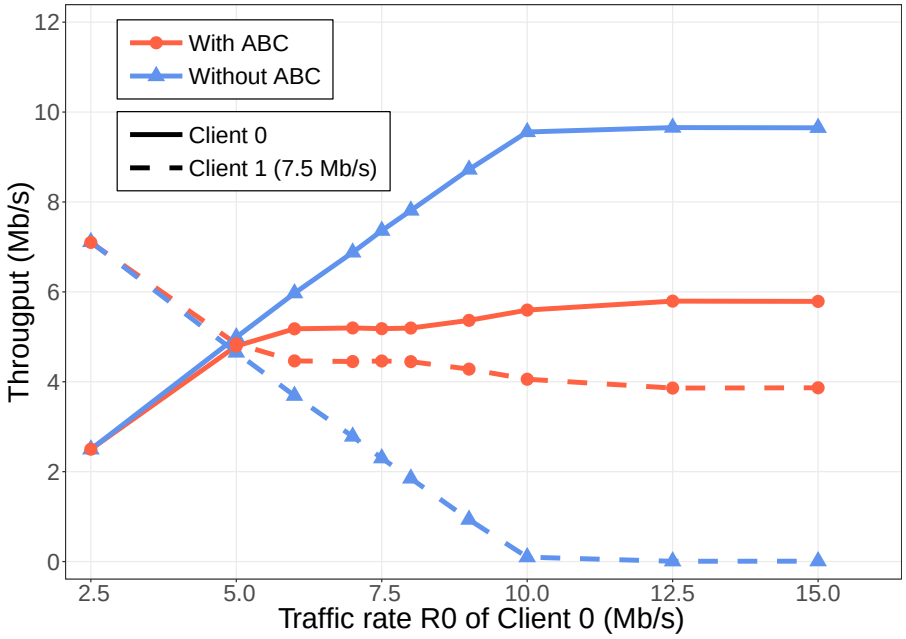


Figure 7.3: Resource sharing of Client 0 and Client 1, both sending CBR traffic.

Without ABC, the throughput of Client 0 continuously increases with increasing traffic rate R_0 while the throughput of Client 1 decreases. In particular

the throughput is proportional to the sent traffic rate of both clients. Thus, unfair sending behaviour is rewarded with higher throughput. This is different with ABC. The throughput of Client 0 continuously increases with increasing traffic rate R_0 while the throughput of Client 1 continuously decreases only as long as its traffic rate R_0 is lower than the traffic rate R_1 of Client 1. For larger traffic rates R_0, Client 0 obtains only a very small throughput while almost all traffic of Client 1 reaches the server. Reason for that phenomenon is that traffic from more active users is preferentially dropped in case of congestion. In this experiment, resources are not shared fairly but the client with lower throughput is able to increase its packet loss probability by reducing his transmission rate, which improves its throughput as long as the transmission rate is larger than the fair share. Thus, ABC creates an ecosystem that rewards clients sending at their fair share and which incentivizes the use of congestion controlled transport protocols. Note that ABC creates this ecosystem based on the parameter values that are provided apriori. The fairness comes from the amount of sent traffic of a client and its contracted traffic rate.

Resource Sharing with TCP Traffic

In a second experiment series both clients send TCP traffic. We vary the number of TCP saturated connections of Client 0 while Client 1 has only a single saturated TCP connection. Thus, Client 0 is a heavy user while Client 1 is a light user. Figure 7.4 shows the obtained throughput for both clients.

Without ABC, the throughput of Client 0 clearly increases with increasing number of TCP connections while the throughput of Client 1 clearly decreases. With ABC, the throughput of both clients remains around 5 Mb/s if the number of TCP connections for Client 0 increases. Thus, fair resource sharing is well approximated.

Resource Sharing with CBR and TCP Traffic

In the third experiment series Client 0 sends CBR traffic at different rates $R_0 \in \{2.5, 5, 6, 7, 7.5, 8, 9, 10, 12.5, 15\}$ Mb/s while Client 1 has a single saturated TCP connection. Figure 7.5 shows the obtained throughput for both clients.

Without ABC, the throughput of Client 0 increases with increasing transmission rate while the throughput of Client 1 decreases to such an extent that it can only use the bandwidth left over by Client 0. If the transmission rate of Client 0 exceeds the capacity of the bottleneck link, Client 1 achieves hardly any throughput. This is different with ABC. Client 0 can increase its throug-

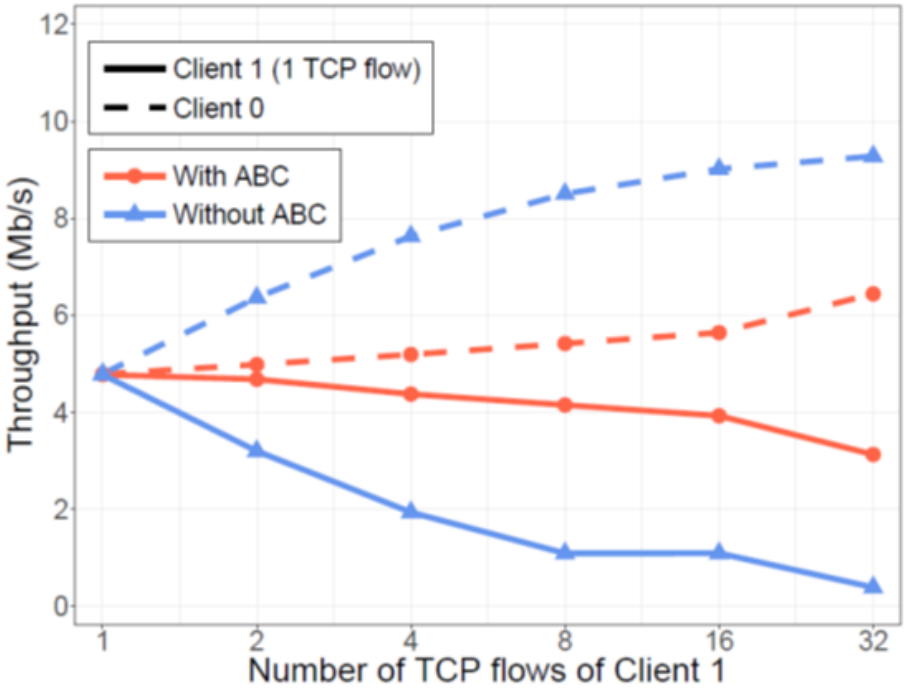


Figure 7.4: Resource sharing of Client 0 and Client 1, both sending TCP traffic.

hput by increasing its transmission rate only up to a value that is only slightly larger than its fair share of 5 Mb/s and the remaining capacity is utilized by Client 1. Thus, ABC supports fair resource sharing even under challenging conditions.

7.8 Conclusion

In this chapter we reviewed activity-based congestion management (ABC) for fair resource sharing and reported a prototype implementation in P4 and experimental results. Our prototype proves basic technical feasibility of ABC on programmable switches. However, the prototype leveraged several extern

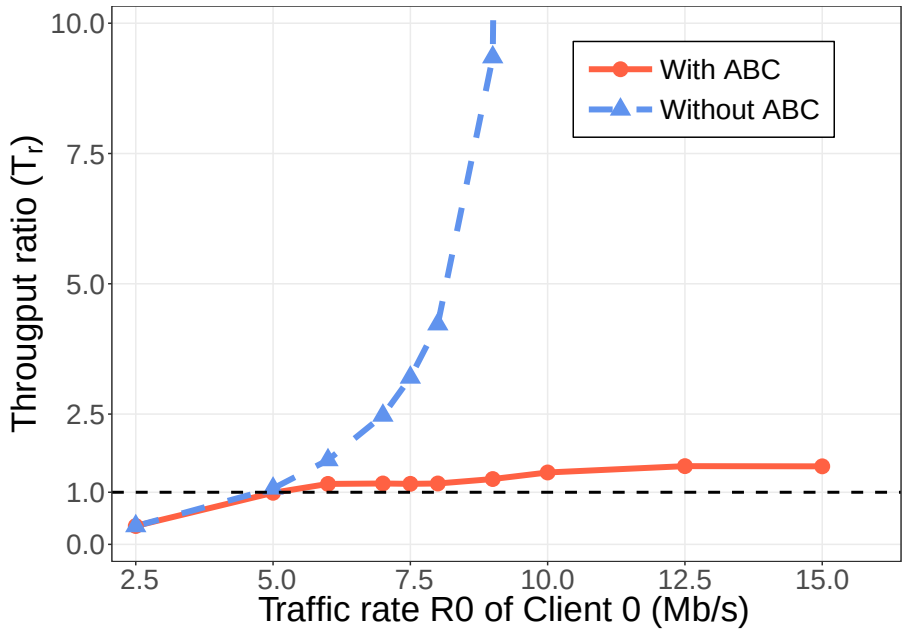


Figure 7.5: Resource sharing of Client 0 and Client 1, Client 0 sends CBR traffic while Client 1 has 1 TCP connection.

functions that can be implemented on a software switch but may currently not be available on hardware switches. Thus, P4-capable hardware switches should provide a wider range of externs to support richer use cases. Experimental results with the prototype implementation confirmed the resource sharing results previously obtained with simulations [MZ18]. We reported only results for a bottleneck link delays of 5 ms, but experimental results for bottleneck link delays of 50 ms are similar. Thus, ABC works not only in a simulation environment but also in an emulation environment with real networking stacks. After all, ABC provides an ecosystem where users can maximize their throughput by sending at their fair share in case of congestion, which incentivizes the use of congestion controlled transport protocols. As ABC does not require per-user states in core nodes, it is a scalable mechanism for fair bandwidth sharing which may be attractive for 5G transport networks, data center networks, or

residential access networks of ISPs.

Chapter 8

Conclusions

Issuing value-added network services in the inter-domain networks leads to several challenging problems in terms of privacy, provisioning, and scalability due to the co-existence of several technologies and protocols. These challenges can be mitigated by employing Software-Defined Networking (SDN). The SDN allows the network administrators to easily define network services on the SDN-based networks. However, relying on a centralized controller to manage the whole network suffers from many problems like scalability and privacy. In this work, we improve privacy, provisioning, and scalability in inter-domain networks with and without leveraging SDN.

First in chapter 3, we studied the privacy of routing policies at IXPs. An IXP can leverage router server (RS) to offer multi-lateral peering to IXP's participants. Current RS at IXPs failed to provide guarantee on the privacy of agreements among the members. We proposed an RS implementation for IXPs not only guarantees the privacy of routing policies but also ensures the security of RS software. We validate our system through a simulated environment and the results showed that the system adds a negligible amount of overhead to ensure the privacy of routing policies. The members of IXP can attest the RS-software. We leveraged the open source implementation of Intel SGX to secure our RS-software and to attest the running software. The performed test confirmed that the implementation can detect any malicious behavior at RS machine.

In chapter 4, we introduce SDNetkit as an emulator to perform experiments on SDN-based inter-domain networks. We categorized the current state-of-the-art works on available simulators and emulators based on their supported

OpenFlow versions. We point out several interesting use-cases that can be performed by SDNetkit which are not supported by current tools. Running several protocols on each virtual machine of SDNetkit is one of the main features which give the opportunities to run the required routing daemons, e.g., Qugga, for inter-domain routing. SDNetkit requires the configuration files for each VM as they are demanded to run the experiments in a real physical system.

In chapter 5, we showed that how a complex and time-consuming efforts to setup and provision a network service spanning over several domains can be simplified at least from configuration point of view by using SDN. We proposed an architecture for federated networks to issue value-added services by leveraging the advantages of SDN and the available backbone network of customers. Best practice like Geant suggests to use federated PoP to issue such a service which increases the cost because of renting a physical place, buying racks, electricity costs, etc. We devised a configuration language with simple yet effective primitives for the customers of federated networks to easily join and leave a federated VPN service. The proposed system exploits Domain Name System (DNS) to identify the customers aiming at sharing their resources. The control plane evaluation results showed that the proposed system does not introduce any overhead message to establish a federated VPN service. The system uses an SDN controller for each customer of a federated network which aims at scalability of the system.

In chapter 6, we proposed a decentralized architecture to improve the scalability and privacy of SDX-based IXPs, i.e., SDXes. The members of an IXP can bring their SDN-controllers along with OpenFlow-enabled switches to establish peering with other members. The proposed architecture is IXP fabric agnostic meaning that the physical infrastructure remains unchanged. We proposed a new policy language for the system which allows the members to describe their customized routing policies regarding their goals like traffic engineering. The language ables to detect dependencies among the routing policies which is never considered by current state-of-the-art works. We evaluated the performance of our system regarding the physical resource consumption and the results showed that the SDN-controller of each member does not occupy much resources from each virtual machine. In our system, each member stores its own routing policies and agreements with other IXP's participants at its SDN-controller aiming at not revealing information to any third-party. Like other SDX solution, our system allows overriding of BGP paths meaning that the controller can select non-BGP best path to steer the traffic of members aiming at load-balancing.

In chapter 7, we implemented activity-based congestion management (ABC) in P4 network programming language which is able to process packets at line

rate. With ABS edge nodes perform activity metering and tag packets with activity value while forwarding nodes leverage this information to drop the packets of heavy users in the case of congestion. ABC makes an ecosystem so that users can maximize their throughput by sending fair traffic rate. However, ABC drops packets of heavy users in the case of congestion because they have higher activity value. We showed that ABC fairly works with responsive, non-responsive, and combinations thereof.

We believe that there are several open problems that can be considered as research directions. Leveraging SGX for RS-software comes with overhead. There are solutions to decrease the SGX overhead [ATG+16, TSB18]. SGX is also prone to attacks like side channel attack [OTK+18]. This requires further investigation in RS-software implementation. The next open question is how a federated VPN service can be issued leveraging data plane techniques. Interacting with the SDN-controller to issue a federated VPN service adds to the system which can be removed by programming the network directly in data plane instead of control plane. The proposals for Software-Defined eXchange (SDX) assumed that there is no dependency among the routing policies of the members. We proposed a mechanism to find dependency among routing policies. However, more efficient algorithms can be designed to find the dependency among the policies even more to generate the SDN-based forwarding rules from the policies because the capability of SDN-enabled switches to place the rules are limited.

List Of Publications

Published papers

1. Habib Mostafaei, Michael Menth, Mohammad S Obaidat, et al. A learning automaton-based controller placement algorithm for software-defined networks. pages 1–6, 2018

2. Habib Mostafaei, Gabriele Lospoto, Roberto di Lallo, Massimo Rimondini, and Giuseppe Di Battista. Sdnetkit: A testbed for experimenting sdn in multi-domain networks. In *Network Softwarization (NetSoft), 2017 IEEE Conference on*, pages 1–6. IEEE, 2017

3. Marco Chiesa, Roberto di Lallo, Gabriele Lospoto, Habib Mostafaei, Massimo Rimondini, and Giuseppe Di Battista. Prixp: Preserving the privacy of routing policies at internet exchange points. In *Integrated Network and Service Management (IM), 2017 IFIP/IEEE Symposium on*, pages 435–441. IEEE, 2017

4. Roberto di Lallo, Federico Griscioli, Gabriele Lospoto, Habib Mostafaei, Maurizio Pizzonia, and Massimo Rimondini. Leveraging sdn to monitor critical infrastructure networks in a smarter way. In *Integrated Network and Service Management (IM), 2017 IFIP/IEEE Symposium on*, pages 608–611. IEEE, 2017

5. Habib Mostafaei, Gabriele Lospoto, Andrea Brandimartey, Roberto Di Lallo, Massimo Rimondini, and Giuseppe Di Battista. Sdns: Exploiting sdn and the dns to exchange traffic in a federated network. In *Network Softwarization (NetSoft), 2017 IEEE Conference on*, pages 1–5. IEEE, 2017

Under review papers

1. Habib Mostafaei, Gabriele Lospoto, Roberto di Lallo, Massimo Rimondini, and Giuseppe Di Battista. A framework for multi-provider virtual private networks in software-defined federated networks. *International Journal of Network Management*, 2019. Under review

2. Habib Mostafaei, Daniel Merling, and Michael Menth. Experience from a p4-based prototype for activity-based congestion management (abc). *IEEE Communications Magazine*, 2019. Under review

3. Davinder Kumar, Gabriele Lospoto, Habib Mostafaei, Marco Chiesa, and Giuseppe Di Battista. Desi: A decentralized software-defined network architecture for internet exchange points. *IEEE Transactions on Network and Service Management*, 2019. Under submission

Other Published Papers

1. H. Mostafaei. Energy-efficient algorithm for reliable routing of wireless sensor networks. *IEEE Transactions on Industrial Electronics*, 66(7):5567–5575, July 2019

2. Habib Mostafaei and Michael Menth. Software-defined wireless sensor networks: A survey. *Journal of Network and Computer Applications*, 119:42 – 56, 2018

3. Mahmood Javadi, Habib Mostafaei, Morshed U. Chowdhurry, and Jemal H. Abawajy. Learning automaton based topology control protocol for extending wireless sensor networks lifetime. *Journal of Network and Computer Applications*, 122:128 – 136, 2018

4. Habib Mostafaei, Antonio Montieri, Valerio Persico, and Antonio Pescapé. A sleep scheduling approach based on learning automata for wsn partialcoverage. *Journal of Network and Computer Applications*, 80:67–78, 2017

5. H. Mostafaei, M. U. Chowdhury, and M. S. Obaidat. Border surveillance with wsn systems in a distributed manner. *IEEE Systems Journal*, 12(4):3703–3712, Dec 2018

6. Paola G Vinueza Naranjo, Mohammad Shojafar, Habib Mostafaei, Zahra Pooranian, and Enzo Baccarelli. P-sep: A prolong stable election routing algorithm for energy-limited heterogeneous fog-supported wireless sensor networks. *The Journal of Supercomputing*, 73(2):733–755, 2017

7. Habib Mostafaei. Learning automaton-based self-protection algorithm for wireless sensor networks. *IET Networks*, 7:353–361(8), September 2018

8. Habib Mostafaei, Mohammad Shojafar, Bahman Zaher, and Mukesh Singhal. Barrier coverage of wsns with the imperialist competitive algorithm. *The Journal of Supercomputing*, 73(11):4957–4980, 2017

9. Habib Mostafaei and Mohammad S Obaidat. A greedy overlap-based algorithm for partial coverage of heterogeneous wsns. In *GLOBECOM 2017-2017 IEEE Global Communications Conference*, pages 1–6. IEEE, 2017

10. Habib Mostafaei and Mohammad S Obaidat. A distributed efficient algorithm for self-protection of wireless sensor networks. In *2018 IEEE International Conference on Communications (ICC)*, pages 1–6. IEEE, 2018

11. Habib Mostafaei, Antonio Montieri, Valerio Persico, and Antonio Pescapé. An efficient partial coverage algorithm for wireless sensor networks. In *2016 IEEE Symposium on Computers and Communication (ISCC)*, pages 501–506. IEEE, 2016

Bibliography

[Th] The P4 Language Consortium. P416 language specification. https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.pdf. Accessed June. 2018.

[ACC+17] G. Antichi, I. Castro, M. Chiesa, E. L. Fernandes, R. Lapeyrade, D. Kopp, J. H. Han, M. Bruyere, C. Dietzel, M. Gusat, A. W. Moore, P. Owezarski, S. Uhlig, and M. Canini. Endeavour: A scalable sdn architecture for real-world ixps. *IEEE Journal on Selected Areas in Communications*, 35(11):2553–2562, Nov 2017.

[AI16] AMS-IX. Amsterdam internet exchange point members, Sept 2016.

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

[AOC+10] Alberto Alvarez, Rafael Orea, Sergio Cabrero, Xabiel G. Pañeda, Roberto García, and David Melendi. Limitations of network emulation with single-machine and distributed ns-3. In *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques*, SIMUTools '10, pages 67:1–67:9, ICST, Brussels, Belgium, Belgium, 2010. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[APB09] Mark Allman, Vern Paxson, and Ethan Blanton. Tcp congestion control. RFC 5681, September 2009.

[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.

[ATG+16] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure linux containers with intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 689–703, Savannah, GA, 2016. USENIX Association.

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

[BDG+14] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[bea17] Beacon European Project. http://www.beacon-project.eu/, 2017.

[BGFV17] Rüdiger Birkner, Arpit Gupta, Nick Feamster, and Laurent Vanbever. Sdx-based flexibility or internet correctness?: Pick two! In *Proceedings of the Symposium on SDN Research*, SOSR '17, pages 1–7. ACM, 2017.

[BILD18] Gaetano Bonfiglio, Veronica Iovinella, Gabriele Lospoto, and Giuseppe Di Battista. Kathará: A container-based framework for implementing network function virtualization and software defined networks. In *Proc. IFIP/IEEE Network Operations and Management Symposium (NOMS 2018)*, 2018. To appear.

[bin17] Bind name server software. https://www.isc.org/downloads/bind/, Jan 2017.

[Bon11] Olivier Bonaventure. *Computer Networking: Principles, Protocols, and Practice*. The Saylor Foundation, 2011.

[Bro13] Broadband Internet Technical Advisory Group (BITAG). Real-time network management of internet congestion. Technical report, 2013.

[Bru16] Marc Bruyère. *An outright open source approach for simple and pragmatic Internet exchange*. PhD thesis, Université de Toulouse, Université Toulouse III-Paul Sabatier, 2016.

[CBB+17] C. Cascone, N. Bonelli, L. Bianchi, A. Capone, and B. Sansò. Towards approximate fair bandwidth sharing via dynamic priority queuing. In *2017 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*, pages 1–6, June 2017.

[CDA+16] Marco Chiesa, Christoph Dietzel, Gianni Antichi, Marc Bruyere, Ignacio Castro, Mitch Gusat, Thomas King, Andrew W. Moore, Thanh Dang Nguyen, Philippe Owezarski, Steve Uhlig, and Marco Canini. Inter-domain Networking Innovation on Steroids: Empowering IXPs with SDN Capabilities. *IEEE Communications Magazine*, 54(10):102–108, 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.

[CDC+17] Marco Chiesa, Daniel Demmler, Marco Canini, Michael Schapira, and Thomas Schneider. SIXPACK: Securing Internet eXchange Points Against Curious onlooKers. In *In Proc. CoNEXT 2017*, 2017.

[CdLL+17] Marco Chiesa, Roberto di Lallo, Gabriele Lospoto, Habib Mostafaei, Massimo Rimondini, and Giuseppe Di Battista. Prixp: Preserving the privacy of routing policies at internet exchange points. In *Integrated Network and Service Management (IM), 2017 IFIP/IEEE Symposium on*, pages 435–441. IEEE, 2017.

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

[CRB+11] Rodrigo N. Calheiros, Rajiv Ranjan, Anton Beloglazov, César 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.

[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.

[dLGL+17] Roberto di Lallo, Federico Griscioli, Gabriele Lospoto, Habib Mostafaei, Maurizio Pizzonia, and Massimo Rimondini. Leveraging sdn to monitor critical infrastructure networks in a smarter way. In *Integrated Network and Service Management (IM), 2017 IFIP/IEEE Symposium on*, pages 608–611. IEEE, 2017.

[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.

[EN16] Exa-Networks. Exabgp. https://github.com/Exa-Networks/exabgp, Sep 2016.

[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.

[Flo00] S. Floyd. Congestion Control Principles. RFC 2914, September 2000.

[FRA16] FRANCE-IX. France Internet eXchange Point members. https://www.franceix.net/en/france-ix-paris/members-in-paris/, Sept 2016.

[G+13] Technical Working Group et al. Real-time network management of internet congestion. Technical report, 2013.

[gea17a] Géant European Project. http://www.geant.net, 2017.

[gea17b] Géant European Project VPN services. https://www.geant.org/Services/Connectivity_and_network/Pages/VPN_Services.aspx, 2017.

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

[GMB+16] Arpit Gupta, Robert MacDavid, Rüdiger Birkner, Marco Canini, Nick Feamster, Jennifer Rexford, and Laurent Vanbever. An industrial-scale software defined internet exchange point. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, pages 1–14, Berkeley, CA, USA, 2016. USENIX Association.

[goo17] Google ipv6 statistics. https://www.google.com/intl/en/ipv6/statistics.html, Jan 2017.

[GR01] Lixin Gao and J. Rexford. Stable internet routing without global coordination. *IEEE/ACM Transactions on Networking*, 9(6):681–692, Dec 2001.

[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+15] 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. *ACM SIGCOMM Computer Communication Review*, 44(4):551–562, 2015.

[Has95] Dimitry Haskin. A bgp/idrp route server alternative to a full mesh routing. RFC 1863, October 1995.

[HVSC16] Siem Hermans, Ariën Vijn, Jeroen Schutrup, and Joris Claassen. On the feasibility of converting AMS-IX to an Industrial-Scale Software Defined Internet Exchange Point, 2016.

[IYZR16] Jared Ivey, Hemin Yang, Chuanji Zhang, and George Riley. Comparing a scalable sdn simulation framework built on ns-3 and dce with existing sdn simulators and emulators. In *Proceedings of the 2016 Annual ACM Conference on SIGSIM Principles of Advanced Discrete Simulation*, SIGSIM-PADS '16, pages 153–164, New York, NY, USA, 2016. ACM.

[JDS+16] Prerit Jain, Soham Jayesh Desai, Ming-Wei Shih, Taesoo Kim, Seong Min Kim, Jae-Hyuk Lee, Changho Choi, Youjung Shin, Brent Byunghoon Kang, and Dongsu Han. Opensgx: An open platform for sgx research. In *NDSS*, 2016.

[JHRB16a] 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.

[JHRB16b] Elisa Jasinska, Nick Hilliard, Robert Raszuk, and Neils Bakker. Internet exchange bgp route server. RFC 7947, September 2016.

[JITT16] Mona Jaber, Muhammad Ali Imran, Rahim Tafazolli, and Anvar Tukmanov. 5G Backhaul Challenges and Emerging Research Directions: A Survey. *IEEE Access*, 4:1743 – 1766, April 2016.

[JMCA18] Mahmood Javadi, Habib Mostafaei, Morshed U. Chowdhurry, and Jemal H. Abawajy. Learning automaton based topology control protocol for extending wireless sensor networks lifetime. *Journal of Network and Computer Applications*, 122:128 – 136, 2018.

[Jur13] P. Jurkiewicz. Link modeling using ns-3. https://github.com/mininet/mininet/wiki/Link-modeling-using-ns-3, Sep 2013.

[KARW16] Naga Katta, Omid Alipourfard, Jennifer Rexford, and David Walker. Cacheflow: Dependency-aware rule-caching for software-defined networks. In *Proceedings of the Symposium on SDN Research*, page 6. ACM, 2016.

[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.

[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.

[KLM+19] Davinder Kumar, Gabriele Lospoto, Habib Mostafaei, Marco Chiesa, and Giuseppe Di Battista. Desi: A decentralized software-defined network architecture for internet exchange points. *IEEE Transactions on Network and Service Management*, 2019. Under submission.

[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.

[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+15a] Diego Kreutz, Fernando M. V. Ramos, Paulo Esteves Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-Defined Networking: A Comprehensive Survey. *Proceedings of the IEEE* , 103(1):14–76, 2015.

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

[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.

[LHM10] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: Rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX, pages 19:1–19:6, New York, NY, USA, 2010. ACM.

[LIN16] LINX. London internet exchange point route server members. https://www.linx.net/tech-info-help/route-servers, Sept 2016.

[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.

[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, March 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.

[MBHM13] Ali Jose Mashtizadeh, Andrea Bittau, Yifeng Frank Huang, and David Mazieres. Replication, History, and Grafting in the Ori File System. In *In Proc. SOSP*, 2013.

[MCO18] H. Mostafaei, M. U. Chowdhury, and M. S. Obaidat. Border surveillance with wsn systems in a distributed manner. *IEEE Systems Journal*, 12(4):3703–3712, Dec 2018.

[MH17] Michael Menth and Frederik Hauser. On Moving Averages, Histograms and Time-Dependent Rates for Online Measurement. In *ACM/SPEC International Conference on Performance Engineering (ICPE)*, L'Aquila, Italy, April 2017.

[min17] Mininext, mininet extended. https://github.com/USC-NSL/miniNExT, Mar 2017.

[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.

[MLB+17a] 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.

[MLB+17b] Habib Mostafaei, Gabriele Lospoto, Andrea Brandimartey, Roberto Di Lallo, Massimo Rimondini, and Giuseppe Di Battista. Sdns: Exploiting sdn and the dns to exchange traffic in a federated network. In *Network Softwarization (NetSoft), 2017 IEEE Conference on*, pages 1–5. IEEE, 2017.

[MLdL+17] Habib Mostafaei, Gabriele Lospoto, Roberto di Lallo, Massimo Rimondini, and Giuseppe Di Battista. Sdnetkit: A testbed for experimenting sdn in multi-domain networks. In *Network Softwarization (NetSoft), 2017 IEEE Conference on*, pages 1–6. IEEE, 2017.

[MLdL+19] Habib Mostafaei, Gabriele Lospoto, Roberto di Lallo, Massimo Rimondini, and Giuseppe Di Battista. A framework for multi-provider virtual private networks in software-defined federated networks. *International Journal of Network Management*, 2019. Under review.

[MM18] Habib Mostafaei and Michael Menth. Software-defined wireless sensor networks: A survey. *Journal of Network and Computer Applications*, 119:42 – 56, 2018.

[MMM19] Habib Mostafaei, Daniel Merling, and Michael Menth. Experience from a p4-based prototype for activity-based congestion management (abc). *IEEE Communications Magazine*, 2019. Under review.

[MMO+18] Habib Mostafaei, Michael Menth, Mohammad S Obaidat, et al. A learning automaton-based controller placement algorithm for software-defined networks. pages 1–6, 2018.

[MMPP16] Habib Mostafaei, Antonio Montieri, Valerio Persico, and Antonio Pescapé. An efficient partial coverage algorithm for wireless sensor networks. In *2016 IEEE Symposium on Computers and Communication (ISCC)*, pages 501–506. IEEE, 2016.

[MMPP17] Habib Mostafaei, Antonio Montieri, Valerio Persico, and Antonio Pescapé. A sleep scheduling approach based on learning automata for wsn partialcoverage. *Journal of Network and Computer Applications*, 80:67–78, 2017.

[MO17] Habib Mostafaei and Mohammad S Obaidat. A greedy overlap-based algorithm for partial coverage of heterogeneous wsns. In *GLOBECOM 2017-2017 IEEE Global Communications Conference*, pages 1–6. IEEE, 2017.

[MO18] Habib Mostafaei and Mohammad S Obaidat. A distributed efficient algorithm for self-protection of wireless sensor networks. In *2018 IEEE International Conference on Communications (ICC)*, pages 1–6. IEEE, 2018.

[Mos18] Habib Mostafaei. Learning automaton-based self-protection algorithm for wireless sensor networks. *IET Networks*, 7:353–361(8), September 2018.

[Mos19] H. Mostafaei. Energy-efficient algorithm for reliable routing of wireless sensor networks. *IEEE Transactions on Industrial Electronics*, 66(7):5567–5575, July 2019.

[MSZS17] Habib Mostafaei, Mohammad Shojafar, Bahman Zaher, and Mukesh Singhal. Barrier coverage of wsns with the imperialist competitive algorithm. *The Journal of Supercomputing*, 73(11):4957–4980, 2017.

[MZ18] M. Menth and N. Zeitler. Fair resource sharing for stateless-core packet-switched networks with prioritization. *IEEE Access*, 6:42702–42720, 2018.

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

[ns317] Openflow 1.3 module for ns-3. http://www.lrc.ic.unicamp.br/ofswitch13/, Apr 2017.

[NSM+17] Paola G Vinueza Naranjo, Mohammad Shojafar, Habib Mostafaei, Zahra Pooranian, and Enzo Baccarelli. P-sep: A prolong stable election routing algorithm for energy-limited heterogeneous fog-supported wireless sensor networks. *The Journal of Supercomputing*, 73(2):733–755, 2017.

[ofe17] Openflow in europe: Linking infrastructure and applications. http://www.fp7-ofelia.eu/, Mar 2017.

[Ope14] Open Networking Foundation. OpenFlow Switch Specification, version 1.3.4, Mar 2014.

[Ope16] OpenSGX. "opensgx: An open platform for intel sgx". https://github.com/sslab-gatech/opensgx/, Sep 2016.

[Ope18] Open Networking Foundation. OpenFlow Switch Specification, 2018.

[OTK+18] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. Varys: Protecting SGX enclaves from practical side-channel attacks. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 227–240, Boston, MA, 2018. USENIX Association.

[ovs17] Openvswitch. http://openvswitch.org/, Jan 2017.

[P4S17] The P4 Language Specification Version 1.0.4, May 2017.

[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.

[Pos81] Jon Postel. Transmission control protocol. RFC 793, September 1981.

[PR08] Maurizio Pizzonia and Massimo Rimondini. Netkit: easy emulation of complex networks on inexpensive hardware. In *Proceedings of the 4th International Conference on Testbeds and research infrastructures for the development of networks & communities*, page 7. ICST (Institute for Computer Sciences, Social-Informatics and ..., 2008.

[Ram13] Sebastian Rampfl. Network simulation and its limitations. In *Proceeding zum Seminar Future Internet (FI), Innovative Internet Technologien und Mobilkommunikation (IITM) und Autonomous Communication Networks (ACN)*, volume 57, 2013.

[rip17] Ripe ipv6 statistics. https://stats.labs.apnic.net/ipv6/, Jan 2017.

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

[RMF+13] Joshua Reich, Christopher Monsanto, Nate Foster, Jennifer Rexford, and David Walker. Modular sdn programming with pyretic. *Technical Reprot of USENIX*, 2013.

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

[RSF+14] 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. ACM, 2014.

[ryu17] Ryu: component-based software defined networking framework. https://osrg.github.io/ryu/, Jan 2017.

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

[SGH14] Y. Shuai, M. Gorius, and T. Herfet. Low-latency Dynamic Adaptive Video Streaming. In *In Proc. Broadband Multimedia Systems and Broadcasting (BMSB),*, 2014.

[SKC+17] Brandon Schlinker, Hyojeong Kim, Timothy Cui, Ethan Katz-Bassett, Harsha V Madhyastha, Italo Cunha, James Quinn, Saif Hasan, Petr Lapukhov, and Hongyi Zeng. Engineering egress with edge fabric. In *Proceedings of the ACM SIGCOMM 2017 Conference (SIGCOMM'17). ACM, New York, NY, USA*, 2017.

[SLAK18] Naveen Kr. Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. Approximating fair queueing on reconfigurable switches. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 1–16, Renton, WA, 2018. USENIX Association.

[SSZ98] Ion Stoica, Scott Shenker, and Hui Zhang. Core-stateless fair queueing: Achieving approximately fair bandwidth allocations in high speed networks. In *Proceedings of the ACM SIGCOMM '98 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '98, pages 118–130, New York, NY, USA, 1998. ACM.

[Sup16] Cisco Support. Configuring and verifying the bgp conditional advertisement feature. http://www.cisco.com/c/en/us/support/docs/ip/border-gateway-protocol-bgp/16137- cond-adv.html, Sep 2016.

[SVL+16] S. Salsano, P. L. Ventre, F. Lombardo, G. Siracusano, M. Gerola, E. Salvadori, M. Santuari, M. Campanella, and L. Prete. Hybrid

ip/sdn networking: Open implementation and experiment management tools. *IEEE Transactions on Network and Service Management*, 13(1):138–153, March 2016.

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

[The13] The Open Networking Foundation. Sdn architecture overview. Technical report, 2013.

[tom17] The topology management tool (tomato). http://tomato-lab.org/, Mar 2017.

[TSB18] Meysam Taassori, Ali Shafiee, and Rajeev Balasubramonian. Vault: Reducing paging overheads in sgx with efficient integrity verification structures. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 665–678, New York, NY, USA, 2018. ACM.

[TW96] Andrew S Tanenbaum and David Wetherall. *Computer networks*. Prentice hall, 1996.

[uml17] The User-mode Linux Kernel. http://user-mode-linux.sourceforge.net/, 2017.

[Uni16] Roma Tre University. "computer networks research groups". https://bitbucket.org/rdl87/prixp/src, Sep 2016.

[WBL+16] Yangyang Wang, Jun Bi, Pingping Lin, Yikai Lin, and Keyao Zhang. Sdi: a multi-domain sdn mechanism for fine-grained inter-domain routing. *Annals of Telecommunications*, 71(11):625–637, Dec 2016.

[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.

[WZHW17] G. Wang, Y. Zhao, J. Huang, and W. Wang. The controller placement problem in software defined networking: A survey. *IEEE Network*, 31(5):21–27, 2017.

[YMR+17] Kok-Kiong Yap, Murtaza Motiwala, Jeremy Rahe, Steve Padgett, Matthew Holliman, Gary Baldus, Marcus Hines, Taeeun Kim, Ashok Narayanan, Ankur Jain, et al. Taking the edge off with espresso: Scale, reliability and programmability for global internet peering. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 432–445. ACM, 2017.

[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. ACM, 2012.