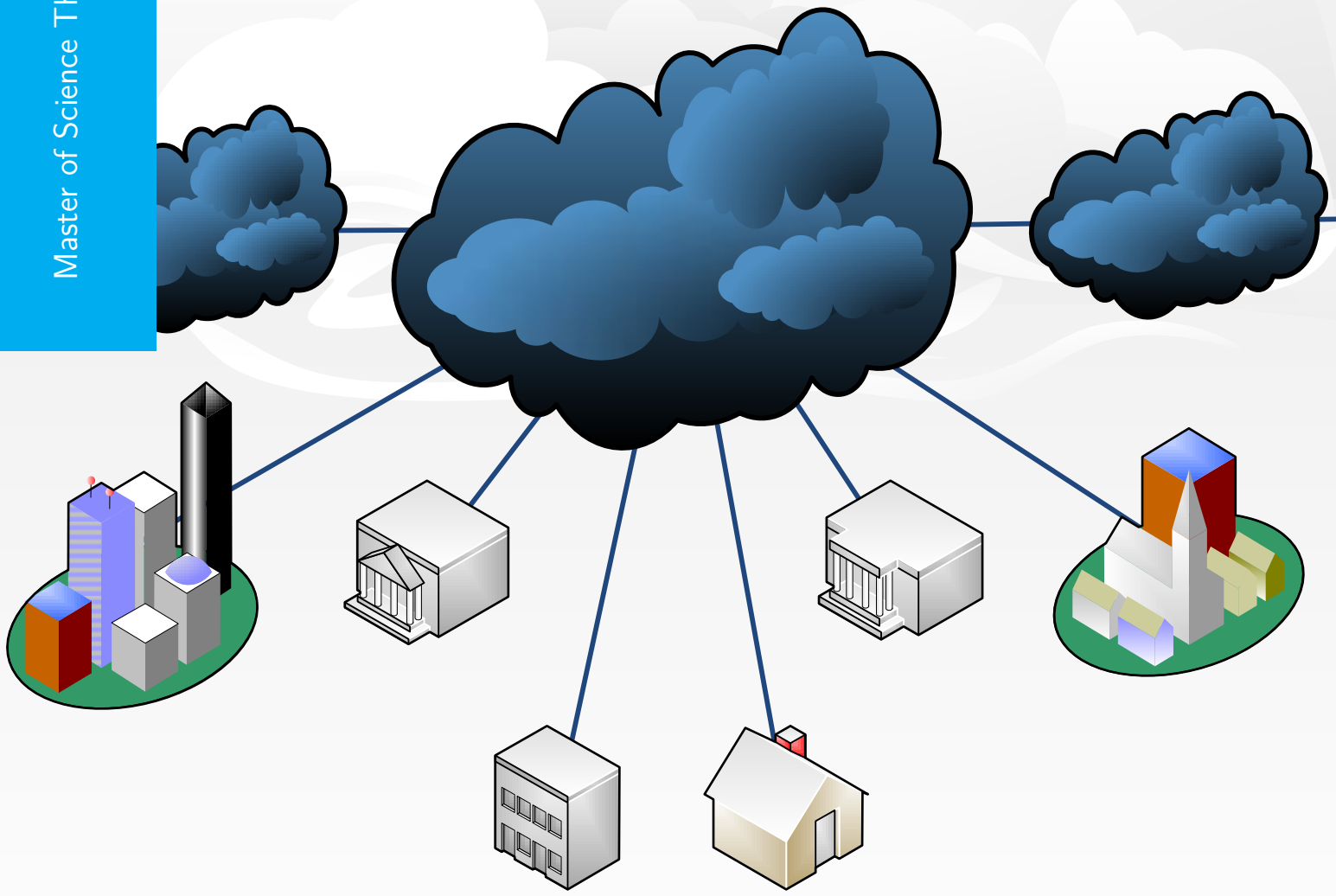
Network-as-a-Service Architecture with SDN and NFV

A Proposed Evolutionary Approach for Service Provider Networks

M.P.V. Manthena

Master of Science Thesis



TU Delft Delft University of Technology

Network Architectures and Services

PVM 2015-083

Faculty of Electrical Engineering, Mathematics and Computer Science
Network Architectures and Services Group

Network-as-a-Service Architecture with SDN and NFV

A Proposed Evolutionary Approach for Service Provider Networks

M.P.V. Manthena
4243846

Committee members:

Supervisor: Dr. Ir. F.A. Kuipers
Mentor: Ir. Casper van den Broek
Member: Dr. R. Venkatesha Prasad
Member: Ir. N.L.M. van Adrichem

February 20, 2015
M.Sc. Thesis No: PVM 2015-083

The work in this thesis was supported by TNO ICT, Delft. Their cooperation is hereby gratefully acknowledged.

Abstract

The Internet continues to grow exponentially with proliferation of devices and users being connected to it along with an exploding demand for various resource and performance intensive network services like multimedia content distribution, security, mobility, and machine-to-machine (M2M) communications. However, the current TCP/IP (Transmission Control Protocol/Internet Protocol) based Internet architecture, which was developed over 40 years ago and was not prepared nor designed to successfully meet such explosive demands of today, is leading to the growing ossification of the Internet with its increasingly closed, complex, and rigid state. Thus, limiting innovation in such networks and their corresponding services. To overcome this ossification problem of the Internet coupled with a lack of innovation in provisioning and management of network services, more and more service providers and network operators are embracing the concept of virtualization for their networks. This trend is largely inspired by the recent success of cloud-based service models along with their chief enabler virtualization in addressing similar problems in the computing and storage fields of Information Technology (IT). Although recent advances in the field of networking are witnessing new virtualization enabling network technologies being proposed, it is still a challenge to logically combine a set of them to realize cloud-based service models for service provider networks. This situation is mainly due to the concerns over these proposed technologies in terms of scalability, reliability, interoperability, and disruptive nature.

In this thesis, an evolutionary approach to implementing the Network-as-a-Service (NaaS) cloud-based service model for service provider networks is proposed with Software-Defined Networking (SDN) and Network Function Virtualization (NFV) as its key virtualization enabling network technologies. *In essence, the proposed evolutionary approach realizes the major benefits of network virtualization such as vendor-neutrality, simplicity, and flexibility while successfully addressing the stated concerns over SDN and NFV technologies in the proposed NaaS architecture.* Furthermore, a proof-of-concept (PoC) implementation of the proposed NaaS architecture on a physical network testbed is demonstrated along with an innovative provisioning and management of basic network connectivity services over it. Finally, the proposed evolutionary approach is validated by an experimental performance evaluation of the PoC physical network testbed along with the recommendations for improvement and future work.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I would like to take this opportunity to thank my thesis supervisor, Dr. Ir. Fernando A. Kuipers, for his guidance and assistance throughout my Master's thesis project and graduation process. During my thesis, he was instrumental in inspiring me to work towards solving a well-defined problem that would result in a publishable article and/or paper. It would not have been possible for me to complete this thesis and graduate without his guidance. I would also like to thank the Network Architectures and Services (NAS) research group at TU Delft for their support during my Master's specialization courses and thesis. A special mention and thanks to Ir. N.L.M. van Adrichem for his feedback and comments on final draft versions of my thesis report, article, and paper.

The work in this thesis was supported by TNO ICT, Delft. Their cooperation is hereby gratefully acknowledged. Most importantly, I would like to express my deepest gratitude to my thesis mentor at TNO, Ir. Casper van den Broek, for his continuous support, encouragement, and advice on my work. During my work at TNO, he always made sure that I have the right amount of space, time, and resources to discover and sort things on my own. Thus, I could not have asked for anything more. A special mention and thanks to Mr. Tim Daeleman, previously employed at TNO, for his support during the initial phases of my thesis work at TNO. I would also like to thank Mr. Arjen Holtzer, Mr. Borgert van der Kluit, and Mr. Otto Baijer for their assistance during the setting up of the proof of concept (PoC) physical network testbed in TNO's IP lab.

Additionally, I would like to extend my appreciation to the open source developer communities of OpenDaylight and Open vSwitch for their willingness to collaborate and share knowledge.

I would like to thank all my professors, colleagues, and friends at both TU Delft and TNO for their cooperation and support. A special mention and thanks to my MSc program coordinator, Dr. Ir. Gerard Janssen, for his guidance and assistance throughout the program.

Finally, I am forever grateful to my family for their love, moral and financial support throughout the years. Without whose support, none of my achievements so far would have been possible.

Chapter 1

Introduction

1-1 Research Motivation

The Internet continues to impact the way we live by revolutionizing various aspects of our lives such as social, economic, health, and education. Meanwhile, the Internet itself is growing exponentially with ever increasing number of devices and users being connected to it along with an exploding demand for various resource and performance intensive network services like multimedia content distribution, security, mobility, and machine-to-machine (M2M) communications. It was predicted that the total number of global Internet users will reach the 3 billion mark (around 40% of the total world population) by the end of the year 2014, which is around 3 times the number in the year 2005 [12]. Furthermore, according to some predictions, the total number of devices that would be connected to the Internet will reach the 50 billion mark (around 6.5 devices per person) in the year 2020 from 12.5 billion devices (around 1.8 devices per person) in the year 2010 [13]. Finally, the total global Internet traffic was predicted to reach the 1 zettabytes (1000 exabytes) per year mark in the year 2016 from around 600 exabytes per year in the year 2013 [14]. This huge proliferation of devices and users in the Internet along with its explosive traffic growth rates is due to the advent of new information and communications technologies (ICT) like Internet of Things (IoT) involving M2M communications, 4G/Long Term Evolution (LTE) for mobile communications, multimedia Content Delivery Networks (CDNs), and Big Data analytics for intelligent cloud-based services and applications.

The current TCP/IP (Transmission Control Protocol/Internet Protocol) based Internet architecture, which was developed over 40 years ago, is not prepared nor designed to effectively and efficiently meet such explosive demands. Moreover, its end-to-end communication model along with its competing stakeholder roles is leading to its increasingly closed, complex, and rigid state. This phenomenon is also resulting in the large scale vendor lock-in of network components in the Internet and its constituent service provider networks, which besides its possible benefits of better support and interoperability, is in turn creating high barriers to entry and slowing down the adoption process of new and disruptive network architectures, technologies,

and services required to enable innovation in the Internet while holistically addressing its existing deficiencies. Thus, greatly limiting innovation in provisioning and management of such networks and their corresponding services. This situation of growing ossification of the Internet coupled with a lack of innovation in provisioning and management of network services is posing non-trivial challenges for several Internet service providers. Some of these significant challenges are high CAPEX and OPEX costs with low ROI, competition from free-riding and over-the-top (OTT) service providers (e.g. Skype, WhatsApp, and Netflix), resource over or under provisioning, and non-conformance with service level agreements (SLAs). Most of these challenges are directly or indirectly related to the three major (ossification) issues of service provider networks that are vendor lock-in, complexity, and inflexibility, which are (in general) deeply ingrained in the key characteristics of such networks.

To address the stated challenges and issues [15, 16, 17], more and more service providers and network operators are starting to embrace the concept of virtualization for their networks. This trend is largely inspired by the overwhelming success of virtualization in addressing similar challenges and issues in the computing and storage fields of Information Technology (IT). Moreover, virtualization is the chief enabler for various cloud-based service models (e.g. IaaS, PaaS, and SaaS), which are in turn largely successful for enabling innovation in provisioning and management of computing and storage services. In essence, virtualization involves vendor-neutral resource abstraction to facilitate flexible, isolated, and efficient utilization of underlying resources while enabling innovation in provisioning and management of services running over it.

In the recent past, as a step towards network virtualization, several service providers and network operators have implemented virtual overlay networks based on encapsulation techniques, which use tunneling, tagging, and labeling protocols such as GRE, PPTP, L2TP, VLAN, VXLAN, and MPLS. Although this type of network virtualization implementation enables innovation by provisioning various customizable network services like migration support to new protocols and technologies (e.g. IPv6), multicast, mobility, and security (e.g. VPN), it does not change the key characteristics of service provider networks (vendor lock-in, complexity, and inflexibility). Thus, it realizes only the short-term benefits of network virtualization for service provider networks. Moreover, overlay networks add additional layers of complexity (encapsulation protocols) to the already (ossified) closed, complex, and rigid network architecture and its components. To overcome this and realize the full potential of network virtualization in solving the stated issues, one needs to essentially enable virtualization in the key components of today's network architecture, which are network control and functions (e.g. routers, firewalls, load balancers, NAT, DNS, and other dedicated network servers).

Recent advances in the field of networking are witnessing new virtualization enabling network technologies being proposed. Among which, Software-Defined Networking (SDN) and Network Function Virtualization (NFV) are at the forefront of current research and innovation in the networking community. SDN and NFV are highly complementary, but are independent of each other. SDN primarily involves implementing the network control plane (intelligence) in a logically centralized and fully-programmable software platform by decoupling it from the underlying network data-forwarding plane (hardware), whereas NFV primarily involves implementing network functions in an open and standardized IT virtualization environment as opposed to vendor-specific and dedicated hardware. In essence, SDN enables network simplicity and lower-layer (L2-L4) resource flexibility, whereas NFV avoids vendor lock-in and enables higher-layer (L4-L7) resource flexibility. Moreover, open standards based SDN (e.g.

OpenFlow [18, 19], the de-facto standard for SDN) avoids vendor lock-in. Thus, a logical combination of SDN and NFV could potentially avoid the vendor lock-in, complexity, and inflexibility of current service provider networks while enabling them with the long-term benefits of network virtualization.

The Network-as-a-Service (NaaS) cloud-based service model leveraging this type of network virtualization implementation (logical combination of SDN and NFV) could potentially enable innovation in provisioning and management of network services while solving the major (ossification) issues, which are vendor lock-in, complexity, and inflexibility, in service provider networks. In essence, NaaS is a cloud-based service model, which is similar to IaaS, PaaS, and SaaS, that offers on-demand, customizable, and utility-based innovative network connectivity services virtually over the Internet and its constituent service provider networks. However, it is still a challenge to realize this type of (NaaS) cloud-based service model for service provider networks. This situation is mainly due to the concerns over both SDN and NFV technologies in terms of scalability, reliability, interoperability, and disruptive nature.

1-2 Problem Description

During the preliminary research phase, it was found that the NaaS cloud-based service model with SDN and NFV as its key virtualization enabling network technologies could potentially solve the major (ossification) issues, which are vendor lock-in, complexity, and inflexibility, in service provider networks while enabling innovation in provisioning and management of their network services. However, it was found that realizing this type of (NaaS) cloud-based service model for service provider networks still remains as a challenge due to the concerns over both SDN and NFV technologies in terms of scalability, reliability, interoperability, and disruptive nature. Thus, to realize this NaaS cloud-based service model for service provider networks, the following two problems must be solved. The first problem involves implementing a full-stack (complete and modular) network architecture based on the NaaS cloud-based service model with SDN and NFV as its key virtualization enabling network technologies. The second problem involves successfully addressing the concerns in terms of scalability, reliability, interoperability, and disruptive nature over SDN and NFV technologies in the proposed NaaS architecture. Both of these problems must be holistically solved to realize this NaaS cloud-based service model for service provider networks.

1-3 Research Objective

The main research objective for this thesis is to realize the NaaS cloud-based service model for service provider networks with SDN and NFV as its key virtualization enabling network technologies to solve their major (ossification) issues in terms of vendor lock-in, complexity, and inflexibility while enabling innovation in provisioning and management of their network services. From this main objective the following step-by-step sub-objectives are derived:

1. To Identify the related work and state-of-the-art research in the field of networking, which is relating to the NaaS cloud-based service model, SDN, and NFV technologies and their implementations for networks;
2. To Identify the relevant, including the existing (traditional), network technologies and architectures along with their major benefits, issues, and concerns;
3. To propose an implementation approach and its strategy to realize the main research objective for this thesis while holistically addressing the identified issues and concerns over the involved technologies in the proposed NaaS architecture;
4. To demonstrate a PoC implementation of the proposed NaaS architecture on a physical network testbed along with an innovative provisioning and management of basic network connectivity services over it;
5. To validate the proposed implementation approach by an experimental performance evaluation of the PoC physical network testbed.

1-4 Research Questions

From the research objectives for this thesis the following main research question can be deduced: *Which implementation approach and its strategy should be proposed that would realize the NaaS cloud-based service model for service provider networks with SDN and NFV as its key virtualization enabling network technologies while holistically addressing the major issues and concerns over the involved technologies in the proposed NaaS architecture?* From this main question the following step-by-step research questions for this thesis are derived:

1. What is the state-of-the-art research, including its shortcomings and potential enhancements, relating to the implementation of NaaS cloud-based service model, SDN, and NFV technologies for networks?
2. What are the major benefits, issues, and concerns relating to the relevant, including the existing, network technologies and architectures, which are in accordance with the main research question for this thesis?
3. How to propose an implementation approach and its strategy, which is the resulting NaaS architecture based on identified network technologies and architectures, that solves the main research question for this thesis?

4. How to demonstrate a PoC implementation of the proposed NaaS architecture on a physical network testbed along with an innovative provisioning and management of basic network connectivity services over it?

5. How to validate the proposed implementation approach by an experimental performance evaluation of the PoC physical network testbed?

1-5 Research Scope

The work in this thesis primarily focuses on single domain service provider core and edge networks, and it does not take into account any federated and multi domain network architectures. Moreover, it does not consider any stakeholder roles (e.g. infrastructure provider, virtual infrastructure provider, network operator, and service provider) and their corresponding interactions in the proposed NaaS architecture. Nevertheless, the proposed NaaS architecture can be easily extended to any type of network architecture due to its inherent highly modular and abstract nature.

In general, this thesis aims at successfully answering the proposed research questions. Furthermore, the basic network connectivity services in the PoC implementation of the proposed NaaS architecture on a physical network testbed involve the following virtualized network functions: *basic connectivity, firewalling, optimal path computation, and load balancing.* Finally, the validation of the proposed implementation approach by an experimental performance evaluation of the PoC physical network testbed involves the performance analysis of the involved network control overhead and basic network connectivity services.

1-6 Related Work

As far as the problem description of this thesis is concerned, the related work and state-of-the-art research in the field of networking can be classified into two distinct groups each of them addressing a distinct problem in implementing the NaaS cloud-based service model for service provider networks (existing network architectures) with SDN and NFV as its key virtualization enabling network technologies. Accordingly, at first, the related work that focuses on the full-stack (complete and modular) implementation of NaaS cloud-based service model, SDN, and NFV technologies for networks is discussed (subsection 1-6-1). Later, the related work that mainly focuses on addressing the concerns in terms of scalability, reliability, interoperability, and disruptive nature over SDN and NFV technologies is discussed (subsection 1-6-2).

1-6-1 Full-stack Implementation

Duan et al. [2] presented a comprehensive survey on how service-oriented architecture (SOA) principles when applied to network virtualization in telecommunications and the future Internet will enable the NaaS paradigm, which in turn may facilitate a highly complementary convergence of networking and cloud computing. In the survey, they only considered and analyzed the high-level (application and service level) requirements of such approaches and frameworks. As a development in this direction, the author in [20] presented a framework that integrates the NaaS paradigm with SDN by abstracting the SDN control plane to implement a high-level network service orchestration model based on SOA principles. This orchestration model was proposed to enable application-aware network services with end-to-end Quality of Service (QoS) provisioning. Similarly, Bueno et al. [21] proposed a software framework based on NaaS paradigm and OpenFlow based SDN solution along with dynamic network configuration and status monitoring to enable on-demand, customizable, and application-aware network services with end-to-end QoS guarantees. However, none of these stated approaches and frameworks have been validated by a proof of concept implementation on a physical network testbed and its corresponding experimental performance evaluation.

On the other hand, Gouveia et al. [22] proposed a framework that realizes the full-stack implementation of OpenFlow based SDN solution in provisioning and managing network connectivity services both effectively and efficiently, which is validated by a network testbed implementation and its corresponding experimental performance evaluation. However, this framework needs to be further optimized and adapted to be relevant for existing network architectures, and it does not consider any corresponding implementation approach and strategy. Nevertheless, it was already shown that OpenFlow based SDN solution can implement, improve, and optimize any type of traditional and complex control plane such as MPLS Traffic Engineering [23], MPLS-based VPNs [23], and Path Computational Element (PCE) [24]. Finally, various complementary implementations of NFV leveraging the OpenFlow based SDN solution [25, 26, 27] are being proposed to implement existing network functions in a full-stack virtualized environment, which realizes the long-term benefits of network virtualization for existing network architectures.

In conclusion, the stated related work mainly focuses on the full-stack implementation of SDN and NFV technologies, but none of them address the relevant concerns in terms of scalability, reliability, interoperability, and disruptive nature over these virtualization enabling

network technologies in existing network architectures, which is essential for realizing and implementing the NaaS cloud-based service model for service provider networks.

1-6-2 Addressing Concerns

Casado et al. [1] presented the major drawbacks of OpenFlow based SDN architecture in terms of complexity in network address mapping and evolutionary inflexibility of corresponding network core and edge, and proposed a hybrid approach that retrospectively applies the insights underlying MPLS, which are simplified hardware along with distinction between network core and edge, to the OpenFlow based SDN architecture to overcome those major drawbacks. This hybrid approach, which was inspired from the idea of network fabrics, involves implementing the network core as a simple network fabric (fast and cheap packet transportation) while pushing the complexity (complex network functions and operations) towards the network edge to facilitate flexible and independent evolution of both network core and edge. In Figure 1-1, the proposed network fabric design by Casado et al. [1] is shown.

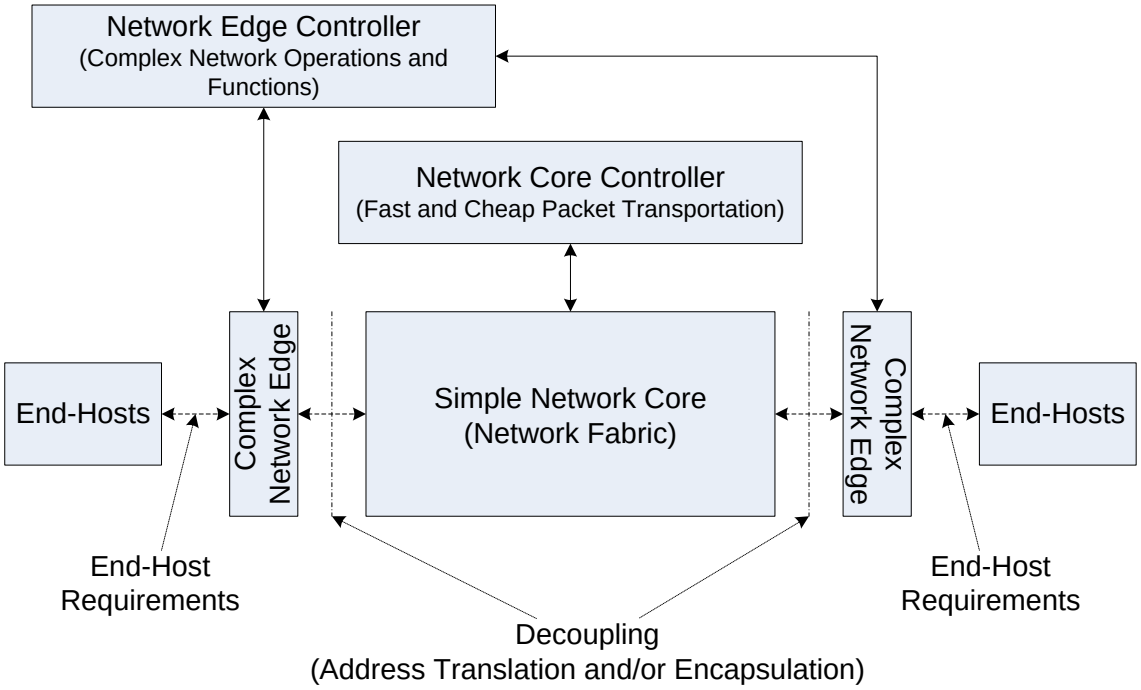


Figure 1-1: The proposed network fabric design by Casado et al. [1].

As seen in Figure 1-1, the simple network core (network fabric) is decoupled from the end-host requirements and the complex network edge context to facilitate flexible and independent evolution of both network core and edge. Thus, facilitating the decoupling of network core from its edge. However, this hybrid approach does not explicitly consider any deployment scenario (implementation approach and strategy) for OpenFlow based SDN solution in existing network architectures.

On the other hand, Hampel et al. [28] proposed an approach for extending the concept of SDN to tackle challenges of telecom domain in terms of over dependence on tunneling solu-

tions from specialized and vendor-specific edge gateways without any (flow-level) granularity in network control over them. This approach involves replacing only the specialized edge gateways with SDN nodes (incremental deployment) and equipping those nodes with encapsulation techniques on top of IP fundamental operations, which is named by the authors as vertical forwarding. However, this proposed approach does not facilitate flexible and independent evolution of network architecture as it is based on tunneling on top of IP fundamental operations, which realizes only the short-term benefits of network virtualization for existing network architectures.

To address the scalability and reliability concerns over OpenFlow based SDN solution, Mogul et al. [29] proposed a modification to the OpenFlow model, which is called as DevoFlow, for reducing unnecessary overheads in switch-controller interactions during flow setup and statistics gathering while increasing the overall network reliability. DevoFlow involves slight decoupling of centralized visibility from centralized control while trying to delegate most of the appropriate decisions to the underlying switches. Thus, reducing the involved overheads while increasing the overall network reliability (e.g. protection against single-point of failure). Finally, various complementary implementations of NFV leveraging the OpenFlow based SDN solution [25, 26, 27] are being proposed. Thus, the concerns over NFV are directly related to those of OpenFlow based SDN solution in such complementary implementations.

In conclusion, the stated related work mainly focuses on addressing the concerns over SDN and NFV technologies, but none of them present the full-stack implementation of their proposed solutions, which is essential for realizing and implementing the NaaS cloud-based service model for service provider networks.

1-7 Contribution

As far as the contribution of this thesis and its relation to the related work and state-of-the-art research in the field of networking is concerned, the proposed implementation approach of this thesis aims at making progress in both directions of the stated related work and state-of-the-art research in the field of networking, which is presented and comprehensively discussed in Chapters 3, 4, and 5 of this thesis. The proposed implementation approach builds on the positives of the stated related work while addressing their corresponding stated drawbacks. In essence, the proposed implementation approach aims to holistically address both of the distinct problems that are stated in the problem description of this thesis, which are full-stack implementation and addressing the concerns over SDN and NFV technologies in the proposed NaaS architecture, to realize and implement the NaaS cloud-based service model for service provider networks. In general, this thesis aims at successfully answering the proposed research questions.

1-8 Thesis Structure

This thesis is organized to answer the proposed research questions in a step-by-step fashion and is accordingly structured as follows. Chapter 2 introduces and discusses the relevant, including the existing, network technologies and architectures along with their major benefits, issues, and concerns. In Chapter 3, the proposed implementation approach and its strategy is introduced and discussed, which aims at realizing the NaaS cloud-based service model for service provider networks with SDN and NFV as its key virtualization enabling network technologies while holistically addressing the major issues and concerns over the involved technologies in the proposed NaaS architecture. In Chapter 4, a PoC implementation of the proposed NaaS architecture on a physical network testbed is demonstrated along with an innovative provisioning and management of basic network connectivity services over it. Chapter 5 presents performance evaluation experiments on the PoC physical network testbed, and an analysis and discussion of those experimental results to validate the proposed implementation approach. Finally, Chapter 6 concludes this thesis along with the recommendations for future work.

Chapter 2

Relevant Technologies and Architectures

In this chapter, the relevant, including the existing (traditional), network technologies and architectures are introduced and discussed along with their major benefits, issues, and concerns. This chapter includes the following network technologies and architectures: Network-as-a-Service (NaaS) cloud-based service model (section 2-1), Software-Defined Networking (SDN) along with the OpenFlow protocol and its switch specification (section 2-2), Network Function Virtualization (NFV) (section 2-3), TCP/IP (section 2-4), Multiprotocol Label Switching (MPLS) (section 2-5), state-of-the-art network management technologies involving network monitoring and configuration technologies (section 2-6), and Open vSwitch (OVS) implementations (section 2-7).

2-1 Network-as-a-Service

Network-as-a-Service (NaaS) is a cloud-based service model that offers network connectivity services virtually over the Internet and its constituent service provider networks. NaaS is also a cloud-based business model, which is similar to other cloud-based service models such as IaaS, PaaS, and SaaS, in which network connectivity services can be provisioned as utilities to its customers on a pay-per-use or monthly subscription basis. In essence, NaaS was proposed to enable on-demand, customizable, and innovative provisioning and management of network services by primarily leveraging virtualized network infrastructures and platforms (network virtualization). Furthermore, NaaS was proposed to transform the whole network architecture into a single big switch (hypothetical black box) that interacts and exchanges information with its customers and their corresponding applications through its web portals, dashboards, and externally exposed interfaces (APIs) while abstracting its internal details and complexities. Thus, it is the responsibility of the NaaS provider to orchestrate all the involved complex network and service management operations in meeting the on-demand, dynamic, and custom network service requirements of NaaS customers while continuously conforming

with the involved SLAs. Finally, this model was proposed to consider and also integrate networking, computing, and storage resources as a unified whole to facilitate a highly complementary convergence of networking and cloud computing. In Figure 2-1, an implementation of the NaaS cloud-based service model that leverages service-oriented architecture (SOA) and network virtualization principles, which is proposed by Duan et al. [2], is shown.

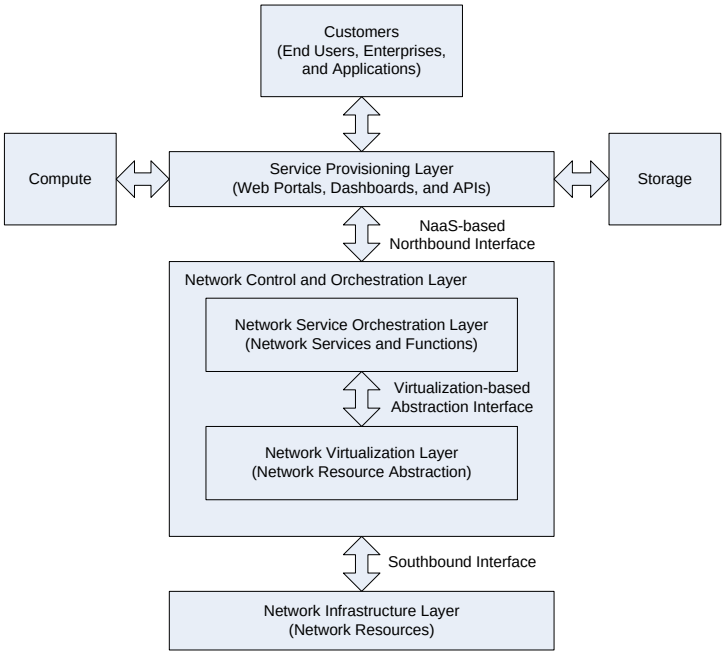


Figure 2-1: An implementation of the NaaS cloud-based service model that leverages service-oriented architecture (SOA) and network virtualization principles, which is proposed by Duan et al. [2].

As seen in Figure 2-1, the underlying network resources are abstracted through network virtualization and are exposed as software-based generic network capabilities for composing customizable network services and functions. These customizable network services and functions are further orchestrated as per the network service requirements of NaaS customers. Moreover, these customers are provisioned with simple and abstracted interfaces to subscribe and define their on-demand, dynamic, and custom network service requirements along with those of computing and storage cloud-based service models.

NaaS is not a new concept, but its adoption and deployment has been restricted so far due to the inability of Internet service providers in continuously conforming with the involved SLAs along with their lack of innovation in provisioning and management of network services. This situation is mainly due to the ossification of such networks. Nevertheless, in the recent past, the concept of network virtualization was proposed to effectively address and solve this problem of ossification in the Internet and its constituent service provider networks [15, 16, 17]. However, the current state-of-the-art network virtualization research and innovation still needs to address quite a few challenges and technical issues to successfully realize network virtualization for the Internet and its constituent service provider networks, which are the involved challenges in realizing an open, flexible, and heterogeneous networking environment for the Internet [30].

2-2 Software-Defined Networking

Software-Defined Networking (SDN) is an approach to networking that primarily involves implementing the network control plane (intelligence) in a logically centralized and fully-programmable software platform by decoupling it from the underlying network data-forwarding plane (hardware) through the process of abstraction. In essence, SDN was proposed to logically centralize network intelligence and state (global network view) while abstracting the underlying network resources (data forwarding function) as a set of fully-programmable software functions (facilitating network virtualization) to enable a simplified, flexible, and fully-programmable network control logic (software). Furthermore, this type of fully-programmable network control logic can be provisioned and exposed through APIs as a set of on-demand, customizable, and innovative network services to other internal and external applications like business software, orchestration software, and policy engines. Finally, SDN can be implemented in several ways primarily based on the method of communication employed between the control and the data plane in its architecture (type of control-data plane interface).

One such popular implementation is by using open-standards based OpenFlow protocol as the control-data plane interface [19]. OpenFlow is the de-facto standard for SDN due to the level of generality, flow-level granularity, and vendor-neutrality it provides to the decoupled network control plane while being an enabler of innovation in networks [18]. The OpenFlow protocol and its switch specification is further (comprehensively) introduced and discussed in subsection 2-2-1. Nevertheless, alternative implementations of SDN, which are ranging from logically centralized routing control platforms to vendor-specific SDN solutions, are also set out to make networks more programmable [31]. Some of these alternative implementations of SDN are briefly introduced and discussed below.

One of the early implementations of network control and data plane separation involved the work from Internet Engineering Task Force (IETF) working group ForCES (Forwarding and Control Element Separation) [32], which proposed a standard for an open interface (API) between the control and data planes to enable innovation in network control logic. However, over the years, the unwillingness of major network equipment vendors to adopt it due to its disruptive nature has greatly hindered its incremental deployment. In the recent past, there has been a fair amount of work that is directed towards the separation of control and data planes in the existing TCP/IP based network architectures, especially in the context of routing and signaling. Among which, works such as the Routing Control Platform (RCP) [33] and the Path Computation Element (PCE) [34] proposed a logically centralized network control. Furthermore, there are some solutions of network virtualization (virtual overlay networks) involving SDN technology as their chief enabler without any support from the underlying (existing) network hardware. Among which, the Network Virtualization Platform (NVP) from Nicira/VMware [35] employs SDN supporting software switches (Open vSwitch [36, 37]) in virtual machines to flexibly encapsulate their traffic and dynamically direct it across existing network hardware to create logical (virtual overlay) networks for its cloud tenants. Finally, major network equipment vendors like Cisco came up with their own vendor-specific SDN solutions that involve application-centric network programmability through their closed (distributed) network operating systems and interfaces [38].

Although most of these stated solutions are relatively pragmatic compared to the OpenFlow based SDN solution and can enable application specific innovation in network control logic,

they inherently lack generality (vendor-neutrality and simplicity) required to address the major (ossification) issues in today's networks. In general, there is no single (available) best solution of SDN that suits and fits all as each of them have their own benefits and tradeoffs. Thus, it is up to the individual service provider and network operator to decide on a particular SDN solution or combination of solutions for implementing SDN in their corresponding networks. In Figure 2-2, a logical view of the basic SDN architecture, which is proposed by the Open Networking Foundation (ONF) [3], is shown.

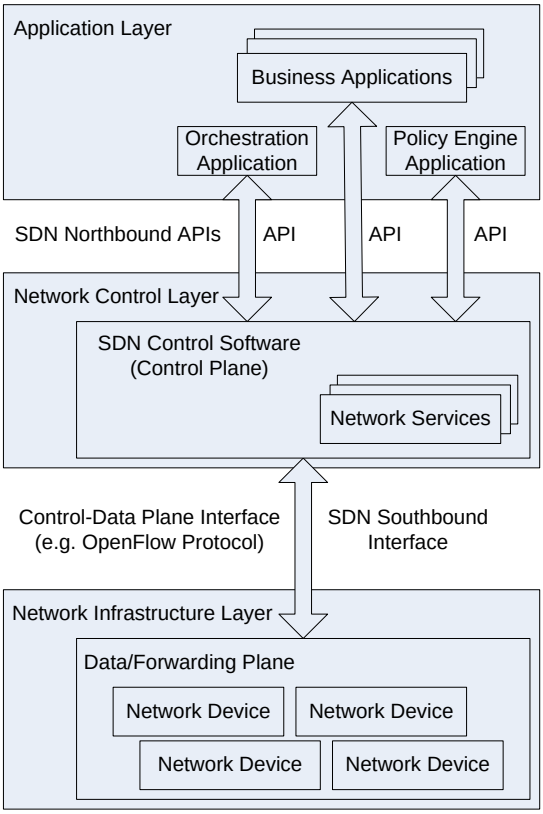


Figure 2-2: A logical view of the basic SDN architecture, which is proposed by the Open Networking Foundation (ONF) [3].

As seen in Figure 2-2, the network control plane is decoupled from the underlying network devices (data-forwarding plane) through the process of abstraction and represented in a logically centralized and fully-programmable software platform called SDN control software. This control software (logic) interacts with (manages, controls, and programs) the underlying data forwarding function through the control-data plane interface (e.g. OpenFlow protocol, the de-facto standard for SDN) called SDN southbound interface by leveraging the global network view and fully-programmable underlying network resource abstractions provided by the SDN architecture. Moreover, this control software is provisioned and exposed through SDN northbound APIs as a set of on-demand, customizable, and innovative network services to other internal and external applications like business software, orchestration software, and policy engines.

2-2-1 OpenFlow Protocol and Switch Specification

The OpenFlow based SDN solution evolved from the collaborative work done at Stanford University and UC Berkeley, in and around the year 2008, which collaboration is now an integral part of the Open Networking Research Center (ONRC) [39]. Initially, OpenFlow was proposed as a uniform way for researchers to run and evaluate experimental protocols in heterogeneous network devices (e.g. switches, routers, and access points) to enable innovation in campus networks [18]. Recently, since its inception in the year 2011 as a user-driven organization, ONF is promoting and managing the OpenFlow standard along with its corresponding SDN concepts and frameworks [19]. Moreover, ONF maintains and regularly updates the OpenFlow protocol and its switch specification documentation [40]. In Figure 2-3, an OpenFlow switch architecture, which is described in the OpenFlow switch specification [4], is shown.

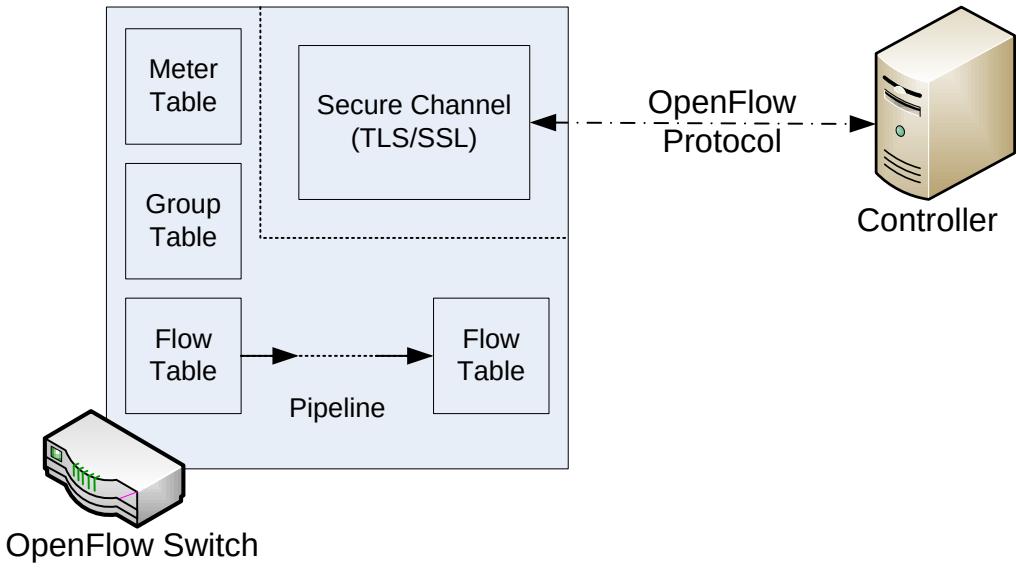


Figure 2-3: An OpenFlow switch architecture, which is described in the OpenFlow switch specification [4].

As seen in Figure 2-3, an OpenFlow switch consists of three main components: (1) one or more flow tables, a group table, and a meter table, which describe how to process and/or forward the incoming packet (traffic flow[1]) by matching it against their table (flow/group) entries, (2) a secure channel (TLS/SSL)to enable communication between the switch and the external controller, which manages the switch and controls its traffic flows by adding, updating, and deleting corresponding flow entries in flow tables, both reactively (upon packet/flow arrival) and proactively, and (3) the OpenFlow protocol, which defines an open and standard mechanism for communication between the switch and the external controller.

The OpenFlow pipeline processing involves multiple flow tables, and each flow table has multiple flow entries. In Table 2-1, the main components of a flow entry in a flow table of an OpenFlow switch, which is described in the OpenFlow switch specification [4], is shown.

[1]In general, a network traffic flow is a uniquely identified sequence of packets based on their header fields and ingress port.

Match Fields	Priority	Counters	Instructions	Timeouts	Cookie

Table 2-1: The main components of a flow entry in a flow table of an OpenFlow switch, which is described in the OpenFlow switch specification [4].

As seen in Table 2-1, each flow entry consists of six main components: (1) match fields, which include the ingress port and packet header fields, and optionally metadata specified by the previous table during the pipeline processing to match incoming packets against them (2) priority, which describes the matching precedence of a flow entry, (3) counters, which describe the matched packets to a flow entry, (4) instructions, which modify the action set or pipeline processing being applied to the matched packets, (5) timeouts, which define the maximum time or idle time before the flow entry is expired by the switch, and (6) cookie, which describes flow entry filtering data chosen by the external controller. In general, flow entries are uniquely identified by their match fields and priority. The flow entry that omits all the fields and has a priority 0 is called as the table-miss flow entry, which describes the further actions to be applied to an incoming packet in case it has no matching flow entry in that particular flow table.

The OpenFlow pipeline processing also involves a group table to which packets are directed by flow entries in flow tables during the pipeline processing to trigger additional methods of forwarding (e.g. flooding, multipath, and link aggregation) with a general layer of forwarding indirection on incoming traffic flows. Each group entry in a group table consists of a group identifier, a group type, counters, and action buckets (buckets containing set of actions to apply to matching packets). In Table 2-2, the main components of a group entry in the group table of an OpenFlow switch, which is described in the OpenFlow switch specification [4], is shown.

Group Identifier	Group Type	Counters	Action Buckets

Table 2-2: The main components of a group entry in the group table of an OpenFlow switch, which is described in the OpenFlow switch specification [4].

Finally, the OpenFlow pipeline processing involves a meter table to which packets are directed by flow entries in flow tables during the pipeline processing to trigger various performance-related (QoS) actions on incoming traffic flows. Each meter entry in a meter table consists of a meter identifier, meter bands, and counters. Meter bands specify various performance specific packet processing types and rates, and each meter band is further identified by its band type, rate, counters, and type specific arguments. In Table 2-3, the main components of a meter entry in the meter table of an OpenFlow switch, which is described in the OpenFlow switch specification [4], is shown.

Meter Identifier	Meter Bands	Counters

Table 2-3: The main components of a meter entry in the meter table of an OpenFlow switch, which is described in the OpenFlow switch specification [4].

The OpenFlow pipeline processing can be further explained by a flow diagram. In Figure 2-4, a flow diagram of the OpenFlow pipeline processing, which is described in the OpenFlow switch specification [4], is shown.

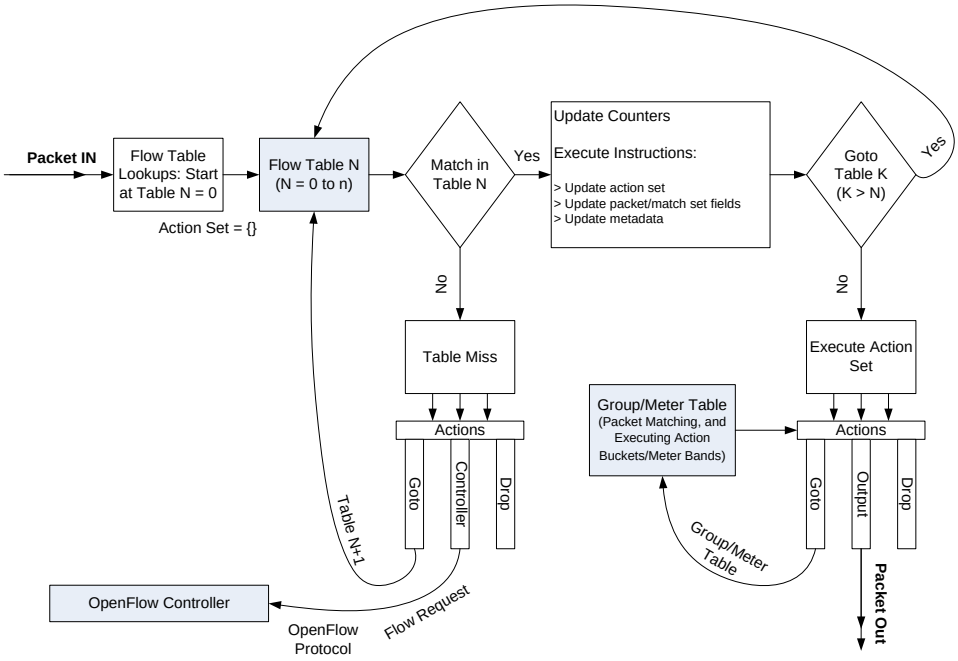


Figure 2-4: A flow diagram of the OpenFlow pipeline processing, which is described in the OpenFlow switch specification [4].

As seen in Figure 2-4, the OpenFlow pipeline processing always starts at flow table 0. The flow tables in an OpenFlow switch are sequentially numbered, starting from 0 till n, where number of flow tables in an OpenFlow switch are always greater than or equal to one ($n \geq 0$). Moreover, the OpenFlow pipeline processing always goes forward and never backwards. Upon arrival of an incoming packet through an ingress port, the packet is first matched against flow entries in flow table 0. If a flow entry match is found, the instruction set of that flow entry is executed, which may involve actions such as updating the packet action set, match fields, metadata, and goto flow table k. If the instruction set involves goto flow table K action, the updated packet is directed to the flow table K, where the stated process is repeated again. At any moment, if there is no goto flow table K action in the instruction set of a matched flow entry, the packet exists the OpenFlow pipeline and its action set is executed, which may result in actions such as goto group or meter table, output the packet through an egress port, and drop the packet. If a packet does not match any flow entry in a flow table, it is called as table-miss, and the packet by default matches the table-miss flow entry. Instruction set of a table-miss flow entry may involve actions such as drop the packet, send it to the external OpenFlow controller as a flow request via the OpenFlow protocol, and goto the next adjacent flow table. In general, every first packet of a new incoming flow is sent to the external OpenFlow controller as a flow request via the OpenFlow protocol, which processes the flow requests as per the user defined policies and instructs the underlying switches to install the corresponding flow entries in their flow tables.

The OpenFlow protocol enables an open, standard, and secure communication between the switch and the external controller, which primarily involves switch and flow management messages such as switch features and statistics polling by the controller, switch messages (both asynchronous and symmetric) to the controller, and controller instructions to the switch. Furthermore, the OpenFlow protocol and its switch specification support real-time network monitoring by exposing counters such as per flow table, per flow, per port, and per queue. Finally, the currently implemented OpenFlow switch specification version 1.3 additionally supports MPLS and IPv6 match fields along with their corresponding operations such as tunneling and tagging [4].

The OpenFlow controller is the most important element of the whole OpenFlow switch architecture. Thus, a lot of recent SDN efforts are focused on designing and implementing these controllers. NOX [41] is the first OpenFlow controller. It has been widely used as a base for OpenFlow experimentation and building new OpenFlow controllers. Throughout the past years, several open-source based OpenFlow supporting SDN controllers have been released. Among which, Floodlight [42] and OpenDaylight [43] are two of the most popular OpenFlow based SDN controllers due to their active development cycles and developer communities.

OpenFlow switches can be implemented either as OpenFlow-only switches supporting only OpenFlow operation or as OpenFlow-hybrid switches supporting both OpenFlow operation and normal Ethernet switching (L2/L3) operation. In the latter case, a classification mechanism for routing the incoming traffic between the OpenFlow pipeline processing and the normal pipeline processing is provided outside the OpenFlow context. Moreover, OpenFlow switches can be implemented as both physical and virtual switches (e.g. general-purpose x86/hypervisor-based, ASIC-based, and FPGA-based). Furthermore, an OpenFlow switch supports three types of ports, which are physical ports, logical ports, and reserved ports. Among which, reserved ports specify generic packet forwarding actions such as start of the OpenFlow pipeline, sending to the external controller, flooding or forwarding using traditional non-OpenFlow (normal) pipeline processing of an OpenFlow-hybrid switch. Finally, OpenFlow switches can be implemented as part of platform virtualization software and virtualized network testbeds to facilitate network virtualization and experimentation.

Recently, OpenFlow is being implemented in virtual switches such as Open vSwitch [36, 37], which can operate both as a software switch running in hypervisors and as the control stack in physical switches. The Open vSwitch implementations are further introduced and discussed in subsection 2-7. It is also being implemented in network emulators such as Mininet [44, 45], which creates virtual networks on laptops and PCs to enable rapid prototyping of SDNs. Alternatively, a special purpose OpenFlow controller called FlowVisor [46, 47] slices the underlying network resources to form isolated SDNs by acting as a transparent proxy between OpenFlow switches and multiple OpenFlow controllers.

In spite of all the promising opportunities and benefits associated with SDN, especially with the OpenFlow based SDN solution, it encounters certain technical challenges and concerns that are restricting its deployment in service provider networks. These encountered technical challenges and concerns are mostly in terms of its scalability, reliability, interoperability, and disruptive nature. Nevertheless, most of the current research on OpenFlow based SDN solution is directed towards addressing its technical challenges in terms of its scalability and reliability [48, 49]. However, there is hardly any research addressing the other involved major concerns (interoperability and disruptive nature).

2-3 Network Function Virtualization

Network Function Virtualization (NFV) is an approach to networking that primarily involves implementing network functions in an open and standardized IT virtualization environment as opposed to vendor-specific and dedicated hardware. In essence, NFV was proposed to decouple network functions (e.g. routers, firewalls, load balancers, NAT, DNS, and other dedicated network servers) from their dedicated hardware and implement them as software components on fully-virtualized network infrastructures by leveraging standard IT virtualization technologies and techniques to optimize and enable innovation in network service provisioning and management for service provider networks. Thus, this approach can greatly reduce the involved CAPEX and OPEX costs for service providers and network operators as it promotes the use of commodity hardware switches and servers instead of proprietary hardware appliances while involving only short and innovative software-based development and deployment cycles.

Furthermore, NFV is highly complementary to SDN, but both of them are independent of each other and can be implemented individually without other being required. In essence, SDN enables network simplicity and lower-layer (L2-L4) resource flexibility, whereas NFV avoids vendor lock-in and enables higher-layer (L4-L7) resource flexibility. Moreover, open standards based SDN (e.g. OpenFlow [18, 19], the de-facto standard for SDN) avoids vendor lock-in. Thus, a logically combined approach of these two concepts with open innovation can result in adding much more benefits and value to service provider networks as these concepts are mutually beneficial and can ease each other's implementation and deployment.

Finally, to make NFV a reality in future for service provider networks, a new network operator-led Industry Specification Group called "Network Functions Virtualisation" (NFV ISG) was formed under the umbrella organization European Telecommunications Standards Institute (ETSI). In Figure 2-5, the relationship between Network Functions Virtualisation and SDN, which is described in the NFV ISG introductory white paper [5], is shown.

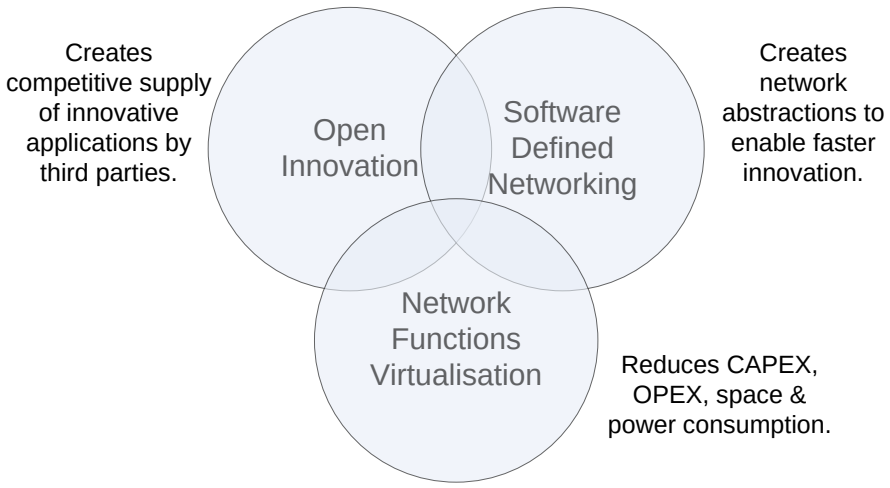


Figure 2-5: The relationship between Network Functions Virtualisation and SDN, which is described in the NFV ISG introductory white paper [5].

The main purpose of this group is to define the involved requirements and to develop an architecture for the virtualization of various possible network functions in service provider networks while addressing the technical challenges in doing so [6]. In Figure 2-6, the Network Functions Virtualisation architectural framework, which is described in the NFV ISG update white paper [6], is shown.

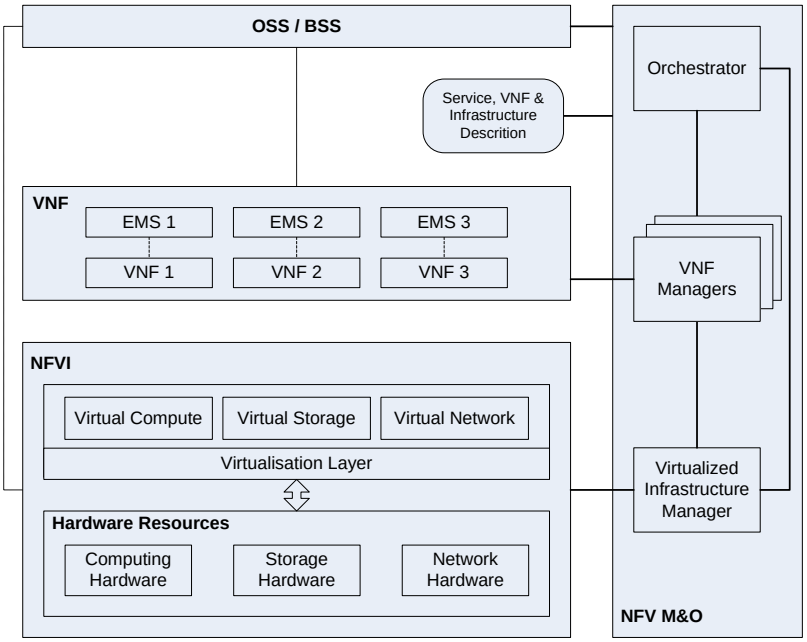


Figure 2-6: The Network Functions Virtualisation architectural framework, which is described in the NFV ISG update white paper [6].

As seen in Figure 2-6, the main components of the Network Functions Virtualisation architectural framework are the NFVI (Network Functions Virtualisation Infrastructure), the VNF (Virtualised Network Function), and the NFV M&O (Network Functions Virtualisation Management and Orchestration). The NFVI abstracts the underlying computing, storage, and network resources through virtualization and provides them as virtual resources to the VNF. The VNF consists of VNFs, which are the software implementations of (traditionally underlying) network functions. As per the requirements of network functions, each VNF can also be accompanied by an Element Management System (EMS). The NFV M&O involves orchestration and lifecycle management of physical and software resources including the VNFs. Moreover, this architectural framework was build to interact and co-exist with existing management platforms (e.g. OSS/BSS landscape). Finally, the entire system is driven by a set of metadata describing the involved requirements.

Although the NFV ISG group proposed the high-level requirements for NFV, it is still a challenge to implement virtualized network functions in service provider networks. This situation is mainly due to the availability of hardly any design and implementation specific details and research work required to realize such implementations. Nevertheless, various complementary implementations of NFV leveraging the OpenFlow based SDN solution [25, 26, 27] are being proposed to implement existing network functions in a full-stack virtualized environment. However, such combined approaches of NFV and SDN should first address the technical challenges and adoption concerns over SDN to realize NFV for service provider networks.

2-4 TCP/IP

TCP/IP (Transmission Control Protocol/Internet Protocol) is the most commonly used term to describe the Internet protocol suite, because TCP and IP are two of its most important and early standardized protocols. The Internet protocol suite is a set of communications protocols that define the networking model for the Internet and other private networks using it. In essence, the TCP/IP networking model follows a layered architecture that involves grouping of protocols into layers based on their generic functionality while abstracting them (layers) from each other through the process of encapsulation. Accordingly, the TCP/IP model has four layers of abstraction namely the link layer, the internet layer, the transport layer, and the application layer [7, 8]. In Figure 2-7, the basic architecture of TCP/IP (Internet protocol suite), which is described in the IETF RFCs (Internet standards) [7, 8], is shown.

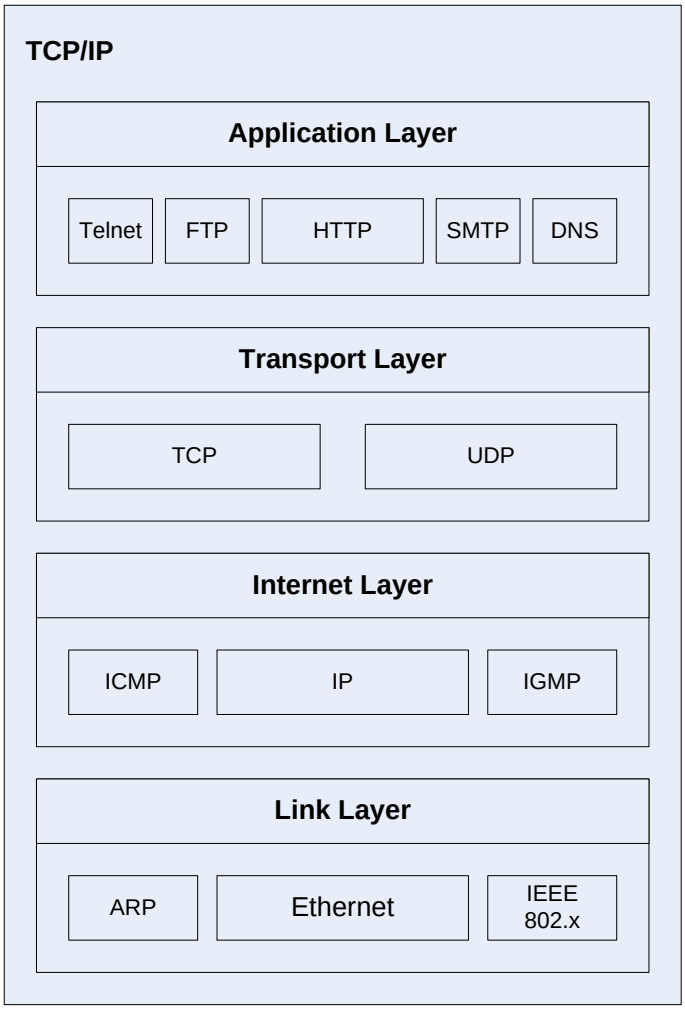


Figure 2-7: The basic architecture of TCP/IP (Internet protocol suite), which is described in the IETF RFCs (Internet standards) [7, 8].

As seen in Figure 2-7, the lowest layer in the TCP/IP architecture is called the link layer. The link layer is responsible for transmission and reception of TCP/IP packets on the network physical medium and it defines the local area networking methods and protocols (e.g. ARP).

In essence, TCP/IP architecture was designed to be independent of the underlying network physical medium, access method, technology, protocol, and format. Moreover, it always assumes an unreliable link layer. Thus, the link layer may constitute any type of LAN and WAN technologies like Ethernet, IEEE 803.x, Token Ring, X.25, Frame Relay, DSL, and ATM. The link layer provisions its services to the next layer in the stack (suite) called the Internet layer. The Internet layer provisions uniform networking methods and protocols across network boundaries (internetworking) and is responsible for uniquely addressing, encapsulating (packaging), and routing packets to their destination. The major protocols of the Internet layer are the Internet Protocol (IP), the Internet Control Message Protocol (ICMP), and the Internet Group Management Protocol (IGMP). The Internet layer further provisions its services to the next layer in the stack called the transport layer. The transport layer enables end-to-end (host-to-host) communication services and is responsible for provisioning datagram and session communication services to the next layer in the stack called the application layer. The major protocols of the transport layer are the Transmission Control Protocol (TCP) and the User Datagram protocol (UDP). TCP is a connection-oriented and reliable communication service, whereas UDP is a connectionless and unreliable communication service. The topmost layer called the application layer enables applications to exchange data with other applications and hosts by defining the corresponding protocols and providing them access to the services being provisioned by its underlying layers. Moreover, this layer also involves TCP/IP network management protocols. The most popular application layer protocols include the Hypertext Transfer Protocol (HTTP), the File Transfer Protocol (FTP), the Simple Mail Transfer Protocol (SMTP), the Telnet, and the Domain Name System (DNS).

Furthermore, the TCP/IP model has been the major enabler and driver for the Internet's worldwide success in terms of its continuing widespread adoption (scalability and reliability) and huge impact on the socio-economic aspects of people all over the world through its provisioned services such as World Wide Web (WWW), communication services, and multimedia. Finally, this huge success of the TCP/IP model is mainly due to some of its basic architectural principles such as layered architecture with abstractions, open standards based protocols, and common addressing scheme for all the TCP/IP enabled devices.

Besides the stated benefits of the TCP/IP model, it has quite a few limitations in terms of its increasingly complex protocol stack, high signaling overhead, competing stakeholder roles, and lack of effective and efficient network control and management. Moreover, TCP/IP's best effort service model is no longer applicable to most of the real-time applications (e.g. voice, video streaming, and online). Although IETF proposed QoS frameworks such as IntServ and DiffServ to enable much better QoS support to the network traffic in the present day Internet, they provide only short-term benefits to such networks and additionally add new layers of complexity (complex protocols) to the already complex TCP/IP protocol stack.

These limitations of the TCP/IP model lead to the growing ossification of the Internet coupled with a lack of innovation in provisioning and management of network services in its constituent service provider networks. Nevertheless, in the recent past, the concept of network virtualization was proposed to effectively address and solve this problem of ossification in the Internet and its constituent service provider networks [15, 16, 17]. However, the current state-of-the-art network virtualization research and innovation still needs to address quite a few challenges and technical issues to successfully realize network virtualization for the Internet and its constituent service provider networks, which are the involved challenges in realizing an open, flexible, and heterogeneous networking environment for the Internet [30].

2-5 Multiprotocol Label Switching

Multiprotocol Label Switching (MPLS) is a hop-by-hop data forwarding mechanism that is designed to enable fast switching in traditional TCP/IP based networks while provisioning them with end-to-end QoS support through its inherent traffic classification and prioritization capabilities (traffic engineering). In essence, for faster switching MPLS employs (fixed-length) label based forwarding table lookups instead of longest prefix matching (e.g. IP) while for the end-to-end QoS support MPLS classifies the incoming network traffic (packets) into forwarding equivalence classes (FECs) based on some predefined packet header match rules and their ingress ports, where each FEC is associated with (at least) a class of service and a label switched path (LSP) across the network [9]. MPLS has interfaces to existing routing protocols (e.g. RSVP, OSPF, and BGP) and requires no change to the existing Internet's backbone routing infrastructure. Moreover, routing in MPLS is done with existing IP routing protocols and is independent of the network layer and link layer protocol being used (protocol independent encapsulation).

MPLS operates by inserting its 32 bit header between traditional layer 2 and layer 3 header fields in a TCP/IP packet (encapsulation). This header consists of a stack of MPLS labels (label stack) with a size greater than or equal to one. MPLS Label stacking (a single packet carrying multiple labels, organized as a last-in-first-out stack) can be used to create tunnels for aggregation of multiple LSPs into a single LSP, failure protection of LSPs (bypass tunnel), creation of customizable VPN tunnels with QoS support, etc. In Table 2-4, the MPLS header format, which is described in the IETF RFC (proposed standard) [9], is shown.

Label (20 bits)	TC (3 bits)	S (1 bits)	TTL (28 bits)

Table 2-4: The MPLS header format, which is described in the IETF RFC (proposed standard) [9].

As seen in Table 2-4, the MPLS header format consists of a 20 bit label field that defines the packet's FEC. A three bit TC (Traffic Class) field in the MPLS header format defines the packet's class of service. An one bit S (Bottom of Stack) field in the MPLS header format defines whether the packet's topmost label is at the bottom of the label stack. Finally, an eight bit TTL (Time To Live) field in the MPLS header defines the packets' time to live. For multiprotocol label switching, MPLS uses a protocol independent encapsulation technique to push, swap, and pop labels on to the network traffic. In a MPLS domain, a MPLS capable router is called the Label Switching Router (LSR). A LSR at the edge of a MPLS domain is called as the Label Edge Router (LER), which are MPLS domain's ingress and egress LSRs. MPLS LERs in some contexts (e.g. service provider networks) are known as PE (Provider Edge) routers, and core MPLS LSRs are known as P (Provider) routers. In Figure 2-8, an example MPLS label switching operation, which is described in the IETF RFC (proposed standard) [9], is shown.

As seen in Figure 2-8, each packet at the ingress of a MPLS domain is assigned to a Forwarding Equivalence Class (FEC) by the ingress LSR. All the packets belonging to a particular FEC are treated in the same way by all the LSRs in that MPLS domain. Once the ingress LSR determines a packet's FEC, it inserts a 32 bit MPLS header between the link layer header and network layer header of that packet before forwarding it into the MPLS domain. At the

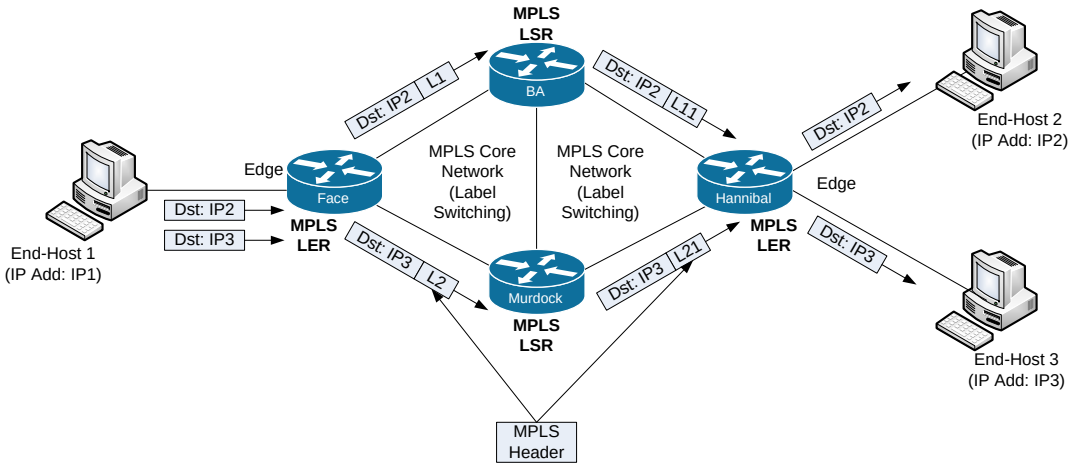


Figure 2-8: An example MPLS label switching operation, which is described in the IETF RFC (proposed standard) [9].

subsequent LSRs in the MPLS domain, the label is used as an index into a forwarding table (Label Forwarding Information Base (LFIB)) that specifies the next hop and a new label to that packet. At each LSR, the old label is replaced with the new label and the packet is forwarded to the next hop. Finally, the egress LSR strips the label and forwards the packet to its final destination based on the IP packet header. Alternatively, to reduce burden on the egress LSR, labels are striped from packets at the penultimate hop LSRs. This phenomenon is called as penultimate hop popping (PHP). Moreover, LER nodes have an additional label to IP prefix binding table called the Label Information Base (LIB) along with an IP forwarding table (FIB) for label binding of IP packets at ingress LER and label removal with subsequent layer 3 route lookup of IP packets at the egress LER. In this way, MPLS enables inter-connectivity growth of networks with minimal addition of network hardware.

For each FEC, a specific unidirectional path called the Label Switched Path (LSP) is assigned. To setup a LSP, each LSR must assign an incoming label to the LSP for the corresponding FEC and then inform its relevant upstream node about the assigned label while learning the label that its downstream node has already assigned to that LSP. These labels have only local significance. Thus, as a result of this label distributions, forwarding tables (LFIB) are created in the individual LSRs of a MPLS domain. This type of path setup mechanism requires label distribution and signaling protocols. Label Distribution Protocol (LDP) and Resource Reservation Protocol-Traffic Engineering (RSVP-TE) are two of the most popular label distribution protocols being used in MPLS today. Each of these label distribution protocols have their own individual advantages and disadvantages. LDP is used for its simplicity and quick path setup, whereas RSVP-TE is used for its good QoS (traffic engineering) support through its signaling protocol.

In general, LSP route setup can be done either by hop-by-hop routing or by explicit routing. Explicit routing has several advantages over hop-by-hop routing as it can establish LSPs based on policy and QoS requirements. It can also provision pre-established LSPs that can be used in case of failures (failure protection). Thus, explicit routing is mostly preferred for LSP route setup.

Furthermore, MPLS integrates the advantages of layer 3 routing (scalability and reliability) with that of layer 2 packet forwarding (fast switching in hardware) to enable several benefits like fast switching, traffic engineering, and provisioning of customizable network services (e.g. layer 2 and 3 VPNs) with end-to-end QoS support. Finally, MPLS facilitates evolutionary flexibility of networks as it mainly employs simplified switching hardware with a clear distinction between the network core and the edge, which involves simplified core that is decoupled from the complex edge to facilitate independent evolution of both.

Besides the stated benefits of MPLS, it has quite a few limitations in terms of its static, domain-specific, high signaling overhead, and non-application aware nature. Most of these limitations are due to the lack of dynamic network control and management in such networks. Nevertheless, concepts like software-defined and driven networking (e.g. OpenFlow, PCE, and OpenFlow based PCE) are being proposed to effectively address and solve this problem in service provider networks [23, 24]. However, these concepts are encountering some technical challenges and adoption concerns in terms of scalability, reliability, interoperability, and disruptive nature that needs to be addressed first to successfully realize them for service provider networks.

2-6 State-of-the-art Network Management Technologies

In this context, the state-of-the-art network management technologies that are relevant to the proposed implementation approach and its implementation strategy are briefly discussed and compared. In essence, network monitoring and configuration are two of the most significant network management operations for any network operator and service provider. Thus, this discussion and comparison is based on state-of-the-art network monitoring and configuration technologies. Moreover, the proposed implementation approach and its implementation strategy aims to achieve an open and heterogeneous network architecture for service provider networks, and therefore vendor-neutral and remote network management (both monitoring and configuration) technologies become very much relevant in this context.

2-6-1 Network Monitoring Technologies

As far as state-of-the-art network monitoring technologies are concerned, they can be primarily classified based on the level of granularity (e.g. per interface, per flow, and per packet) they provide in terms of network traffic monitoring. Furthermore, these technologies also differ from each other based on their features and benefits (e.g. vendor-neutrality, device performance and status monitoring, and low network overhead and costs) they offer to the network operator and service provider. Accordingly, the following four popular and vendor-neutral (open standards based) network monitoring technologies are compared in Table 2-5: Simple Network Management Protocol (SNMP) [50], Internet Protocol Flow Information Export (IPFIX) [51]/NetFlow[2] [52], sFlow [53], and port mirroring (packet capturing).

As seen in Table 2-5, each of the stated network monitoring technologies have their own advantages and shortcomings. Thus, there is no single (available) best solution for network

[2]NetFlow is a Cisco Systems proprietary standard for IP traffic flow monitoring on network routers, whose version 9 was published by them as an IETF informational RFC [52]. Moreover, the IETF Internet standard IPFIX [51] is based on this NetFlow version 9 IETF informational RFC.

Network Monitoring Technology	*Level of Granularity*	*Method of Monitoring*	*Transport Protocol*	*Network Device Support*	*Basic Applications*
SNMP	Per interface	Mostly pull-based	Unreliable UDP	CPU	Bandwidth and fault analysis
IPFIX / NetFlow[2]	Per flow	Mostly push-based	IPFIX: Prefers reliable SCTP; NetFlow: Unreliable UDP	CPU	IP traffic accounting and analysis on the routers
sFlow	Per configured sampling rate of packets and per interface	Push-based	Unreliable UDP	Hardware chip (e.g. ASIC)	Bandwidth and traffic analysis in high speed networks
Port mirroring	Per packet	Capturing packets through a mirrored interface	Reliable packet capturing from a mirrored interface	Hardware chip (e.g. ASIC)	Deep packet inspection and analysis

Table 2-5: A basic comparison of popular and vendor-neutral state-of-the-art network monitoring technologies.

monitoring that suits and fits all. Moreover, there are quite a few vendor-specific network monitoring solutions that are based on and variants of the stated network monitoring technologies, but none of them are relevant for open and heterogeneous network architectures. Finally, the OpenFlow protocol, the de-facto standard for SDN, besides flow-level network control also facilitates network monitoring with flow-level granularity, which exposes the installed flow, group, and meter statistics to the external OpenFlow controller [18, 54]. However, OpenFlow based network monitoring for large scale networks could potentially hamper the corresponding OpenFlow based SDN solution's scalability and reliability due to the involved high control overhead, load, and costs.

2-6-2 Network Configuration Technologies

As far as state-of-the-art network configuration technologies are concerned, they can be primarily classified based on the level of ease and flexibility they offer to the network operators and service providers in terms of configuring the underlying network devices. As far as the current practices are concerned, most of the network operators and service providers either

use vendor-specific configuration methods (e.g. web service based) or command line interfaces (CLIs) to configure each device on the network. However, only very few use vendor-neutral and remote network configuration technologies like SNMP[3] [50] and NETCONF (Network Configuration Protocol) [55]. Nevertheless, with recent advances in the field of networking in terms of new virtualization enabling network technologies being proposed (e.g. SDN and NFV), the requirement for vendor-neutral and remote network configuration technologies has become a necessity. Furthermore, new network configuration technologies like Open vSwitch Database (OVSDB) Management Protocol have been proposed and implemented for the vendor-neutral and remote configuration of Open vSwitch, the de-facto standard for open virtual switches, implementations [56]. Accordingly, a basic comparison of the following state-of-the-art network configuration technologies is done in Table 2-6: CLI, web service based, SNMP, NETCONF, and OVSDB Management Protocol.

As seen in Table 2-6, vendor-neutral and remote network configuration technologies have much more benefits (ease, flexibility, and interoperability[4]) compared to other models when it comes to implementing them in open and heterogeneous network architectures. However, some vendor-neutral solutions are easy to use once implemented, but their implementation can be very hard. Moreover, within the vendor scope, there are quite a few vendor-specific configuration tools that are easy to use and reasonably flexible. In general, there is no single (available) best solution that suits and fits all as each of them have their own benefits and tradeoffs. Thus, it is up to the individual service provider and network operator to decide on a particular network configuration solution or combination of solutions for implementing them in their corresponding networks. Finally, OpenFlow Management and Configuration Protocol (OF-CONFIG), which is based on NETCONF, was recently proposed for the vendor-neutral and remote configuration of the OpenFlow switch implementations [57].

2-7 Open vSwitch Implementations

Open vSwitch (OVS) is an open source, multilayer, and remotely programmable virtual (software) switch that was proposed to enable integration of networking into the virtualization layer [36, 37]. In essence, OVS facilitates flexible automation of network control and management by exposing its programmatic interfaces to external applications, services, and platforms (e.g. OpenFlow based SDN controller, OVSDB manager, OpenStack) while supporting traditional and standard network management technologies, protocols, and interfaces (e.g. NetFlow, sFlow, IPFIX, SPAN, RSPAN, CLI, LACP, VLAN, 802.1ag, and tunneling protocols). Furthermore, OVS can be implemented to operate as a software switch in a hypervisor, and as a multilayer control stack for switching hardware (e.g. ASICs). Thus, OVS has been ported to multiple software and hardware platforms like virtualization platforms and switching hardware (chipsets). Moreover, OVS is being supported by various hypervisors (e.g. XEN, KVM, VMware, and VirtualBox) and Linux based operating systems (e.g. Ubuntu, Debian, and Fedora) in both kernel and user space. Finally, the Mininet network emulator implements

[3] In spite of SNMP being a vendor-neutral (open standard based) and remote network configuration technology, it is not very popular because of its issues in terms of its complex MIB (Management Information Base) structure, non-transactional model, and security concerns. In essence, SNMP as a configuration technology is not very user friendly, but it is reasonably flexible. Nevertheless, SNMP is widely used for performance and fault monitoring in service provider networks.

[4] Interoperability with open and heterogeneous network architectures.

the kernel space OVS for rapid prototyping of SDNs [44, 45]. In Figure 2-9, the high-level architecture of Open vSwitch, which is described in the Open vSwitch project's Git repository [10], is shown.

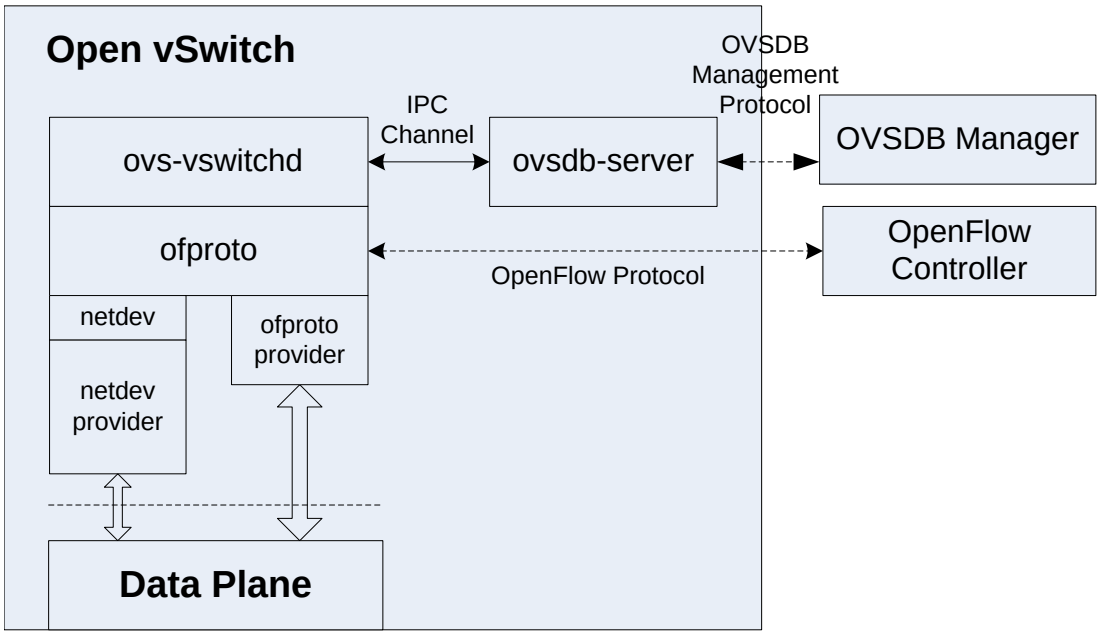


Figure 2-9: The high-level architecture of Open vSwitch, which is described in the Open vSwitch project's Git repository [10].

As seen in Figure 2-9, OVS can be easily ported to any type of data plane, and to both software and hardware platforms (e.g. user space and kernel space). In the OVS architecture, "ovs-vswitchd" is the main user space program that accesses configuration information from the "ovsdb-server" over an IPC channel and sends that information down to the "ofproto" library. Moreover, it sends back status and statistics from the "ofproto" library to the "ovsdb-server". "ovsdb-server" consists of switch-level configuration details such as bridge, interface, tunnel definitions, OVSDB manager, and OpenFlow controller addresses. Moreover, "ovsdb-server" can be remotely configured by an OVSDB manager via the OVSDB management protocol. Furthermore, "ofproto" library implements an OpenFlow switch and it talks to the external OpenFlow controller via the OpenFlow protocol. Moreover, It talks with the switching software and hardware through an "ofproto provider". Finally, "netdev" and "netdev provider" abstract the underlying switch interfaces for the OVS implementation.

Besides the stated benefits of OVS, some of its implementations have certain limitations in terms of slow software based switching, high load on CPU, version mismatches, and other generic problems related with open source based software development process. Nevertheless, the OVS developer community is collaborating in a large way to address these limitations. However, a particular implementation of the OVS should be thoroughly studied and tested before implementing it for a certain application and network type.

Network Configuration Technology	***Type of Network Configuration Technology***	***Type of Managed Network Devices***	***Type of Managed Networks***	***Level of Ease and Flexibility***	***Inter-operability***[4]
CLI	Vendor-specific	Any network device	Small and homo-geneous networks	Low	Low
Web service based	Vendor-specific	Some vendor-specific network devices	Large and homo-geneous networks	Low	Low
SNMP[3]	Vendor-neutral	Any network device	Large and hetero-geneous networks	Medium	Medium
NETCONF	Vendor-neutral	Any network device	Large and hetero-geneous networks	High	High
OVSDB Management Protocol	Vendor-neutral	Open virtual switches (Open vSwitch)	Large and Open vSwitch based networks	High	High

Table 2-6: A basic comparison of state-of-the-art network configuration technologies.

Chapter 3

Proposed Evolutionary Approach and Implementation Strategy

In this chapter, the proposed implementation approach and its strategy for service provider networks is presented and discussed. In accordance with the thesis problem statement and objective, the proposed implementation approach involves an evolutionary approach to implementing the Network-as-a-Service (NaaS) cloud-based service model for service provider networks. The proposed NaaS architecture involves SDN and NFV as its key virtualization enabling network technologies. OpenFlow, the de-facto standard for SDN, is used as the SDN implementation in the proposed NaaS architecture due to its inherent qualities of open nature and as an enabler of innovation in networks [18, 19]. Moreover, the proposed NaaS architecture uses network monitoring and configuration technologies for more scalable, reliable, and granular closed-loop network control and management. Fundamentally, the proposed evolutionary approach, instead of revolutionizing the whole network architecture with the disruptive SDN and NFV technologies (instead of upgrading the whole network at once), involves an implementation strategy that facilitates an incremental deployment scenario through a highly complementary co-existence between these disruptive technologies and the most prominent existing network technologies, which are TCP/IP and MPLS, in service provider networks. In Figure 3-1, a typical single domain service provider network is shown.

As seen in Figure 3-1, provider and provider edge routers are labeled as P and PE respectively. Similarly, customer edge routers are labeled in the figure as CE. In general, provider routers are MPLS label switching routers and provider edge routers are MPLS label edge routers, which are labeled in the figure as LSR and LER respectively. In essence, the proposed evolutionary approach realizes the major benefits of network virtualization such as vendor-neutrality, simplicity, and flexibility while successfully addressing the concerns over SDN and NFV technologies in terms of scalability, reliability, and interoperability. Accordingly, at first, the data plane considerations (section 3-1) followed by the control and management plane considerations (section 3-2) for implementing the proposed evolutionary approach in service provider networks are presented and discussed. Finally, this chapter concludes by presenting and discussing the overall proposed Network-as-a-Service (NaaS) architecture (section 3-3).

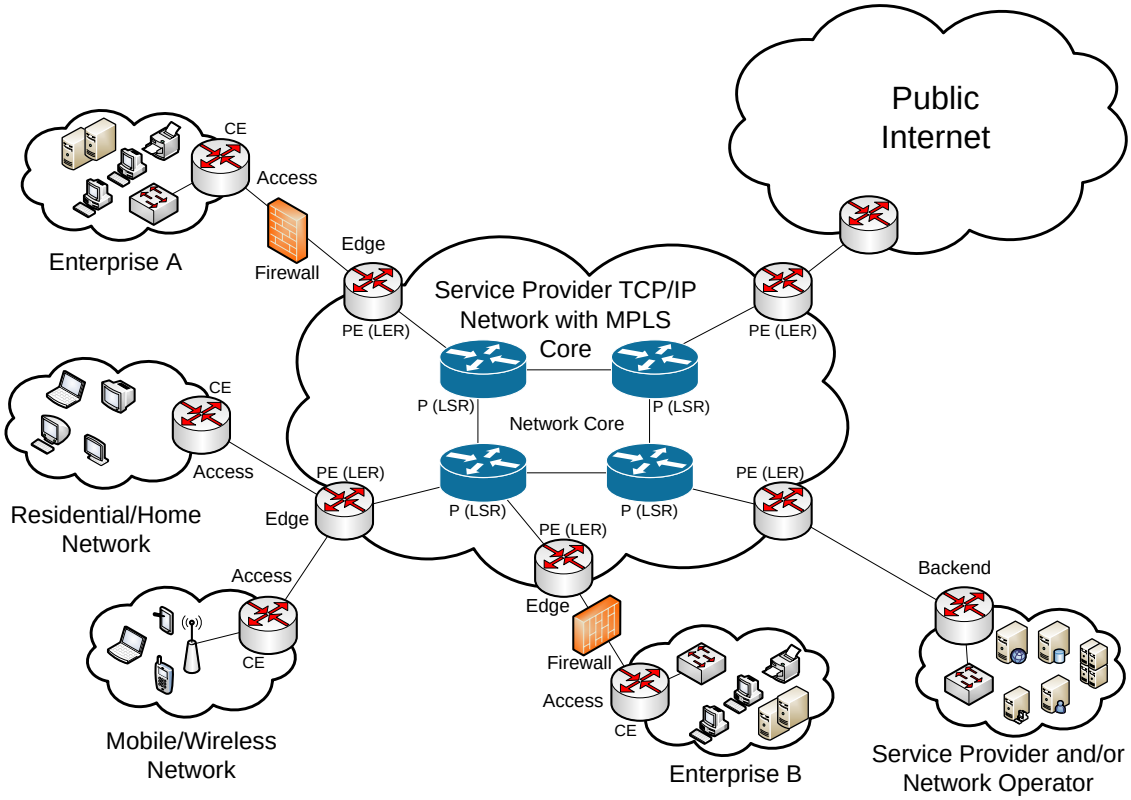


Figure 3-1: A typical single domain service provider network.

3-1 Data Plane Considerations

The primary data plane consideration for implementing the proposed NaaS architecture in service provider networks is inspired by the network fabric design proposed by Casado et al. [1], which is depicted in Figure 1-1 (Chapter 1). According to Casado et al., implementing the network core as a simple network fabric (fast and cheap packet transportation) while pushing the complexity (complex network functions and operations) towards the network edge will facilitate flexible and independent evolution of both network core and edge, which results in the decoupling of network core from its edge. In essence, this network fabric design was proposed to apply the insights underlying MPLS to OpenFlow based SDN architecture. For the proposed implementation strategy, this concept of network fabric design is slightly modified to comply with the proposed evolutionary approach, which involves incremental deployment of SDN like disruptive technologies in service provider networks.

As it is known, MPLS is already widely implemented in service provider networks along with TCP/IP, which is depicted in Figure 3-1. Thus, the primary data plane consideration involves either replacing or updating the edge devices in service provider networks with OpenFlow-enabled network devices while having its network core unchanged, which results in legacy MPLS label switch routers (LSRs) as provider routers while OpenFlow-enabled MPLS label edge routers (LERs) as provider edge routers. Furthermore, the legacy network core involves proactive installation of full-mesh static LSPs instead of dynamic LSPs built through signaling and routing protocols as in today's networks. As a result, the proposed network edge

needs to perform all the complex network operations and functions on the incoming TCP/IP based network traffic and steer it across the simple and static legacy network core through label switching. Finally, the proposed implementation strategy will enable an intelligent and complex network edge that is decoupled from a simple and static legacy network core, which involves the MPLS based network fabric design with dumb pipes, in service provider networks. In Figure 3-2, the data plane considerations for implementing the proposed evolutionary approach in a typical single domain service provider network, which is depicted in Figure 3-1, is shown.

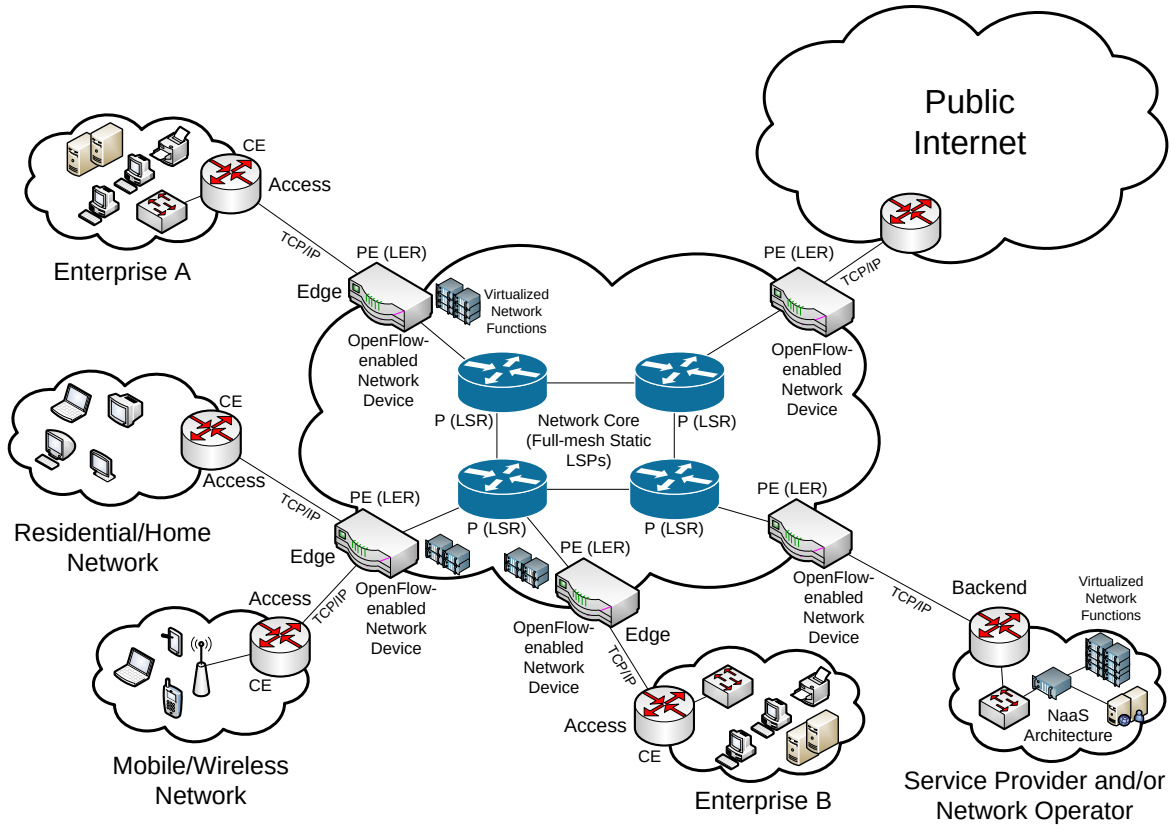


Figure 3-2: The data plane considerations for implementing the proposed evolutionary approach in a typical single domain service provider network, which is depicted in Figure 3-1.

As seen in Figure 3-2, at the provider edge (PE), legacy MPLS LERs are replaced or updated with OpenFlow-enabled MPLS LERs for flow-level granularity in network control and management. Furthermore, the unchanged legacy network core involves proactive installation of full-mesh static LSPs. Thus, this approach results in an intelligent and complex network edge that is decoupled from a simple and static legacy network core, which describes the MPLS based network fabric design with dumb pipes. Moreover, for network-wide visibility and monitoring, sFlow is used at the proposed network edge while SNMP is used at the legacy network core. Finally, virtualized network functions (NFV - VNFs) are implemented either as applications in the proposed NaaS architecture or as separate software components running on virtualized servers enabled with OpenFlow switches that are integrated into the proposed NaaS architecture at the proposed network edge.

OpenFlow, the de-facto standard for SDN, is chosen as the SDN implementation due to the level of generality, flow-level granularity, and vendor-neutrality it provides to the decoupled network control plane while being an enabler of innovation in networks [18, 19]. Moreover, latest versions of the OpenFlow protocol and their switch specifications support key MPLS operations such as label push, swap, and pop with flow-level granularity [4]. This approach is similar to Hampel et al. [28] in incremental deployment of OpenFlow based SDN at network edge, but instead of tunneling network traffic over IP from OpenFlow-enabled network edge devices through legacy network core it employs MPLS to enable fast and cheap packet transportation (network fabric design) at the legacy network core while facilitating the decoupling of network core from its edge.

For network-wide visibility and monitoring to perform closed-loop network control and management, sFlow [53] was proposed to be used as the network monitoring technology at the proposed network edge while SNMP [50] at the legacy network core. sFlow is chosen as the network edge monitoring technology because of its customizable packet sampling rate, which provides sufficient network visibility and information to perform complex network control and management operations at the proposed network edge. Furthermore, as sFlow is a push-based and hardware (e.g. ASIC) supported network monitoring technology it offloads costs and overheads on the underlying network devices while addressing the scalability and reliability concerns over OpenFlow based SDN solution through decoupling of centralized visibility from centralized control, which is in accordance with the concept of Devoflow proposed by Mogul et al [29]. Moreover, most of the commercially available OpenFlow-enabled network devices along with the Open vSwitch, the de-facto standard for open virtual switches, implementations [36, 37] support sFlow. In the legacy network core, SNMP is used as the network monitoring technology because of its widespread adoption in legacy network devices. Furthermore, SNMP provides device and interface statistics, which are sufficient for bandwidth and fault analysis to avoid congestions, service degradation, and packet losses at the simple and static legacy network core. Finally, both sFlow and SNMP are vendor-neutral and remote network monitoring technologies that are highly interoperable in open and heterogeneous network architectures, which is show in Table 2-5 (Chapter 2).

For implementing virtualized network functions (NFV - VNFs) in the proposed NaaS architecture, network functions in service provider networks are proposed to be incrementally decoupled from dedicated hardware and are implemented either as applications in the proposed NaaS architecture (SDN northbound applications as shown in [26]) or as separate software components running on virtualized servers enabled with OpenFlow switches that are integrated into the proposed NaaS architecture at the proposed network edge (complementary implementation of NFV with OpenFlow based SDN as proposed in [25, 27]).

3-2 Control and Management Plane Considerations

The control and management plane considerations of the proposed implementation approach primarily aim to provision simple abstractions of the underlying network resources, which involves enabling network virtualization through SDN and NFV, to the northbound NaaS based network service orchestration platform, which is depicted in Figure 2-1 (Chapter 2) and proposed in [2, 20, 21]. The control and management plane considerations can be directly mapped with the stated data plane considerations. According to the data plane considerations, the

complex and intelligent network edge is decoupled from the simple and static legacy network core, which describes the MPLS based network fabric design with dumb pipes. Thus, the control and management plane considerations of the network core and edge are also decoupled from each other.

For the complex and intelligent network edge, the control and management plane implementation (considerations) involves an OpenFlow based SDN controller for dynamic and flow-level traffic control and MPLS based traffic steering across the network, a sFlow based network analyzer for flow-level traffic monitoring, and a network configuration system that configures the underlying resources at the proposed network edge, which involves OpenFlow-enabled MPLS LERs and virtualized servers enabled with OpenFlow switches hosting the decoupled network functions. Furthermore, for NaaS based network service orchestration and closed-loop network control, OpenFlow based SDN controller and sFlow based network analyzer must expose their open and fully-programmable northbound APIs. Optionally, the network configuration system can also expose its open and fully-programmable northbound APIs. Nevertheless, it can employ any type of configuration protocol and method (e.g. CLI, Web service based, NETCONF, and OVSDB)[1] as the proposed network edge involves very few network devices and components that require frequent changes in network configurations.

For the simple and static legacy network core, one can reuse their legacy network management system while exposing SNMP based network analyzer's open and fully-programmable northbound APIs to the NaaS based network service orchestration platform for interface and device statistics. Optionally, the network configuration system can also expose its open and fully-programmable northbound APIs. Nevertheless, it can employ any type of configuration protocol and method (e.g. CLI, Web service based, SNMP, NETCONF)[1] as the proposed legacy network core involves only simple and static network devices and components.

3-3 Proposed Network-as-a-Service Architecture

The proposed Network-as-a-Service (NaaS) architecture involves implementing the stated data, control, and management plane considerations to provision simple abstractions of the underlying network resources, which involves enabling network virtualization through SDN and NFV, to a northbound NaaS based network service orchestration platform called the NaaS platform. The NaaS platform leverages the provisioned simple abstractions of the underlying network resources to implement network orchestration, police engines, functions, and services as fully-programmable software-based applications. These applications are in turn abstracted and exposed to the NaaS platform's northbound cloud-based service provisioning platform (e.g. OpenStack). These stated applications are provisioned and managed using the exposed abstractions of the OpenFlow based SDN controller (fully-programmable network control), sFlow (proposed network edge traffic flow monitoring) and SNMP (legacy network core interface monitoring) based network analyzers, and (optionally) network edge and core configuration systems. Furthermore, underlying network resources (both networks nodes and links) are abstracted as network graphs (graph theory and complex networks) to provision various innovative and customizable network functions and services such as routing, firewalling, path computations, and traffic engineering, which is in parallel with the proposed

[1] For the list of different types of network configuration technologies and their comparison, refer Table 2-6 (Chapter 2).

approaches in [22, 23, 24, 26]. Moreover, the network orchestration application in the NaaS platform will enable service chaining of virtualized network functions running on the underlying virtual servers, which is proposed in [25, 27]. Finally, this type of fully-programmable and software-based network service orchestration platform (NaaS platform) enables innovation in provisioning and management of network services. The overall proposed Network-as-a-Service (NaaS) architecture is depicted and briefly explained in Figure 3-3.

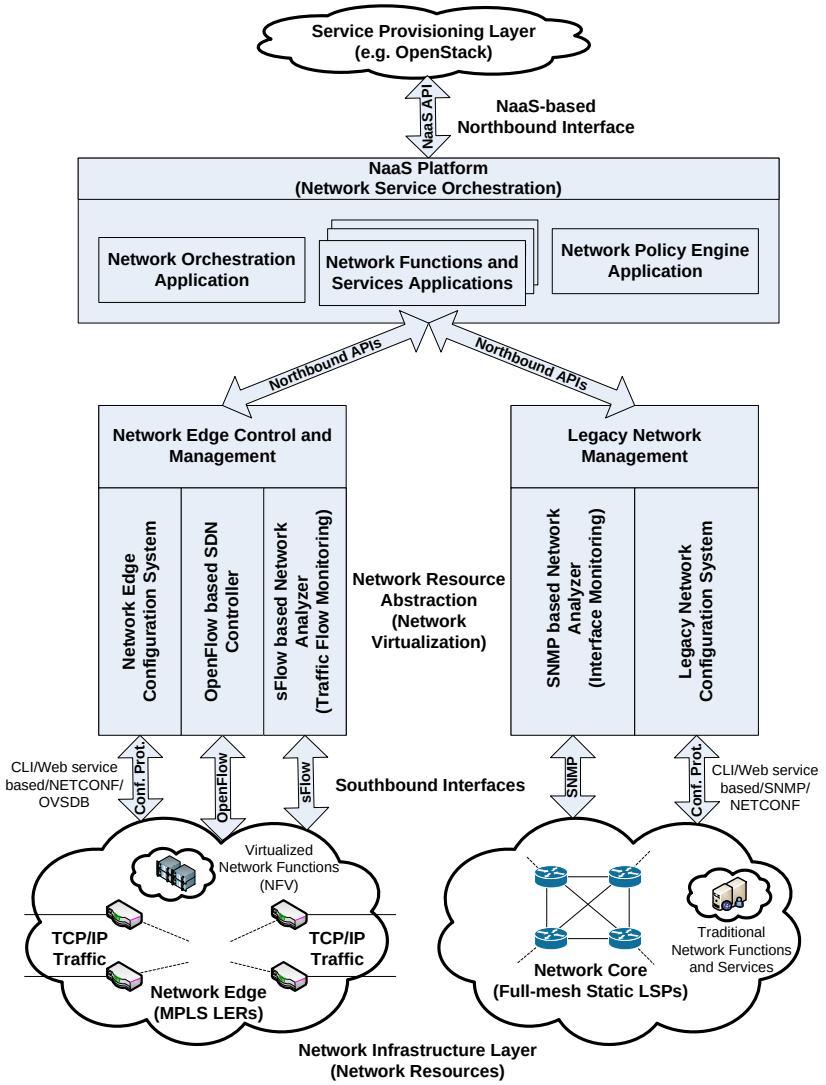


Figure 3-3: The overall proposed Network-as-a-Service (NaaS) architecture.

As seen in Figure 3-3, the data plane involves an intelligent and complex network edge that is decoupled from the simple and static legacy network core along with incrementally (gradually) deployed virtualized network functions (NFV - VNFs). Accordingly, the control and management plane of the network edge and core are also decoupled. The proposed network edge involves an OpenFlow based SDN controller for fully-programmable network control, a sFlow based network analyzer for traffic flow monitoring, and a network configuration system,

whereas the legacy network core involves a SNMP based network analyzer for interface monitoring and a legacy network configuration system. Furthermore, these control and management plane components expose their simple abstractions of the underlying network resources to the northbound NaaS based network service orchestration platform (NaaS platform) that implements network orchestration, police engines, functions, and services as fully-programmable software-based applications. Finally, these applications are in turn abstracted and exposed to the NaaS platform's northbound cloud-based service provisioning platform (e.g. OpenStack).

The proposed evolutionary approach and its implementation strategy addresses and solves the concerns over the involved technologies (SDN, NFV, TCP/IP, and MPLS). Firstly, the proposed NaaS architecture involves innovative provisioning and management of network services and functions along with a logically centralized and fully-programmable network control and management plane, which enables network virtualization through SDN and NFV. Secondly, the proposed NaaS architecture involves a simple and static network core, which describes MPLS based network fabric design with dumb pipes, with pre-installed full-mesh LSPs on vendor-neutral legacy MPLS LSRs, which results in no (in-band) signaling and control protocols and their corresponding overhead. Thus, these two stated features of the proposed NaaS architecture largely address and solve some of the major concerns over the existing network technologies (TCP/IP and MPLS) in service provider networks in terms of their vendor lock-in, complexity, and inflexibility while de-ossifying such networks as it enables flexible, fully-programmable, and abstracted network virtualization without any underlying (in-band) complex signaling and control protocols and their corresponding overhead. Thirdly, the proposed NaaS architecture involves incremental deployment of SDN and NFV technologies at the network edge while co-existing with the existing network technologies, which involves co-existence with TCP/IP at the provider-customer edge and rest of the public internet while with MPLS at the provider edge-core of service provider networks. Thus, this stated feature solves the interoperability and disruptive nature concerns over SDN and NFV technologies. Lastly, the centralized network control (SDN controller) is decoupled from the centralized network visibility and monitoring (sFlow and SNMP network analyzers) to enable much more efficient and effective closed-loop network control and management. Thus, this stated feature solves the scalability and reliability concerns over SDN technology. Moreover, SDN is the chief enabler for NFV in the proposed NaaS architecture. Thus, the concerns over NFV are directly related to those of OpenFlow based SDN solution in such complementary implementations.

Most importantly, the proposed MPLS based network fabric design with dumb pipes decouples the complex and intelligent network edge from the simple and static network core, which in turn will facilitate flexible and independent evolution of both the network core and edge in service provider networks. In conclusion, the proposed evolutionary approach realizes the major benefits of network virtualization such as vendor-neutrality, simplicity, and flexibility while successfully addressing the concerns over SDN and NFV technologies in terms of scalability, reliability, interoperability, and disruptive nature in the proposed NaaS architecture. In other words, it enables innovation in network service provisioning and management while facilitating flexible and independent evolution of both the network core and edge.

Chapter 4

Proof of Concept Implementation on a Physical Network Testbed

In this chapter, the proof of concept (PoC) implementation of the proposed NaaS architecture on a physical network testbed is demonstrated along with the innovative provisioning and management of basic network connectivity services over it. Accordingly, at first, the Proof of Concept design of the proposed NaaS architecture (section 4-1) followed by its implementation on a physical network testbed (section 4-2) are presented and discussed. Finally, this chapter concludes by presenting and discussing the innovative provisioning and management of basic network connectivity services over the PoC implementation (section 4-3).

4-1 Proof of Concept Design

The Proof of Concept (PoC) design of the proposed NaaS architecture is depicted in Figure 4-1. The overall proposed NaaS architecture is depicted in Figure 3-3 (Chapter 3).

As seen in Figure 4-1, when compared with the proposed NaaS architecture (Figure 3-3, Chapter 3), the PoC design additionally (optionally) supports sFlow based network core interface monitoring in its control and management plane. Furthermore, the PoC design does not involve any virtualized network functions (NFV - VNFs) being implemented on the underlying virtual servers at the network edge. Finally, the PoC design involves CLI as the NaaS based northbound service provisioning platform.

In this section, the PoC design components are presented and discussed. The PoC design components can be further explained by referring to Appendix A, which consists of additional information, graphical depictions, and code blocks relating to the PoC design components. Firstly, as per the data plane considerations in the proposed NaaS architecture, this PoC design was first tested and implemented on Mininet [45] network emulator before implementing it on a physical network testbed (section 4-2). Moreover, the PoC design does not involve any virtualized network functions (NFV - VNFs) being implemented on the underlying virtual

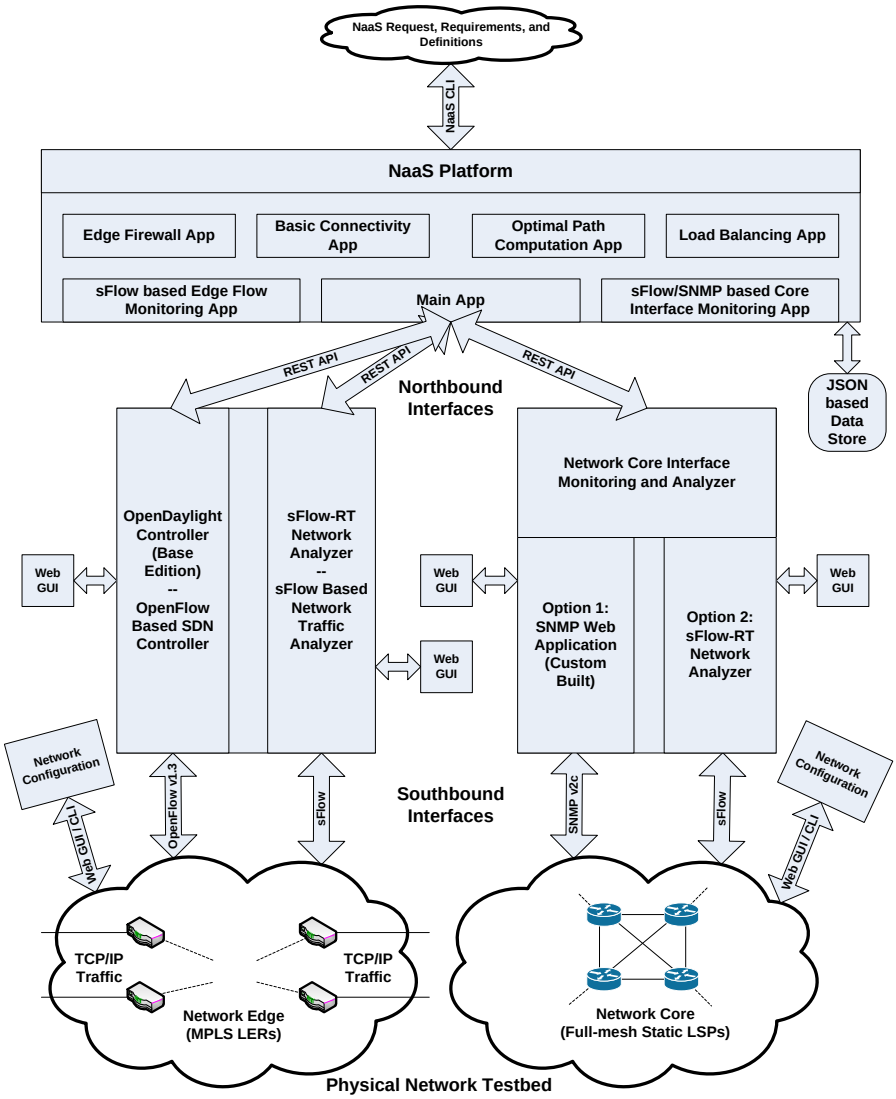


Figure 4-1: The Proof of Concept (PoC) design of the proposed NaaS architecture, which is depicted in Figure 3-3 (Chapter 3).

servers at the network edge, instead basic network connectivity functions and services are implemented as applications in the NaaS platform. Secondly, OpenDaylight Controller (Base edition) [11] is chosen as the OpenFlow based SDN controller because of its full-stack support for SDN and NFV, well documented northbound APIs (REST[1] APIs), and large developer community [43]. Moreover, the OpenDaylight Controller has built-in base network service functions such as host tracker, ARP handler, topology, stats, switch, and forwarding rules manager, which are also exposed as northbound REST APIs. In Figure 4-2, the OpenDaylight Base edition architecture, which is described in its user guide [11], is shown.

[1]REST (REpresentational State Transfer) is a simple and stateless web based architecture, which generally runs over HTTP (Hypertext Transfer Protocol). It is most commonly used for exposing application programming interfaces (APIs).

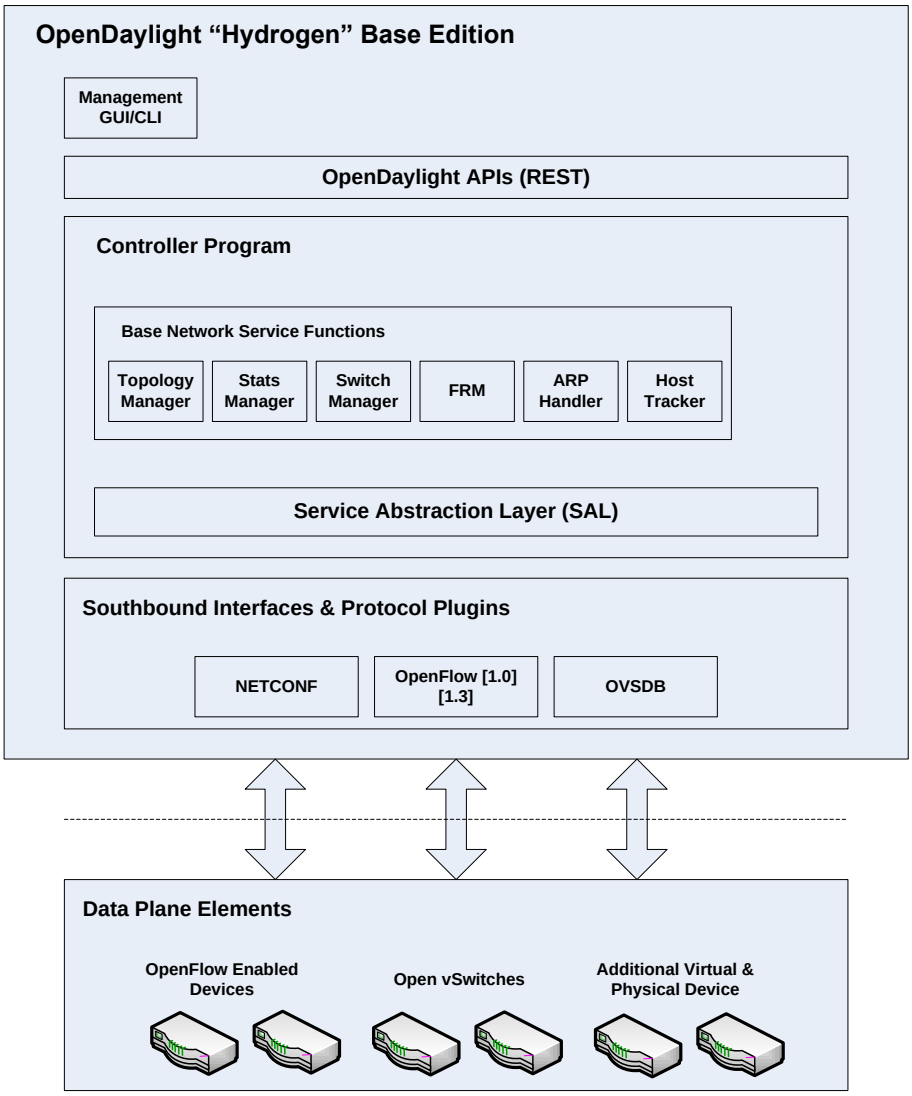


Figure 4-2: The OpenDaylight Base edition architecture, which is described in its user guide [11].

As seen in Figure 4-2, the OpenDaylight Base edition architecture supports both physical and virtual devices, especially OpenFlow-enabled devices and Open vSwitches. Moreover, it has NETCONF, OpenFlow versions 1.0 and 1.3, and OVSDB as its southbound interfaces and protocol plugins for programmable network control and management (configuration) of the underlying network devices. Furthermore, its controller program has an abstraction layer called Service Abstraction Layer (SAL) for underlying plugin management, capability abstractions, flow programming, and inventory, etc. Other components of the controller program such as built-in network service functions make use of these abstractions provisioned by the SAL. Finally, controller program components and capabilities including the built-in network service functions are exposed as REST APIs (northbound interface) to network applications orchestrations, services (e.g. OpenStack), and its management GUI (refer Appendix A-1) and CLI.

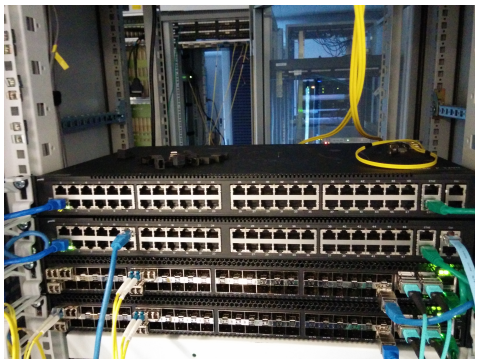
Thirdly, sFlow-RT network analyzer [58] is chosen as the sFlow based network traffic analyzer because of its support in real-time visibility to a wide range of SDN applications, well documented northbound APIs (REST APIs), and fully-customizable real-time traffic metrics, events, and thresholds (refer Appendix A-2). Furthermore, the developer of sFlow-RT network analyzer (InMon Corp.) maintains an active web blog that discusses SDN analytics and control using sFlow standard and sFlow-RT network analyzer [59]. Fourthly, a custom SNMP based web application was built with well documented northbound APIs (REST APIs) for network core interface monitoring, which exposes link failures and utilization events with customizable polling intervals and thresholds (refer Appendix A-3), using the Python modules Bottle [60] web framework and PySNMP [61] SNMP engine implementation. Optionally, sFlow-RT network analyzer can also be used for network core interface monitoring instead of the custom built SNMP web application. Thus, this approach facilitates incremental replacement of legacy network core routers with much more open and flexible network switches (e.g. Open vSwitch implementations enabled with OpenFlow and sFlow protocol), which enables flexible and independent evolution of the network core. Lastly, vendor-specific CLIs and web GUIs are used for network device configurations as the PoC design involves only static and proactive network configurations.

The NaaS platform involves an application called the Main App that acts as an abstraction layer to present all the underlying REST APIs as simple abstractions and function calls to other applications in the platform, it uses the Python module Requests [62] HTTP library for REST API calls (refer Appendix A-4). The NaaS platform's network service and function applications are discussed briefly in section 4-3. Moreover, the JSON (Java Script Object Notation) [63] based data store, which is a custom built NoSQL database, of the NaaS platform contains all the configuration details of the underlying PoC design components and network devices along with the ingress MPLS label to path bindings of all the pre-installed full-mesh static LSPs at the network core (refer Appendix A-5). Thus, the underlying network resources are abstracted as network graphs to the NaaS platform's network service and function applications. Fundamentally, the NaaS platform and the custom SNMP web application were built using the Python programming language in less than 4000 lines of code[2] while using JSON file format for data storage and data interchange through REST API calls (refer Appendix A-6). Finally, the PoC design involves CLI as the NaaS based northbound service provisioning platform (refer Appendix A-7). Nevertheless, any type of service provisioning platform (e.g. OpenStack) can be integrated with the PoC design as it involves open and fully-programmable network service abstractions.

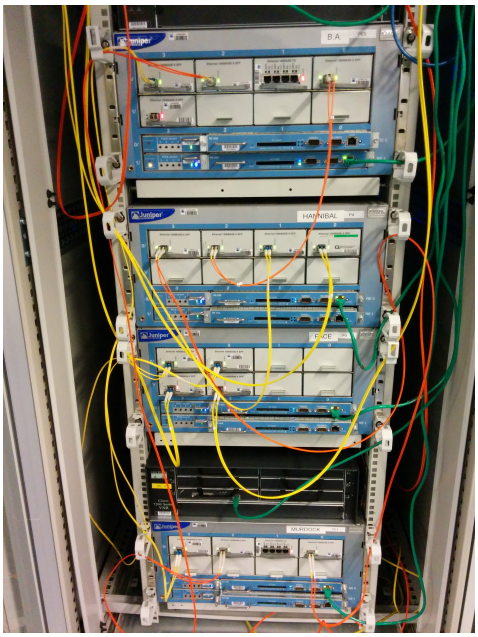
4-2 Implementation on a Physical Network Testbed

The physical network testbed consists of four Pica8 open switches, two 48 port 10 GbE P-3922 and two 48 port 1 GbE P-3290 switches [65], four Juniper M10i series routers [66], seven Linux based virtual machines (VMs) running on five different servers with VMware vSphere server virtualization operating system (OS) [67], and a Linux based remote PC. The four Pica8 open

[2]Upon publication of this thesis at TU Delft's institutional web repository and its related research paper (submitted) at IEEE NetSoft 2015 conference, the whole source code of the NaaS platform and the custom built SNMP web application will be made publicly available as a GitHub repository under the author's or his research group's GitHub profile [64].



(a) The four Pica8 open switches, two 48 port 10 GbE P-3922 and two 48 port 1 GbE P-3290 switches [65].



(b) The four Juniper M10i series routers [66].

Figure 4-3: The switches and routers that are used in the physical network testbed.

switches can either be used in a L2/L3 mode or Open vSwitch [36, 37] mode. Moreover, these Pica8 open switches are built as custom implementations on Application-specific Integrated Circuits (ASICs) with Linux network OS. Furthermore, the switches and routers (network devices) that are used in the physical network testbed are depicted in Figure 4-3. Finally, the physical network testbed can be further explained by referring to Appendix B, which consists of additional information on configuration, specification, and implementation of the physical network testbed components.

Fundamentally, the physical network testbed tries to emulate a service provider network being implemented with the proposed NaaS architecture. In the physical network testbed, a VM running on a server is used for running the OpenDaylight Controller and sFlow-RT network analyzer for edge flow control and monitoring, a VM running on a server is used for running the custom built SNMP web application and (optionally) the sFlow-RT network analyzer for core interface monitoring, five VMs running on three different servers are used as the five end-hosts, and a remote PC for hosting the NaaS platform. Moreover, one of the end-hosts (end-host 5) is configured to be in a different IP subnet compared to the other end-hosts in the physical network testbed to emulate a service provider network's connection to the Internet and other (external) network domains.

The physical network testbed involves two different implementations (testbed setups) called Testbed Setup A[3] and Testbed Setup B, which are depicted in Figure 4-4 and 4-5 respectively. At the network edge of the two testbed setups, the two P-3922 Pica8 open switches are used

[3]In Testbed Setup A, at the network edge, the Juniper legacy routers that are directly connected to the edge Pica8 open switches (OpenFlow-enabled switches) appear as end-hosts to them. However, by default, these legacy routers only respond to ARP requests from devices within their local network (IP subnet). Thus, interface MAC addresses of the edge Pica8 open switches are permanently published as static ARP table entries

as the OpenFlow-enabled MPLS LERs in the Open vSwitch mode with the OpenFlow switch specification version 1.3 [4] (refer Appendix B-1). At the network core, Testbed Setup A involves the four Juniper M10i series routers as the legacy (traditional) and static network core MPLS LSRs (refer Appendix B-2), whereas Testbed Setup B involves the two P-3290 Pica8 open switches in the Open vSwitch mode as the future and static network core MPLS LSRs (refer Appendix B-3). Thus, sFlow is used instead of SNMP for network core interface monitoring in Testbed Setup B as the Pica8 open switches support sFlow in their hardware ASICs. In essence, the physical network testbed is implemented as two different testbed setups to promote and demonstrate the support of the proposed NaaS architecture in enabling flexible evolution of both network core and edge, where Testbed Setup A consists of legacy and existing network core while Testbed Setup B consists of evolved and future network core. In other words, it realizes incremental replacement of legacy network core routers with much more open and flexible network switches (Open vSwitch implementations enabled with OpenFlow and sFlow protocol), which enables flexible and independent evolution of the network core.

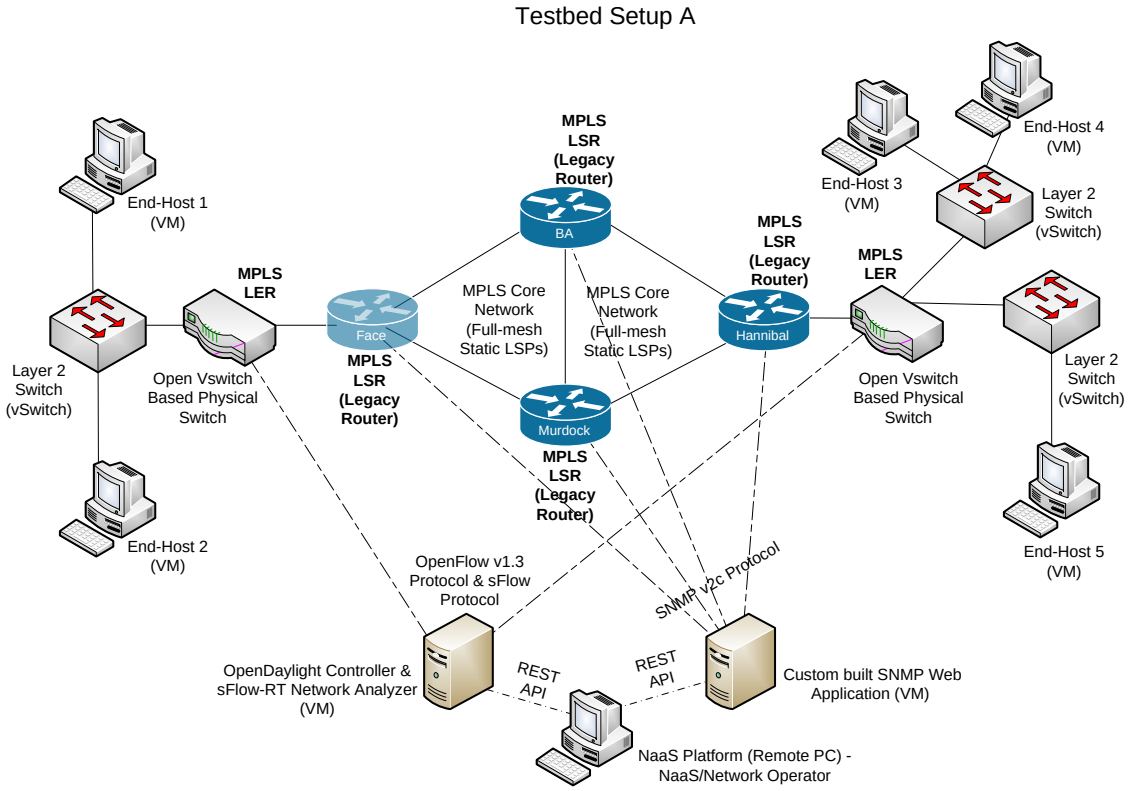


Figure 4-4: The Testbed Setup A[3] with Juniper M10i series legacy routers as the network core MPLS LSRs.

In both these testbed setups, network devices are connected with 1Gb fiber-optic cables while the end-host VMs are connected to the network through gigabit Ethernet (1Gb) cables. Furthermore, full-mesh static LSPs with penultimate hop popping (PHP) are pre-installed in the network core of both the testbed setups (refer Appendix B-2 and B-3). Thus, the OpenFlow-

in these legacy routers to enable communication between them. Alternatively, the OpenDaylight Controller can be tweaked to handle these type of involved ARP requests and replies.

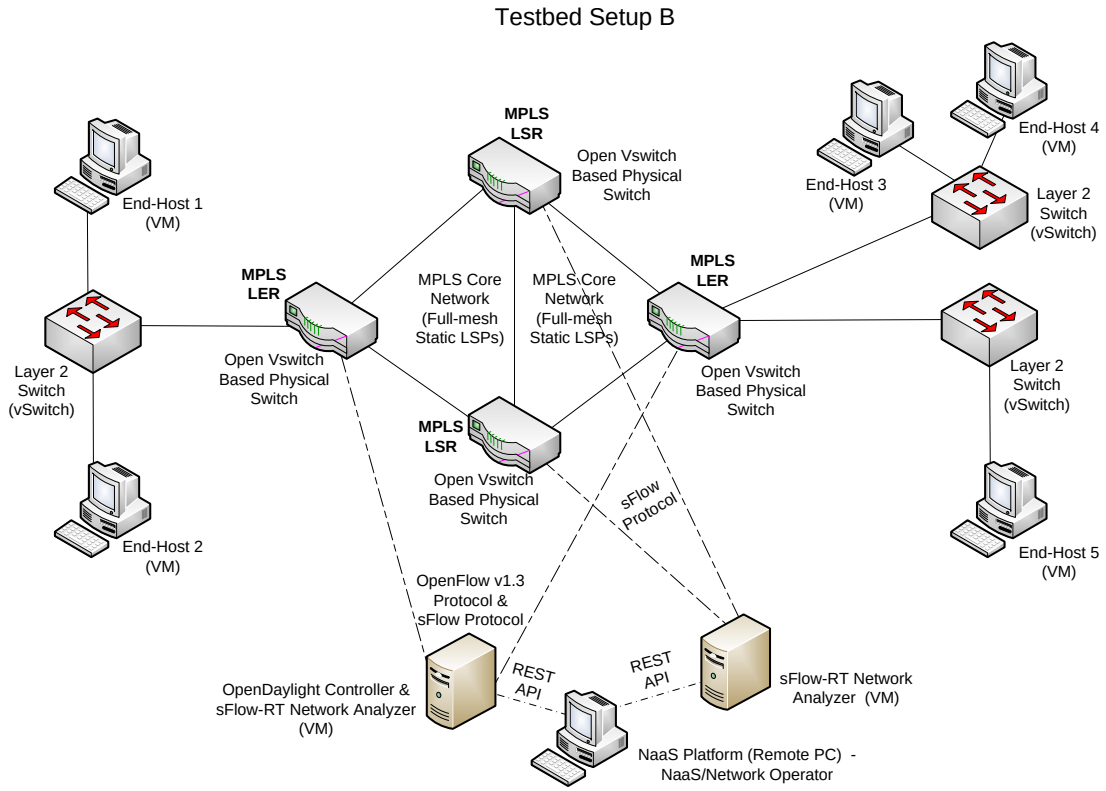


Figure 4-5: The Testbed Setup B with Pica8 P-3290 open switches as the network core MPLS LSRs.

enabled MPLS LERs just need to push the corresponding MPLS labels onto the incoming network traffic (packets) to steer it across the testbed network (refer Appendix B-1) while performing the associated basic network connectivity functions and services on them (basic connectivity, edge firewalling, and load balancing), which are discussed in section 4-3. Finally, the OpenFlow-enabled MPLS LERs need to replace the destination MAC addresses of the incoming network traffic (packets) with that of the next-hop router in Testbed Setup A (refer Figure 4-4) as the involved Juniper legacy MPLS LSRs only match MPLS traffic (packets) with their interface MAC addresses in the Layer 2 destination (packet) header fields. Thus, a JSON file format similar to one shown in Appendix A-6 is used to install such flows in the OpenFlow-enabled MPLS LERs of Testbed Setup A through the OpenDaylight Controller's REST API.

4-3 Basic Network Connectivity Services

The PoC design involves basic network connectivity functions and services (basic connectivity, edge firewalling, optimal path computation, and load balancing) as applications in the NaaS platform, which are innovatively provisioned and managed as basic network connectivity services through the NaaS based northbound service provisioning platform (CLI in the PoC design, refer Appendix A-7). Accordingly, the key applications in the NaaS platform that

enable these basic network connectivity services are first presented and discussed. Later, the innovative provisioning and management of the involved basic network connectivity services is presented and discussed. Moreover, these involved basic network connectivity services can be further explained by referring to Appendix C, which consists of additional information on configuration and management of them and their key enabling components (NaaS platform's applications).

Firstly, the NaaS platform involves an application called Main App that acts as an abstraction layer to present all the underlying REST APIs as simple abstractions and function calls to other applications in the platform (refer Appendix A-4). Thus, the underlying network resources are abstracted as network graphs (graph theory and complex networks) to the NaaS platform's network service and function applications.

Secondly, the NaaS platform has an application called sFlow based Edge Flow Monitoring App that configures the underlying sFlow-RT network analyzer through its Main App abstractions for customizable traffic flow monitoring at the network edge, which involves customizable definitions of traffic flow name (unique identifier), keys (packet headers), value (frames or bytes), filters (e.g. address groups), and thresholds (minimum flow value to trigger a flow event in the underlying sFlow-RT network analyzer). Moreover, this can be further explained by referring to Appendix C-1.

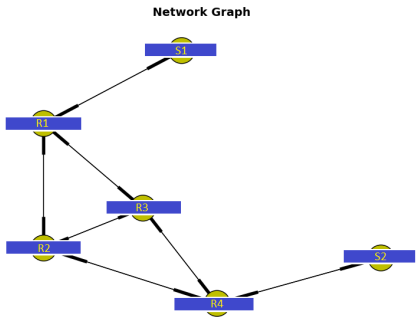
Thirdly, the NaaS platform has an application called sFlow/SNMP based Core Interface Monitoring App that configures the underlying SNMP web application or sFlow-RT network analyzer (as per the type of testbed setup, which is either SNMP web application as in Testbed Setup A - refer Figure 4-4 - or sFlow-RT network analyzer as in Testbed Setup B - refer Figure 4-5) through its Main App abstractions to trigger link failures and customizable high interface utilization events at the network core, which involves customizable definition of interface utilization threshold as percentage of the total available link bandwidth. Moreover, this can be further explained by referring to Appendix C-2.

Lastly, the NaaS platform has an application called Optimal Path Computation App that performs programmable optimal path computations by first constructing a network connectivity matrix, adjacency matrix with programmable link weights, and then running the Dijkstra's shortest path algorithm over it. This optimal path computation application uses the Python based software package NetworkX [68] for the involved graph theory and complex networks related computations. Moreover, it uses the Python 2D plotting library matplotlib [69] for provisioning graphical visualization of the optimal path computations. This can be further explained by referring to Appendix C-3. Fundamentally, this optimal path computation involves multi-constraint based QoS routing with only multiplicative QoS metrics and measures (e.g. link failures and threshold based high utilization events) [70]. An example network connectivity matrix, adjacency matrix with unity link weights, along with its graphical visualization is depicted in Figure 4-6, which represent Testbed Setup A as depicted in Figure 4-4.

The PoC design provisions and manages three basic network connectivity services, which are basic connectivity, load balancing, and edge firewalling. These services are provisioned and managed by orchestrating the stated key applications in the NaaS platform. Fundamentally, these network connectivity services are highly customizable, programmable, and reusable with several instances of them running at any given time. Furthermore, the innovative provisioning and management of the involved basic network connectivity services is presented and discussed

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}$$

(a) Network connectivity matrix.



(b) Graphical visualization.

Figure 4-6: An example network connectivity matrix, adjacency matrix with unity link weights, along with its graphical visualization, which represent Testbed Setup A as depicted in Figure 4-4.

in the following subsections (basic connectivity service in subsection 4-3-1, load balancing service in subsection 4-3-2, and edge firewalling service in subsection 4-3-3). Finally, the basic connectivity service and edge firewalling service are further explained and validated through an experimental evaluation in Chapter 5.

4-3-1 Basic Connectivity Service

In basic connectivity service, the Basic connectivity App in the NaaS platform continuously queries the OpenDaylight Controller through its Main App abstractions for the list of detected end-hosts. Upon detection of a new end host, it calls the Optimal Path Computation App for computing the shortest paths across the network to reach the new detected end-host and then calls the OpenDaylight Controller via its Main App abstractions to install the corresponding ingress MPLS push label flows for those paths (ingress MPLS label to static LSP path bindings, refer Appendix A-5) in the involved underlying OpenFlow-enabled network edge switches. In Figure 4-7, a high level flow diagram of the basic connectivity service is shown.

4-3-2 Load Balancing Service

In load balancing service, the Load Balancing App in the NaaS platform initially configures the underlying network core interface monitoring analyzer for detecting link failures and high bandwidth utilization events at the network core by calling the SNMP/sFlow based Core Interface Monitoring App, and then it regularly queries the analyzer through its Main App abstractions for those events. Upon detection of either a link failure event or a high utilization event, it calls the Optimal Path Computation App for computing the optimal paths by updating the network connectivity matrix with new link weights, '0' for link failure and a large number 'n' greater than the length of the matrix for high interface utilizations, and then running the Dijkstra's shortest path algorithm over it. Later, it steers the involved network traffic to the computed optimal paths as per the network policy and the subscribed

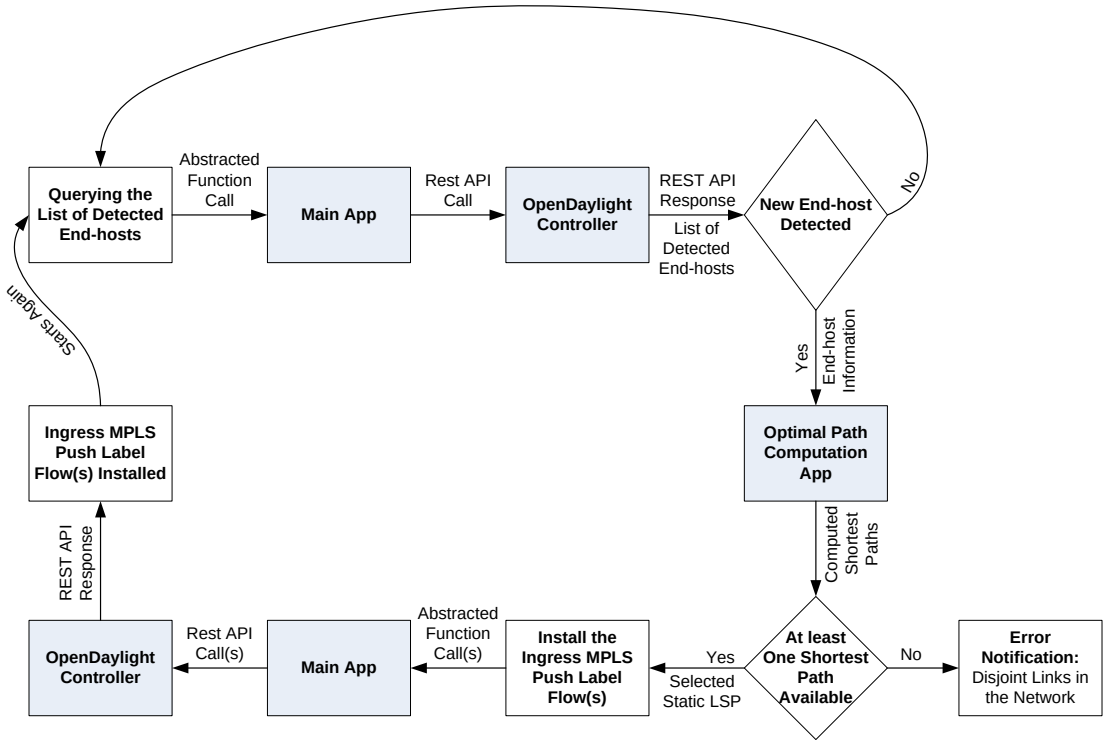


Figure 4-7: A high level flow diagram of the basic connectivity service.

service requirements by installing the corresponding ingress MPLS push label flows for those paths (ingress MPLS label to static LSP path bindings, refer Appendix A-5) in the involved underlying OpenFlow-enabled network edge switches through the OpenDaylight Controller's Main App abstractions with a priority higher than that of the existing flows in those switches. In Figure 4-8, a high level flow diagram of the load balancing service is shown.

4-3-3 Edge Firewalling Service

In edge firewalling service, the Edge Firewall App in the NaaS platform initially configures the underlying sFlow based network edge traffic analyzer (sFlow-RT network analyzer) for monitoring and detecting un-trusted traffic and security vulnerability events with flow-level granularity at the network edge by calling the sFlow based Edge Flow Monitoring App, and then it regularly queries the analyzer through its Main App abstractions for those events. Upon detection of a security vulnerability event, it gathers the information about the attacker(s) from the analyzer and installs a drop action flow with the highest priority to block the attacker(s) traffic for certain time and after that time it deletes (un-blocks) those installed drop action flows in the involved underlying OpenFlow-enabled network edge switches through the OpenDaylight Controller's Main App abstractions. In Figure 4-9, a high level flow diagram of the edge firewalling service is shown.

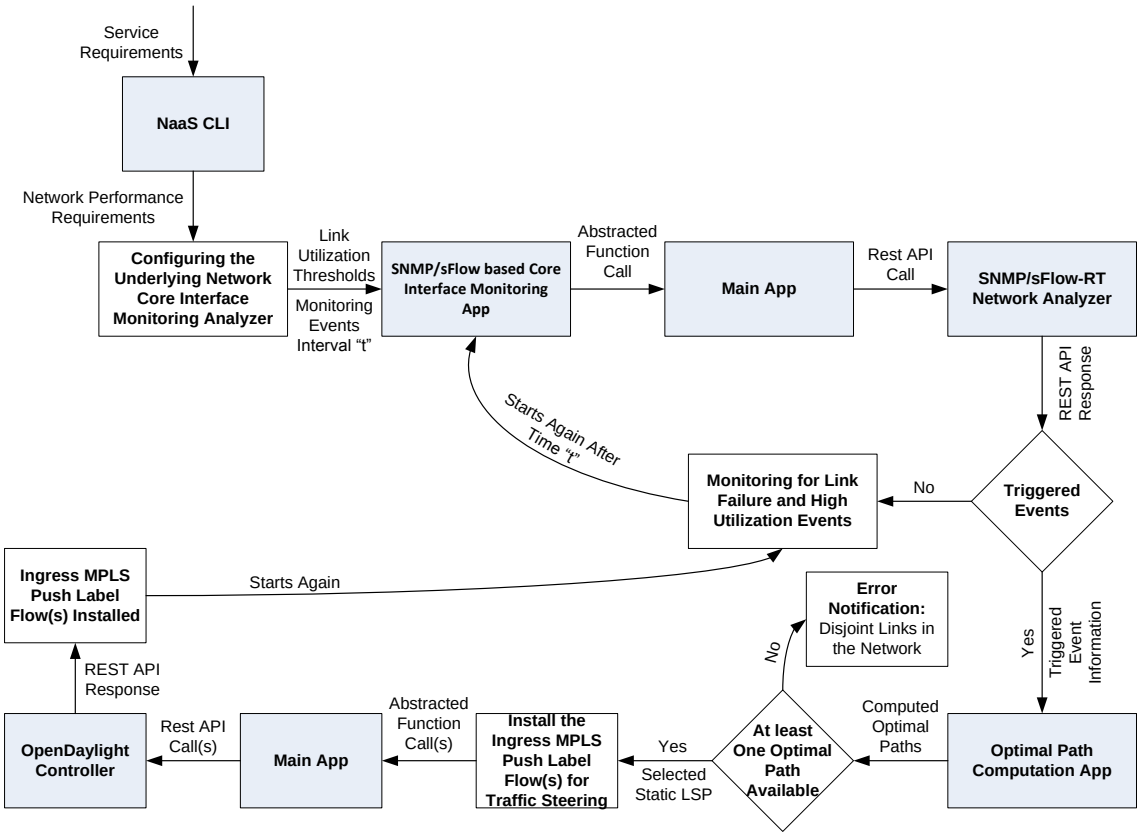


Figure 4-8: A high level flow diagram of the load balancing service.

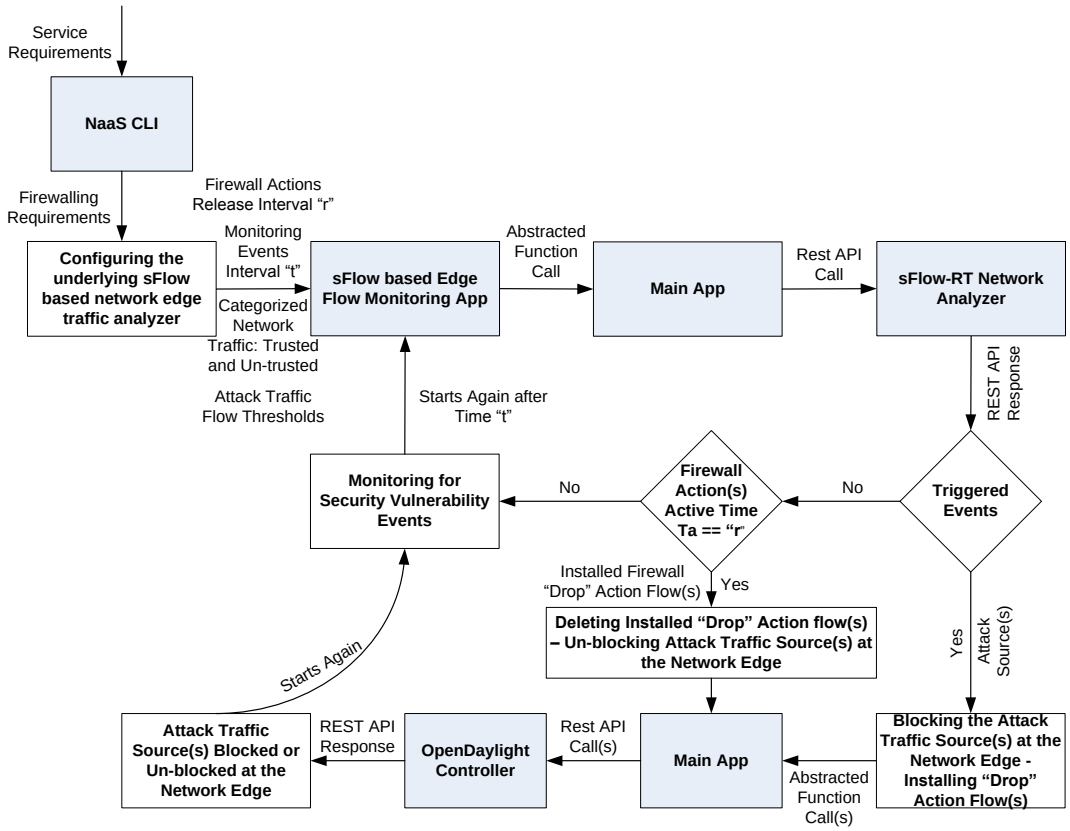


Figure 4-9: A high level flow diagram of the edge firewalling service.

Chapter 5

Experimental Performance Evaluation and Validation

In this Chapter, the experimental performance evaluation and validation of the proposed evolutionary approach are presented and discussed. The proposed evolutionary approach realizes the major benefits of network virtualization such as vendor-neutrality, simplicity, and flexibility while successfully addressing the concerns over SDN and NFV technologies in terms of scalability, reliability, interoperability, and disruptive nature in the proposed NaaS architecture. In other words, it enables innovation in network service provisioning and management while facilitating flexible and independent evolution of both the network core and edge. Thus, this experimental performance evaluation and validation of the proposed evolutionary approach involves both of the stated directions, which are enabling benefits and successfully addressing the involved concerns. Furthermore, this experimental performance evaluation is carried out on the PoC physical network testbed that is demonstrated in Chapter 4, which involves Testbed Setup A and Testbed Setup B as shown in Figures 4-4 and 4-5 respectively. Accordingly, this experimental performance evaluation focuses on two major characteristics of the proposed NaaS architecture and its PoC physical network testbed, which are performance analysis of the involved network control overhead (closed-loop OpenFlow based network control with sFlow and SNMP based network monitoring in section 5-1) and the involved basic network connectivity services (basic connectivity, edge firewalling, and load balancing in section 5-2). Moreover, this experimental performance evaluation can be further explained by referring to Appendix D, which consists of additional information on the involved experimental performance analysis data and plots. Finally, based on this experimental performance evaluation along with the obtained knowledge and experience throughout this research, the proposed evolutionary approach is validated (section 5-3).

For this experimental performance evaluation, network packets are first captured using the packet analyzer Wireshark v1.12 [71] at the involved VMs (controllers, analyzers, monitors, and end-hosts) during the performance evaluation experiments on the PoC physical network testbed. In essence, these performance evaluation experiments mostly involve active measurements (e.g. ping). Moreover, the latest Wireshark release v1.12 is used for packet capturing

because of its support to the OpenFlow protocol as it has a built-in OpenFlow packet dissector. Later, this captured packet information is studied and analyzed by plotting network performance plots using MATLAB [72] for data analysis and visualization. Fundamentally, as the captured packet information from Wireshark involves time-stamped data, this time series data is resampled into buckets of one second by summing over the numerical values of the captured data (number of frames and length of frames) in that second to plot network load in terms of frames per second, bytes per second, and bits per second over this resampled time.

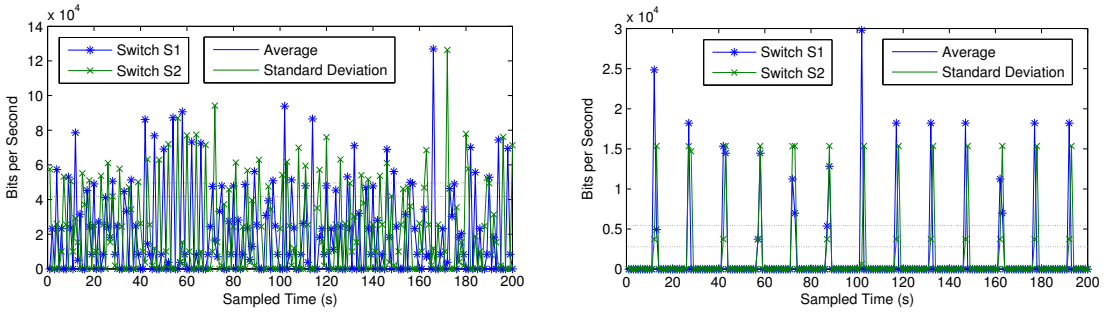
During the performance evaluation experiments on the PoC physical network testbed, it was found that Testbed Setup B involves slightly higher round-trip delay/time (RTD/RTT) compared to that of Testbed Setup A (by around 100 milliseconds). The main reason behind this is that the two open switches, 48 port 1 GbE Pica8 P-3290 open switches, in Testbed Setup B's network core support MPLS switching operations (push, pop, and swap) in their CPU instead of their hardware ASICs. Thus, the involved open switches in the proposed evolutionary approach must support MPLS switching operations in their hardware ASICs for much faster switching, avoidance of network bottlenecks, and to prevent unnecessary load on the switch CPUs.

5-1 Network Control Overhead Performance Analysis

For the performance analysis of the involved network control overhead in the PoC physical network testbed, the involved OpenFlow, sFlow, and SNMP protocol traffic in Testbed Setup A at the OpenDaylight Controller, sFlow-RT network analyzer, and custom built SNMP web application respectively are first studied and analyzed while provisioning and managing the PoC basic network connectivity services over it, which are discussed in section 4-3 of Chapter 4. Later, similar analysis is carried out in Testbed Setup B and compared with that of Testbed Setup A. In essence, Testbed Setup B only differs from Testbed Setup A in terms of its network core (open switches instead of legacy routers and sFlow instead of SNMP based network monitoring). However, the overall control overhead is found to be much higher in Testbed Setup A compared to that of Testbed Setup B (refer Appendix D-1).

At the network edge, the average of the total OpenFlow protocol traffic load at the OpenDaylight Controller to and from the two underlying open switches in Testbed Setup A is around 45 Kbps with a standard deviation of around 50 Kbps, whereas in Testbed Setup B the average value is around 1.5 Kbps with a standard deviation of around 4.5 Kbps. This huge difference can be explained by analyzing their involved traffic load samples, which are shown in Figure 5-1. A 200 second sample of the OpenFlow protocol traffic load measurement in both Testbed Setup A and B are shown in Sub-Figures 5-1a and 5-1b respectively.

In general, the OpenDaylight Controller gathers statistics from the underlying open switches every 15 seconds, which involves flow, group, meter, port, and table statistics request and reply OpenFlow packets, which can be clearly seen in Sub-Figure 5-1b. Furthermore, the only other major operation of the OpenDaylight controller in the PoC physical network testbed is that of installing and deleting MPLS push and static flows in the underlying open switches, which involve OpenFlow packets of size around 200 bytes (1600 bits) only. However, the reason for this high traffic load in Testbed Setup A is due to the involved (configured) link



(a) Testbed Setup A. **(b)** Testbed Setup B.

Figure 5-1: The OpenFlow protocol traffic load samples at the OpenDaylight Controller in the PoC physical network testbed.

state protocol, OSPF (Open Shortest Path First) protocol, in its legacy network core, for which the legacy routers send out multicast OSPF "Hello" request packets every 8 seconds. These edge open switches upon receiving those packets send them to the OpenDaylight Controller via the OpenFlow protocol for corresponding handling and action, which in turn sends them back to the underlying open switches as ARP requests (multicast packets). In this way they cause high traffic load in the network and at the OpenDaylight Controller. In Figure 5-2, the Wireshark capture of a multicast OSPF "Hello" received by the OpenDaylight Controller via the OpenFlow protocol in Testbed Setup A is shown.

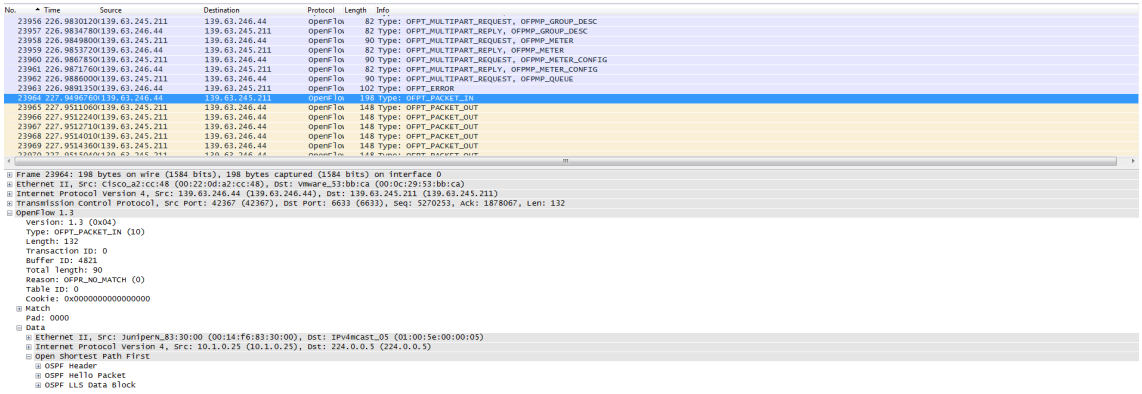


Figure 5-2: The Wireshark capture of a multicast OSPF "Hello" received by the OpenDaylight Controller via the OpenFlow protocol in Testbed Setup A.

Similarly, the end-host VMs are installed with a LLDP (Link Layer Discovery Protocol) implementation daemon called lldpd [73], which sends out multicast LLDP requests every 30 seconds and similarly induces high traffic load in the network and at the OpenDaylight Controller as that of the OSPF requests. In Figure 5-3, the Wireshark capture of a multicast LLDP request received by the OpenDaylight Controller via the OpenFlow protocol in Testbed Setup A is shown.

The sFlow based network edge traffic monitoring involves push-based exporting of sampled packets (at a pre-configured sampling rate) and the interface counters (at a pre-configured polling rate) to the sFlow-RT network analyzer. Thus, the sFlow protocol traffic in Testbed Setup A is also slightly over loaded with this (in-band) additional control overhead. The

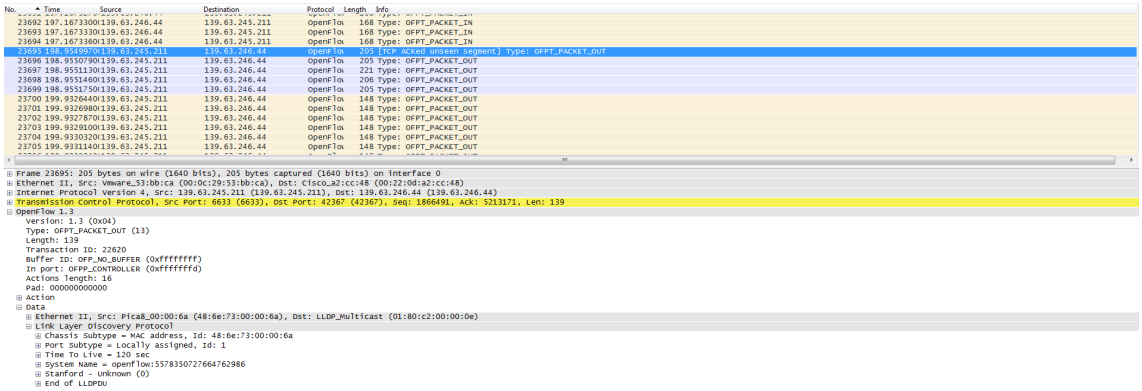


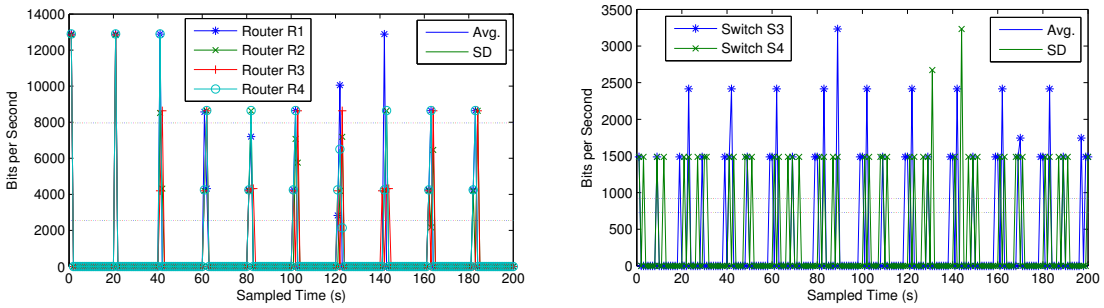
Figure 5-3: The Wireshark capture of a multicast LLDP request received by the OpenDaylight Controller via the OpenFlow protocol in Testbed Setup A.

sFlow protocol traffic is further explained below while discussing sFlow based network core interface monitoring in Testbed Setup B (refer Figure 5-4b). Nevertheless, both LLDP and OSPF are additional features to the PoC physical network testbed and can be removed or handled effectively by the OpenDaylight Controller to avoid this (in-band) additional control overhead. Upon mitigation of OSPF control overhead in Testbed Setup A, it will result in control overhead similar to that of Testbed Setup B, which is depicted in Sub-Figure 5-1b. In essence, Testbed Setup B does not involve any in-band signalling and control protocols and their corresponding overhead in its network as it involves only the open switches in both its network edge and core. However, Testbed Setup B still involves LLDP control overhead as LLDP is implemented in the end-host VMs of the physical network testbed. Thus, upon on removing LLDP implementations in the end-host VMs of the physical network testbed, in which case control overhead can be further reduced in both the testbed setups.

At the network core, the average of the total SNMP protocol traffic load at the custom built SNMP web application to and from the four underlying legacy routers in Testbed Setup A is around 2.5 Kbps with a standard deviation of around 6 Kbps, whereas in Testbed Setup B the average of the total sFlow protocol traffic load at the sFlow-RT network analyzer from the two underlying open switches is around 0.8 Kbps with a standard deviation of around 1 Kbps. This difference can be explained by analyzing their involved traffic load samples, which are shown in Figure 5-4. A 200 second sample of the SNMP[1] and sFlow[2] protocol traffic load measurement in the corresponding testbed setup are shown in Sub-Figures 5-4a and 5-4b respectively.

[1]In Testbed Setup A, at the network core, the custom built SNMP web application gathers only three interface counters from the Management Information Bases (MIBs), which are ifOperStatus, ifInOctetes, and ifOutOctetes in IF-MIB of SNMP MIB-2, in the underlying legacy routers at once every 20 seconds. The involved SNMP v2c request and reply packets in gathering the three interface counters are each of size around 90 bytes (720 bits).

[2]In Testbed Setup B, at the network core, the sFlow standard implementation in the underlying open switches is configured with a packet sampling rate of 1000 and interface counters polling rate of 20 seconds, which involves push based sFlow packet export to the sFlow-RT network analyzer. The involved sFlow packets in exporting the per interface counters and sampled packets are each of size around 186 bytes (1488 bits) and 218 bytes (1744 bits) respectively. Furthermore, two or more interface counters and sampled packets are sometimes exported as a single sFlow packet, in which case the total packet size is less than the sum of the individual packet sizes when sent separately.

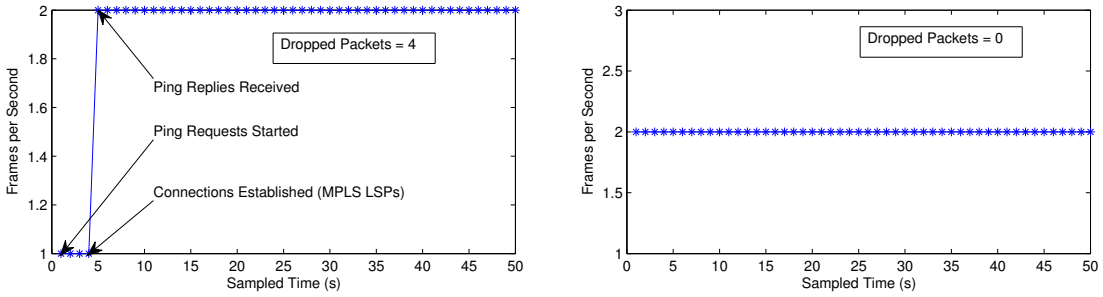


(a) SNMP protocol - Testbed Setup A[1]. **(b)** sFlow protocol - Testbed Setup B[2].

Figure 5-4: The network core interface monitoring traffic load samples in the PoC physical network testbed.

Although Testbed Setup B has less network core devices compared to that of Testbed Setup A, the control overhead at the network core is still logically higher in Testbed Setup A compared to that of Testbed Setup B. Firstly, SNMP based network monitoring in Testbed Setup A involves pull-based gathering of only three interface counters for detecting interface failure and high utilization events, whereas sFlow based network monitoring in Testbed Setup B involves push-based exporting of all the interface counters from the underlying open switches. Secondly, sFlow based network monitoring additionally involves flow sampling for much more visibility into the traffic at the network core. Lastly, SNMP based network monitoring gathers interface counters of the underlying legacy routers at once every 20 seconds, whereas sFlow based network monitoring involves exporting of interface counters per interface every 20 seconds. Nevertheless, SNMP protocol is still relevant for the proposed evolutionary approach due to its wide-spread and vendor-neutral implementation in almost all of the legacy network devices present in service provider networks. Moreover, sFlow standard (open switches) can be deployed incrementally in the network core to reduce the involved control overhead while enabling much better visibility into the network core. However, the open switches must support MPLS switching operations in their hardware ASICs for much faster switching, avoidance of network bottlenecks, and to prevent unnecessary load on the switch CPUs.

On the whole, the proposed evolutionary approach performs much better compared to the legacy solutions in terms of predictability, transparency, and customizability of the involved network control overhead as it avoids the usage of complex (in-band) legacy signaling and control protocols and their corresponding overhead. Furthermore, decoupling of network core from edge along with the separation of network control and monitoring greatly reduces load on OpenFlow based SDN controller and the involved network edge and core analyzers (closed-loop OpenFlow based network control with sFlow and SNMP based network monitoring). Moreover, static MPLS core with intelligent edge further reduces the load on OpenFlow based SDN controller as it involves relatively less devices and traffic flow rules to manage. Finally, it was found out that the incremental deployment of open switches in the network core will greatly reduce the network control overhead while enabling much better visibility into the network core (Testbed Setup B). Given that the open switches support MPLS switching operations in their hardware ASICs for much faster switching, avoidance of network bottlenecks, and to prevent unnecessary load on the switch CPUs.



(a) Reactive connection establishment[3]. **(b)** Proactive connection establishment[4].

Figure 5-5: The basic connectivity service performance analysis in the PoC physical network testbed.

5-2 Basic Network Connectivity Services Performance Analysis

For the performance analysis of the involved basic network connectivity services (basic connectivity, edge firewalling, and load balancing) in the PoC physical network testbed, which are discussed in section 4-3 of Chapter 4, the corresponding performance evaluation experiments that are presented and discussed in this section are carried out on Testbed Setup A (refer Appendix D-2). Nevertheless, similar results are achieved in Testbed Setup B. However, as stated, it was found that Testbed Setup B involves slightly higher round-trip delay/time (RTD/RTT) compared to that of Testbed Setup A (by around 100 milliseconds).

For the performance analysis of the involved basic connectivity service in the PoC physical network testbed, a ping operation is performed at end-host 1 to send ping requests to end-host 5 that is across the network in the PoC physical network testbed while capturing the ping traffic (ping requests and replies) at end-host 1 (VM). This ping operation was performed from the Ubuntu terminal of end-host 1 (VM) by using the ping option [74], which pings the destination host with Internet Control Message Protocol (ICMP) echo (ping) request packets. Each ICMP echo (ping) request and reply packet is of length 98 bytes (784 bits) and ICMP echo (ping) requests are sent to the destination at a rate of one frame per second (0.784 Kbps).

Based on this captured ping requests that are sent to and their corresponding replies received from end-host 5, the response time of the basic connectivity service in establishing connections (LSPs) between the two end-hosts that are across the network from each other is calculated along with the number of dropped packets (ping requests with no replies), which is shown in Figure 5-5. Moreover, this performance analysis is performed under two scenarios that are reactive connection establishment[3] and proactive connection establishment[4], which are shown in Sub-Figure 5-5a and 5-5b respectively.

As seen in Figure 5-5, reactive connection establishment involves around three seconds of response time to establish connections between the end-hosts therefore four ping requests

[3]In reactive connection establishment, the end-hosts are being connected for the first time to the network, in which case during the actual communication (data transfer) between the end-hosts they are first detected by the OpenDaylight Controller through their ARP requests and based on this detected end-host information the basic connectivity service establishes corresponding connections between the detected end-hosts (installs ingress MPLS push label flows for LSPs establishment in the network via the OpenDaylight Controller).

[4]In proactive connection establishment, the stated process in reactive connection establishment is done proactively before the actual communication (data transfer) between the end-hosts.

at start have no replies from end-host 5 (dropped packets), whereas proactive connection establishment neither involves any response time nor any dropped packets. Thus, proactive connection establishment avoids connection setup delays and dropping of packets.

For the performance analysis of the involved edge firewalling service in the PoC physical network testbed, a basic ping flood based denial-of-service (DoS) attack is performed on one of the end-hosts that is across the network in the PoC physical network testbed while capturing the attack traffic at the attack source and target end-host (VM). Later, this captured packet information is studied and analyzed to determine the total involved network load due to the attack along with the response time in mitigation of the attack by the edge firewalling service. This performance analysis leads to similar results in both of the testbed setups as the edge firewalling service involves operations only at the network edge.

For this attack, end-host 1 was chosen as the attack target and end-host 5 as the attack source in Testbed Setup A to mimic an external security threat as end-host 5 is configured to be in a different IP subnet compared to the other end-hosts in the physical network testbed. Accordingly, the edge firewalling service was configured to install an incoming traffic monitoring flow with a filter categorizing network traffic as (trusted) internal and (un-trusted) external based on their IP subnet addresses and a threshold of 1000 frames per second in the underlying sFlow-RT network analyzer to detect a DoS attack at the network edge in the two open switches of Testbed Setup A. Furthermore, the edge firewalling service upon detection of the DoS attack blocks the corresponding attacker traffic at all the involved network edge open switches in Testbed Setup A.

This basic ping flood based DoS attack was performed from the Ubuntu terminal of end-host 5 (VM) by using the flood ping option [74], which floods the destination host with Internet Control Message Protocol (ICMP) echo (ping) request packets at a maximum rate that is possible on the network. Each ICMP echo (ping) request and reply packet is of length 98 bytes (784 bits). The overall basic ping flood based DoS attack and its mitigation is depicted in Figure 5-6, which shows the total attack traffic load in terms of frames per second at the attack source and target.

This basic ping flood based DoS attack mitigation by the edge firewalling service in the PoC physical network testbed is further explained in Figure 5-7. As this attack involves ping flood requests and replies from the attack source and target respectively, their corresponding traffic loads at the attack source and target are represented in Sub-Figures 5-7a and 5-7b. As can be seen in Sub-Figure 5-7a, the DoS attack was started at around 64 second mark and upon detection of the DoS attack, reaching the 1000 frames per second threshold, the edge firewalling service mitigates the attack at the source, blocks the attacker traffic at the ingress network edge open switch, in less than two seconds. This response time of less than two seconds can be further seen in Sub-Figure 5-7b.

After the attacker traffic is blocked at the ingress network edge open switch, the edge firewalling service after time "t" releases the firewalling actions (unblocks attacker traffic). Accordingly, a similar attack is performed again by the attack source after its traffic is unblocked, and the edge firewalling service again unblocks it with similar performance level and response time (less than two seconds). This second attack is similarly explained and depicted in Figure 5-8.

In essence, a distributed denial-of-service (DDoS) attack from multiple attackers can also be performed on the PoC physical network testbed and the edge firewalling service would still

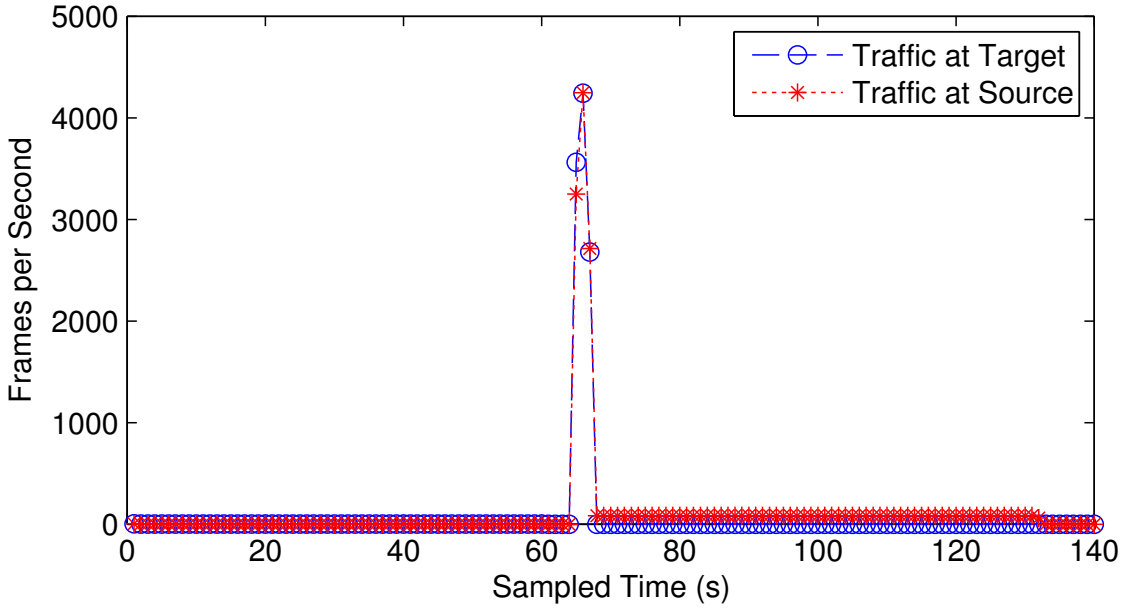
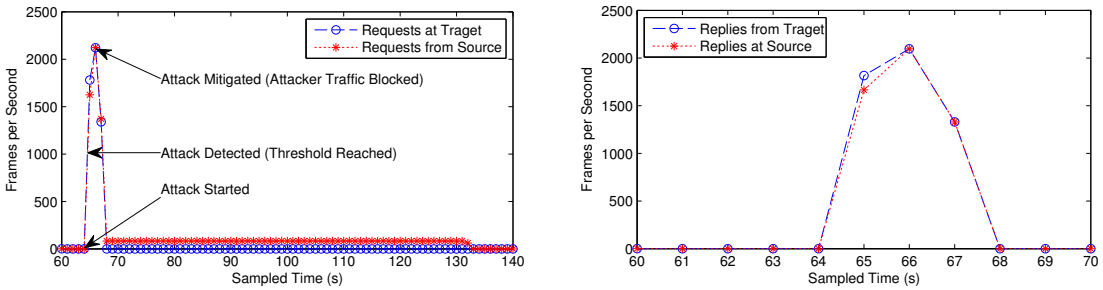


Figure 5-6: The overall basic ping flood based DoS attack and its mitigation.



(a) Ping flood requests.

(b) Ping flood replies.

Figure 5-7: Mitigation of a basic ping flood based DoS attack through the edge firewalling service in the PoC physical network testbed.

mitigate them at a similar performance level and a response time, as the involved process and mechanism in the edge firewalling service still remain the same for both the DoS and DDoS attacks.

On the whole, good performance levels and response times were determined for all the involved basic network connectivity services in the PoC physical network testbed. The basic connectivity service involves a response time of around three seconds in its reactive mode and a response time of zero in its proactive mode. The edge firewalling service involves a response time of less than two seconds. However, the load balancing service response time is completely dependent on the configured network core interface monitoring rate at the network analyzers (SNMP web application in Testbed Setup A and sFlow-RT network analyzer in Testbed Setup B). Nevertheless, real-time interface monitoring or configured monitoring traps in the underlying network core devices can result in a response time similar to that of edge firewalling service (less than two seconds). Thus, the performance levels and response

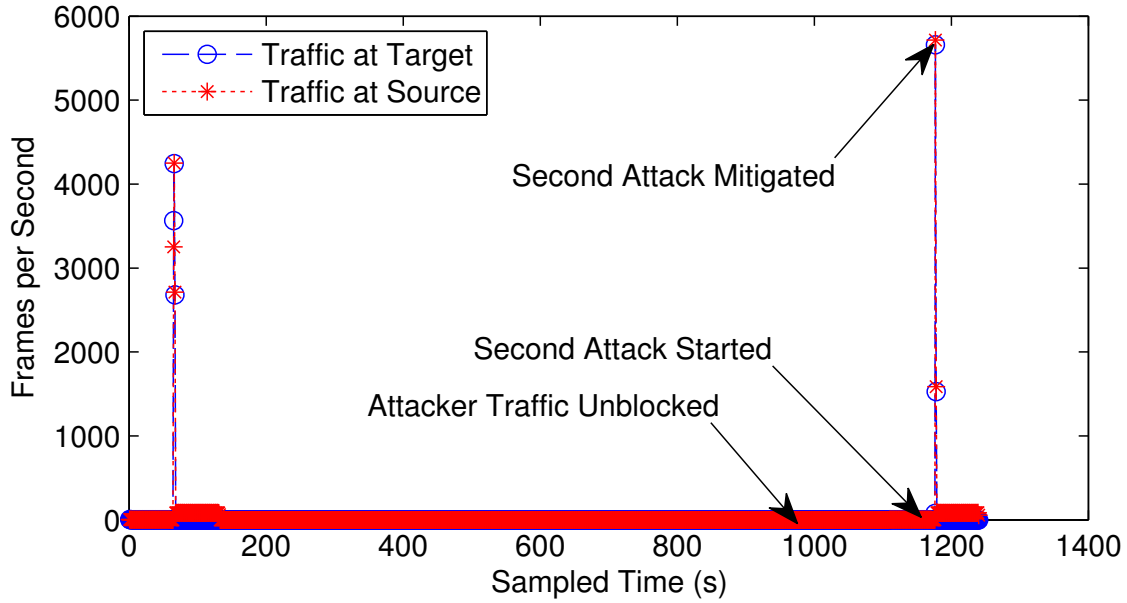


Figure 5-8: The second basic ping flood based DoS attack and its mitigation.

times of the involved basic network connectivity services in the PoC physical network testbed can be further improved by increasing the network monitoring accuracy and rate at the cost of network control overhead, which is a simple tradeoff to make. This enables performance customizability, transparency, and predictability of network services. Furthermore, these network connectivity services are highly customizable, programmable, and reusable with several instances of them running at any given time. Thus, the proposed evolutionary approach performs much better compared to the legacy solutions in terms of innovation in provisioning and management of network services.

5-3 Validation

In essence, the proposed evolutionary approach realizes the major benefits of network virtualization such as vendor-neutrality, simplicity, and flexibility while successfully addressing the concerns over SDN and NFV technologies in terms of scalability, reliability, interoperability, and disruptive nature in the proposed NaaS architecture. Thus, the proposed evolutionary approach and its implementation strategy addresses and solves the concerns over the involved technologies (SDN, NFV, TCP/IP, and MPLS) in the proposed NaaS architecture. In other words, it enables innovation in network service provisioning and management while facilitating flexible and independent evolution of both the network core and edge. For more information on the proposed evolutionary approach refer Chapter 3. This stated characteristics and benefits of the proposed evolutionary approach are independently validated as follows.

Innovative provisioning and management of network services

The proposed NaaS architecture involves innovative provisioning and management of network services and functions along with a logically centralized and fully-programmable network control and management plane, which enables network virtualization through SDN and NFV. Fundamentally, the underlying network resources are abstracted as network graphs (graph theory and complex networks) to the NaaS platform's network service and function applications.

In Chapter 4, the PoC design provisions and manages three basic network connectivity services, which are basic connectivity, load balancing, and edge firewalling. Fundamentally, these network connectivity services are highly customizable, programmable, and reusable with several instances of them running at any given time. This characteristic of the proposed evolutionary approach is successfully implemented and shown in the PoC physical network testbed.

In this chapter, during their experimental performance evaluation, good performance levels and response times were determined for all the involved basic network connectivity services in the PoC physical network testbed. The basic connectivity service involves a response time of around three seconds in its reactive mode and a response time of zero in its proactive mode. The edge firewalling service involves a response time of less than two seconds. However, the load balancing service response time is completely dependent on the configured network core interface monitoring rate at the network analyzers (SNMP web application in Testbed Setup A and sFlow-RT network analyzer in Testbed Setup B). Nevertheless, real-time interface monitoring or configured monitoring traps in the underlying network core devices can result in a response time similar to that of edge firewalling service (less than two seconds). Thus, the performance levels and response times of the involved basic network connectivity services in the PoC physical network testbed can be further improved by increasing the network monitoring accuracy and rate at the cost of network control overhead, which is a simple tradeoff to make. This enables performance customizability, transparency, and predictability of network services.

Flexible and independent evolution of both the network core and edge

Most importantly, the proposed MPLS based network fabric design with dumb pipes decouples the complex and intelligent network edge from the simple and static network core, which in turn will facilitate flexible and independent evolution of both the network core and edge in service provider networks.

In Chapter 4, the PoC physical network testbed involves two different implementations (testbed setups) called Testbed Setup A and Testbed Setup B, which are depicted in Figure 4-4 and 4-5 respectively. In essence, the physical network testbed is implemented as two different testbed setups to promote and demonstrate the support of the proposed NaaS architecture in enabling flexible evolution of both network core and edge, where Testbed Setup A consists of legacy and existing network core while Testbed Setup B consists of evolved and future network core. In other words, it realizes incremental replacement of legacy network core routers with much more open and flexible network switches (Open vSwitch implementations enabled with sFlow protocol), which enables flexible and independent evolution of the network core.

In this chapter, during the experimental performance evaluation, it was found that the incremental deployment of open switches in the network core will greatly reduce the network control overhead while enabling much better visibility into the network core (Testbed Setup B). Given that the open switches support MPLS switching operations in their hardware ASICs for much faster switching, avoidance of network bottlenecks, and to prevent unnecessary load on the switch CPUs.

Network vendor-neutrality, simplicity, and flexibility

The proposed NaaS architecture involves a simple and static network core, which describes MPLS based network fabric design with dumb pipes, with pre-installed full-mesh LSPs on vendor-neutral legacy MPLS LSRs, which results in no (in-band) signaling and control protocols and their corresponding overhead. Thus, these two stated features of the proposed NaaS architecture largely address and solve some of the major concerns over the existing network technologies (TCP/IP and MPLS) in service provider networks in terms of their vendor lock-in, complexity, and inflexibility while de-ossifying such networks as it enables flexible, fully-programmable, and abstracted network virtualization without any underlying (in-band) complex signaling and control protocols and their corresponding overhead.

In Chapter 4, this characteristic of the proposed evolutionary approach is successfully implemented and shown in the PoC physical network testbed. In this chapter, it was found that the proposed evolutionary approach performs much better compared to the legacy solutions in terms of predictability, transparency, and customizability of the involved network control overhead as it avoids the usage of complex (in-band) legacy signaling and control protocols and their corresponding overhead.

SDN and NFV scalability and reliability

The centralized network control (SDN controller) is decoupled from the centralized network visibility and monitoring (sFlow and SNMP network analyzers) to enable much more efficient and effective closed-loop network control and management. Thus, this stated feature solves the scalability and reliability concerns over SDN technology. Moreover, SDN is the chief enabler for NFV in the proposed NaaS architecture. Thus, the concerns over NFV are directly related to those of OpenFlow based SDN solution in such complementary implementations.

In Chapter 4, this characteristic of the proposed evolutionary approach is successfully implemented and shown on the PoC physical network testbed. In this chapter, it was found that the decoupling of network core from its edge along with the separation of network control and monitoring greatly reduces load on OpenFlow based SDN controller and the involved network edge and core analyzers (closed-loop OpenFlow based network control with sFlow and SNMP based network monitoring). Moreover, static MPLS core with intelligent edge further reduces the load on OpenFlow based SDN controller as it involves relatively less devices and traffic flow rules to manage. Finally, good performance levels and response times were determined for all the involved basic network connectivity services in the PoC physical network testbed for which this SDN and NFV implementations are the key enablers.

SDN and NFV interoperability and faster adoption

The proposed NaaS architecture involves incremental deployment of SDN and NFV technologies at the network edge while co-existing with the existing network technologies, which involves co-existence with TCP/IP at the provider-customer edge and rest of the public internet while with MPLS at the provider edge-core of service provider networks. Thus, this stated feature solves the interoperability and disruptive nature concerns over SDN and NFV technologies. In Chapter 4, this characteristic of the proposed evolutionary approach is successfully implemented and shown on the PoC physical network testbed.

Chapter 6

Conclusion and Future Work

6-1 Conclusion

More and more service providers and network operators are embracing the concept of cloud-based service models along with their chief enabler virtualization to address their major problem of network ossification coupled with a lack of innovation in provisioning and management of network services. However, it is still a challenge to logically combine a set of newly proposed virtualization enabling network technologies to realize cloud-based service models for service provider networks because of the involved concerns over these proposed technologies in terms of scalability, reliability, interoperability, and disruptive nature. Moreover, the related work and state-of-the-art research in the field of networking views the stated two challenges, cloud-based service models for networking and adoption concerns over the proposed virtualization enabling network technologies, as two different problems and lacks a holistic approach. In this thesis, an evolutionary approach to implementing the Network-as-a-Service (NaaS) cloud-based service model for service provider networks is proposed with Software-Defined Networking (SDN) and Network Function Virtualization (NFV) as its key virtualization enabling network technologies.

In essence, the proposed evolutionary approach realizes the major benefits of network virtualization such as vendor-neutrality, simplicity, and flexibility while successfully addressing the concerns over SDN and NFV technologies in terms of scalability, reliability, interoperability, and disruptive nature in the proposed NaaS architecture. Thus, the proposed evolutionary approach and its implementation strategy addresses and solves the concerns over the involved technologies (SDN, NFV, TCP/IP, and MPLS) in the proposed NaaS architecture. In other words, it enables innovation in network service provisioning and management while facilitating flexible and independent evolution of both the network core and edge.

Fundamentally, the proposed evolutionary approach, instead of revolutionizing the whole network architecture with the disruptive SDN and NFV technologies (instead of upgrading the whole network at once), involves an implementation strategy that facilitates an incremental deployment scenario through a highly complementary co-existence between these disruptive

technologies and the most prominent existing network technologies, which are TCP/IP and MPLS, in service provider networks. The proposed NaaS architecture involves an intelligent and complex network edge that is decoupled from the simple and static legacy network core, MPLS based network fabric design with dumb pipes, along with incrementally deployed virtualized network functions (NFV - VNFs).

Accordingly, the control and management plane of the network edge and core are also decoupled from each other and the underlying data-forwarding plane. In this regard, the proposed network edge involves an OpenFlow based SDN controller for fully-programmable network control, a sFlow based network analyzer for traffic flow monitoring, and a network configuration system, whereas the legacy network core involves a SNMP based network analyzer for interface monitoring and a legacy network configuration system. Furthermore, these control and management plane components expose their simple abstractions of the underlying network resources to the northbound NaaS based network service orchestration platform (NaaS platform) that implements network orchestration, police engines, functions, and services as fully-programmable software-based applications, which are in turn abstracted and exposed to the NaaS platform's northbound cloud-based service provisioning platform (e.g. OpenStack). Thus, the proposed evolutionary approach enables innovation in network service provisioning and management while facilitating flexible and independent evolution of both the network core and edge.

A proof of concept (PoC) implementation of the proposed NaaS architecture on a physical network testbed is demonstrated along with the innovative provisioning and management of basic network connectivity services over it. These basic network connectivity services involve the following virtualized network functions: basic connectivity, firewalling, optimal path computation, and load balancing. Fundamentally, these involved virtualized network functions use the basic concepts of graph theory and traffic engineering to provision their corresponding basic network connectivity services. Furthermore, these stated network connectivity services are highly customizable, programmable, and reusable with several instances of them running at any given time. Finally, the PoC physical network testbed involves two different implementations (testbed setups) called Testbed Setup A and Testbed Setup B. In essence, the physical network testbed is implemented as two different testbed setups to promote and demonstrate the support of the proposed NaaS architecture in enabling flexible evolution of both network core and edge, where Testbed Setup A consists of legacy and existing network core while Testbed Setup B consists of evolved and future network core. In other words, it realizes incremental replacement of legacy network core routers with much more open and flexible network switches (Open vSwitch implementations enabled with sFlow protocol), which enables flexible and independent evolution of the network core.

The proposed evolutionary approach is validated by the overall gained experience during this research and the experimental performance evaluation of the PoC physical network testbed in two relevant directions, performance analysis of the involved network control overhead and the involved basic network connectivity services. This performance evaluation yielded sufficient results, predictability, transparency, and customizability of the involved network control overhead and the involved basic network connectivity services, that successfully validated the proposed evolutionary approach as an enabler of innovation in network service provisioning and management while facilitating flexible and independent evolution of both the network core and edge.

The proposed evolutionary approach is validated in terms of innovative provisioning and management of network services, flexible and independent evolution of both the network core and edge, network vendor-neutrality, simplicity, and flexibility, SDN and NFV scalability, reliability, interoperability, and faster adoption. This validation results sufficiently proved that the proposed evolutionary approach makes great progress on all the stated characteristics and benefits. However, the proposed evolutionary approach and its implementation strategy still needs to be further improved to implement and validate it in real-world production networks and service provider networks. Nevertheless, its a good starting point in this direction as it enables innovation in network service provisioning and management while facilitating flexible and independent evolution of both the network core and edge.

6-2 Future Work

As stated, the proposed evolutionary approach and its implementation strategy still needs to be further improved to implement and validate it in real-world production networks and service provider networks. Thus, a few future work directions for the proposed evolutionary approach have been proposed and briefly explained as follows.

Adding intelligence to the network core

As the current implementation of the proposed evolutionary approach involves a simple and static core, MPLS based network fabric design with dumb pipes, with pre-installed full-mesh LSPs. However, such a strategy is not very feasible and efficient for networks with a large core as it becomes increasingly difficult to manage such large number of LSRs and their corresponding full-mesh static LSPs. Thus, this proposed future work direction involves adding decoupled and centralized intelligence to the static network core to make it much more scalable and reliable.

Support for fully programmable network configuration technologies

As the current implementation of the proposed evolutionary approach involves mostly static and pro-active network configurations because of its mostly static network components and their corresponding network resources (e.g. links). However, such a strategy is not very feasible and efficient for large networks as it becomes increasingly difficult to manage such large number of components and their corresponding resources. Thus, this proposed future work direction involves support for fully programmable network configuration technologies for much more network flexibility and simplicity,

Innovative provisioning and management of network services with SLA and QoS assurance

To meet network demands of today, the proposed evolutionary approach should be implemented and updated to innovatively provision and manage network services while continuously conforming with SLA and QoS requirements and demands.

Integration with cloud-based service provisioning platforms

As the current implementation of the proposed evolutionary approach involves CLI as the NaaS based northbound service provisioning layer. Thus, this proposed future work direction involves integration with cloud-based service provisioning platforms such as OpenStack for enabling convergence of networking and cloud computing.

Implementation and validation on a real-world production network

As the current implementation of the proposed evolutionary approach involves implementation and validation on a small-scale physical network testbed. Thus, this proposed future work direction involves implementing and validating the proposed NaaS architecture on a real-world production network and service provider network.

Appendix A

Proof of Concept Design Components

A-1 OpenDaylight Controller

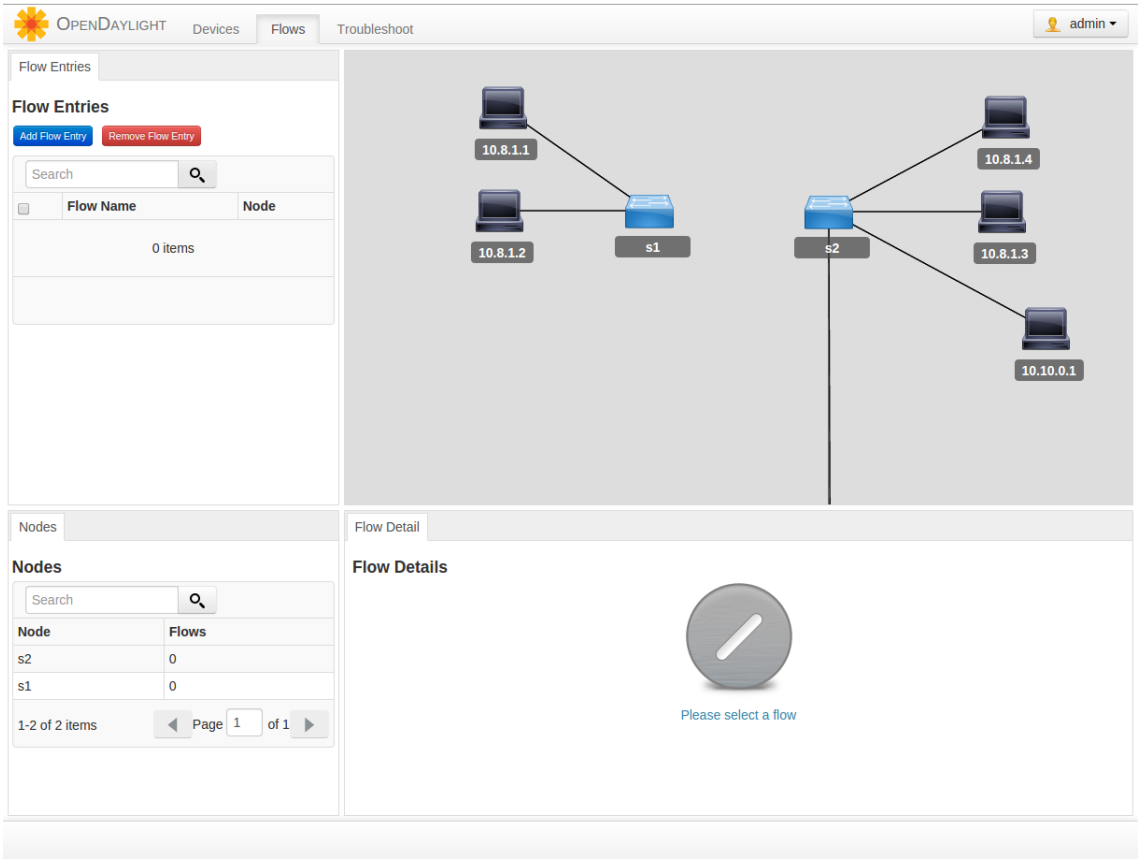


Figure A-1: Management GUI of the OpenDaylight Controller (Base edition).

A-2 sFlow-RT Network Analyzer

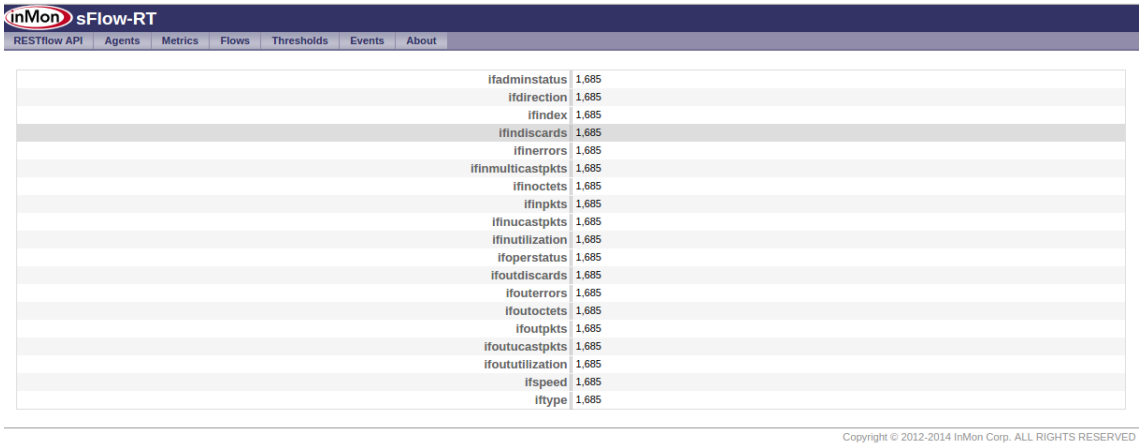


Figure A-2: Interface monitoring metrics in the sFlow-RT network analyzer.

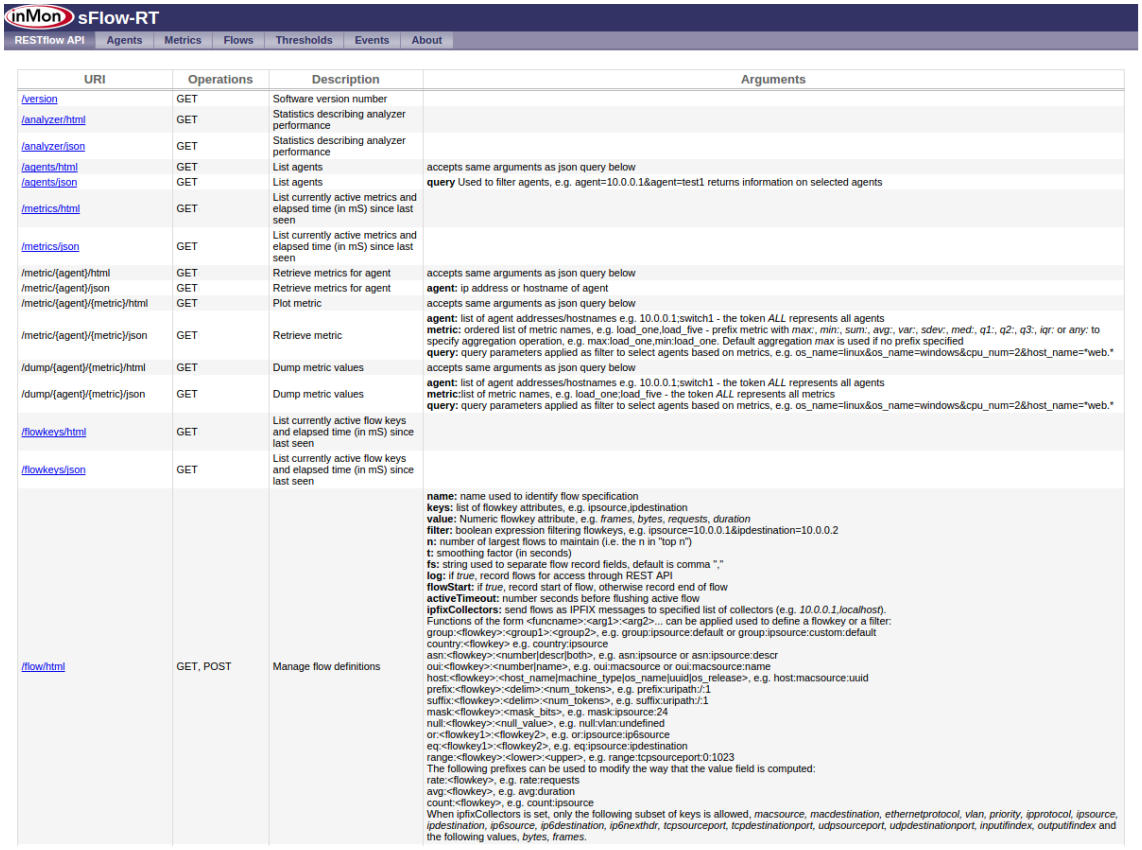


Figure A-3: Management GUI and REST API information of the sFlow-RT network analyzer - 1.

/flow/json	GET	List flow definitions	
/flow/{name}/json	GET, PUT, DELETE	Manage flow definition	**name:** name used to identify flow specification Flow parameters are expressed as JSON object, e.g. {keys:'ipsource,ipdestination', value:'bytes', filter:'ipprotocol=1'}
/activeflows/{agent}/{name}/html	GET	List top active flows, removing duplicates for flows reported by multiple data sources	accepts same arguments as json query below
/activeflows/{agent}/{name}/json	GET	List top active flows, removing duplicates for flows reported by multiple data sources	**agent:** list of agent addresses/hostnames e.g. 10.0.0.1;switch1 - the token *ALL* represents all agents **name:** name used to identify flow specification **query:** set *maxFlows* to change limit number of flow record returned (default is 100), *minValue* to only report flows exceeding specified value, *aggMode* to *sum* or *max* to specify how flows are combined (max is default) e.g. maxFlows=200&minValue=1000&aggMode=sum returns up to 200 active flows with value >= 1000 and summing values for each flow
/flowvalue/{agent}/{name}/json	GET	Get value for a specific flow	**agent:** single agent address / hostname, e.g. 10.0.0.1 **name:** the name used to identify as particular data source and flow metric, e.g 22.tcp queries the tcp flows on interface 22 **query:** the *key* query parameter is used to specify a flow key, e.g. key=10.0.0.1,10.0.0.2,22,45333
/flows/html	GET	List completed flows. Flows will only be logged if *log:true* is specified in the flow specification.	
/flows/json	GET	List completed flows. Flows will only be logged if *log:true* is specified in the flow specification.	**query:** used to filter flows, e.g. name=udp&maxFlows=100 returns most recent 100 flows with name=udp, or to block for flows, e.g. flowID=10&maxFlows=100&timeout=60, waits for up to 60 seconds for flows after flowID 10
/groups/json	GET	List groups and last update times	
/group/{name}/json	GET, PUT, DELETE	Manage IP address groups	Groups define *sourcegroup*, *destinationgroup* attributes for flows, e.g. {external:['0.0.0.0/0'], internal:['10.0.0.0/8','172.16.0.0/12','192.168.0.0/16']}
/threshold/html	GET, POST	Manage thresholds	**name:** name used to identify threshold specification **metric:** metric to apply threshold to, e.g. load_one **value:** threshold value, e.g. 1.0 **filter:** query encoded filter expression consistent with metric query, e.g. os_name=linux&cpu_num=2 **byFlow:** set to *true* to generate a new event for each new flow exceeding threshold, otherwise only first flow generates event **timeout:** seconds of hysteresis before re-arming threshold, i.e. metric value must be below threshold for timeout seconds.
/threshold/json	GET	Retrieve thresholds	
/threshold/{name}/json	GET, PUT, DELETE	Manage definition of threshold	**name:** name used to identify threshold specification Threshold parameters are expressed as JSON object, e.g. {metric:"load_one", value:1, filter:{os_name:["linux"]}}
/events/html	GET	List events	
/events/json	GET	List events	**query:** Used to filter events, e.g. thresholdID=load&maxEvents=100 returns most recent 100 events generated by threshold "load", or to block for events, e.g. ?eventID=10&maxEvents=100&timeout=60, waits for up to 60 seconds for events after eventID 10
/scripts/json	GET	Status of scripts loaded at startup. See System Properties and JavaScript Functions	
/script/{script}/json	GET, POST, PUT, DELETE	*script* specific	Defined by *script*
/forwarding/json	GET	List sFlow forwarding targets	
/forwarding/{name}/json	GET, PUT, DELETE	Manage sFlow forwarding	**name:** name used to identify sFlow target Target is expressed as a JSON object, e.g. {address:'10.0.0.1',port:6343}
/of/switch/{dpid}/json	GET	List ports and switches connected to OpenFlow controller	**dpid** specify a datapath ID, e.g. 0001E8E73277E2B5, to show ports on a specific switch, or ALL to show all switches
/of/rule/{dpid}/{name}/json	GET,PUT,DELETE	Manage OpenFlow rules	**dpid** datapath ID, e.g. 0001E8E73277E2B5 **name** name assigned to rule Rule parameters are expressed as a JSON encoded object, with the following properties: priority, idleTimeout, hardTimeout, match, actions. The value associated with the match property is itself a JSON object, with the following properties: in_port, dl_dst, dl_src, dl_type, dl_vlan, dl_vpcp, nw_dst, nw_src, nw_proto, nw_tos, tp_dst, tp_src. If any of these properties are ommitted, then the field in the match will be a wildcard. For nw_dst and nw_src, a subnet wilcard can be specified using regular subnet notation, eg nw_dst: "10.0.0.0/24". The value associated with the actions property is a JSON array, where each element is a string expression. The expressions are constructed using the tokens output, set_vlan_vid, set_vlan_pcp, strip_vlan, set_dl_src, set_dl_dst, set_nw_src, set_nw_tos, set_tp_src, set_tp_dst, enqueue. For each action, the expression is written as an assignment, eg "set_nw_src=10.0.0.1". The exception is strip_vlan, written just as "strip_vlan". Mac (dl) addresses should be written without separators, eg 0123456789ab, and IP (nw) addresses using conventional dotted notation, eg 10.1.2.3.
/topology/json	GET,PUT	Manage network topology	

Note: RESTflow API is not final and is subject to change in future releases.

Figure A-4: Management GUI and REST API information of the sFlow-RT network analyzer - 2.

A-3 Custom Built SNMP Web Application

Welcome to the SNMP based Web Application (RESTful Web Services) - Home Directory

.

.

.

SNMP based web application - RESTful web services

.

.

.

REST API - URLs Information:

.

.

1) Base URl: http://Host_IP_Address:Port_Number/

=> Example Base URl: http://localhost:8080/

2) http://Host_IP_Address:Port_Number/

=> SNMP based Web Application (RESTful Web Services) - Home Directory

3) http://Host_IP_Address:Port_Number/snmp

=> SNMP based Web Application (RESTful Web Services) - SNMP Directory

4) http://Host_IP_Address:Port_Number/snmp/config

=> SNMP based Web Application (RESTful Web Services) - SNMP Config Directory

5) (GET) http://Host_IP_Address:Port_Number/snmp/config/agents/json

=> For the list of connected/configured SNMP agents.

6) (GET) http://Host_IP_Address:Port_Number/snmp/config/interfaces/'Agent_Name'/json

=> For the list of interfaces of a SNMP agent, replace 'Agent_Name' with the IP address or name of the SNMP agent in the above url.

Figure A-5: REST API information of the custom built SNMP web application - 1.

7) (GET) http://Host_IP_Address:Port_Number/snmp/config/interfaces/mac/'Agent_Name'/json

=> For the MAC addresses of the interfaces of a SNMP agent, replace 'Agent_Name' with the IP address or name of the SNMP agent in the above url.

8) (PUT) http://Host_IP_Address:Port_Number/snmp/config/thresholds/utilization/json

=> For configuring/editing interface utilization thresholds, example JSON request = {'ifinutilization' : '10', 'ifoututilization' : '10'}.

9) (GET) http://Host_IP_Address:Port_Number/snmp/config/thresholds/utilization/json

=> For the configured interface utilization thresholds, example JSON response = {'ifinutilization' : '10', 'ifoututilization' : '10'}.

10) (PUT) http://Host_IP_Address:Port_Number/snmp/events

=> SNMP based Web Application (RESTful Web Services) - SNMP Events Directory

11) (GET) http://Host_IP_Address:Port_Number/snmp/events/interface/status/'Agent_Name'/json

=> For the 'down' operational status events (i.e. link failure events) of an SNMP agent's interfaces, replace 'Agent_Name' with the IP address or name of the SNMP agent in the above url, example JSON response = {'Interface_ID' : 'down', ... }.

12) (GET) http://Host_IP_Address:Port_Number/snmp/events/interface/status/Agent_Name/json

=> For the high interface utilization events (i.e. high link bandwidth utilization events according to the above configured utilization threshold values) of an SNMP agent's interfaces, replace 'Agent_Name' with the IP address or name of the SNMP agent in the above url, example JSON response = {'Interface_ID' : 'Utilization_Value', ... }.

.

.

.

Author Details:

Name: Mani Prashanth Varma Manthena

Contact: me@prashanthvarma.com

Figure A-6: REST API information of the custom built SNMP web application - 2.

A-4 Main App - NaaS Platform's Abstraction Layer

An example abstraction of a REST API call by the Main App in the NaaS platform

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

# This script requires installation of a python module called Requests
# It is a simple and elegant HTTP library for Python

# Importing Python modules
import requests # Python module for oppening URLs (mostly HTTP)
from requests.auth import HTTPBasicAuth # For basic HTTP authentication
import json # Python module for JSON

class odl_api_calls():
    # OpenDaylight (ODL) controller's REST API call:
    # For the list of connected OF/OVS switches to the ODL controller
    # OF - OpenFlow, OVS - Open vSwitch
    # ODL controller's REST API base url:
    # url_odl = http://ip_add:port_no/
    # Deafult url_odl = http://localhost:8080/

    def odl_list_conn(self, url_odl, name, password):
        # Connecting to the ODL controller
        url = url_odl
        url += 'controller/nb/v2/connectionmanager/nodes'
        response = requests.get(url, auth = (name, password))
        if response.status_code != 200:
            # Unable to communicate with ODL controller
            print response.headers
        else:
            # Can successfully communicate with ODL controller
            # List of connected switches to the ODL controller
            list_conn = response.json()
            if list_conn == {}:
                print 'There are no connected OF/OVS switches'
            else:
                odl_switches = []
                for node in list_conn['node']:
                    print 'Switch ID: ' + node['id']
                    print 'Connection Type: ' + node['type']
                    odl_switches.append(node['id'])
                return odl_switches

# Abstracted function call
odl_switch = odl_api_calls().odl_list_conn(url_odl, name, password)
```

```
"odl list conn"     -- For the list of connected OF/OVS switches to the ODL controller
"odl switch ip"     -- For the configured management IP addresses of the OF/OVS switches that are connected to the ODL controller
"edit switch ip"    -- For editing/changing above entries of the configured management IP addresses of the underlying OF/OVS switches
"odl switch alias"  -- For the assigned alias names of the OF/OVS switches that are connected to the ODL controller
"edit switch alias" -- For editing/changing above entries of the alias names of the underlying OF/OVS switches
"odl topo"          -- For the underlying OF/OVS topology with the list of interfaces and their properties
"odl switch prop"   -- For the list of all the underlying OF/OVS switches along with their properties
"odl list hosts"    -- For the list of all the connected end-user hosts along with their configurations
"odl flow stat"     -- For the list of all the installed flows in the underlying OF/OVS switches along with their statistics
"odl port stat"     -- For the list of all the available ports in the underlying OF/OVS switches along with their statistics
"odl table stat"    -- For the list of all the available flow tables in the underlying OF/OVS switches along with their statistics
"static flow inst"  -- For installing static flows in the underlying OF/OVS switch flow tables through the ODL controller's REST API
"static flow stat"  -- For the list of all the installed static flows in the underlying OF/OVS switch flow tables
"static flow del"   -- For deleting static flows in the underlying OF/OVS switch flow tables through the ODL controller's REST API
"mpls push inst"    -- For installing MPLS push flows in the OF/OVS switch flow tables through the ODL controller's RESTCONF API
"hyb mpls push"     -- For installing above MPLS push flows in the case of a hybrid MPLS network with legacy switches as its LSRs
"mpls push stat"    -- For the list of all the installed MPLS push flows in the underlying OF/OVS switch flow tables
"mpls push del"     -- For deleting MPLS push flows in the OF/OVS switch flow tables through the ODL controller's RESTCONF API
"sflow perform"     -- For the statistics describing sFlow-RT analyzer performance
"sflow agents"      -- For the list of connected sFlow agents to the sFlow-RT network analyzer, along with their statistics
"edge sflow alias"  -- For the assigned alias names of the edge sFlow agents that are connected to the edge sFlow-RT network analyzer
"edit edge alias"   -- For editing/changing above entries of the alias names of the underlying network edge sFlow agents
"core sflow alias"  -- For the assigned alias names of the core sFlow agents that are connected to the core sFlow-RT network analyzer
"edit core alias"   -- For editing/changing above entries of the alias names of the underlying network core sFlow agents
"sflow metrics"     -- For the list of currently active sFlow metrics
"sflow met val"     -- For the list of currently active sFlow metrics
"sflow ifs"         -- For the list of interfaces of an sFlow agent(s)
"sflow if met val"  -- For a sFlow metric value of a sFlow agent's interface
"sflow flowkeys"    -- For the list of currently active sFlow flow keys
"sflow flows"       -- For the list of sFlow flow definitions
"sflow flow def"    -- For a sFlow flow definition/details
"sflow flow add"    -- For defining/adding a sFlow flow
"sflow flow del"    -- For deleting a sFlow flow
"sflow activeflows" -- For the list of top active sFlow flows
"sflow comp flows"  -- For the list of completed sFlow flows
"sflow groups"      -- For the list of names of the defined/added sFlow address groups
"sflow group def"   -- For a sFlow address group definition/details
"sflow group add"   -- For defining/adding a sFlow address group to categorize network traffic
"sflow group del"   -- For deleting a sFlow address group
"sflow thresholds"  -- For the list of sFlow thresholds
"sflow thresh def"  -- For a sFlow threshold definition/details
"sflow thresh add"  -- For defining/adding a sFlow threshold
"sflow thresh del"  -- For deleting a sFlow threshold
"sflow events"      -- For the list of sFlow events
"core snmp agents"  -- For the list of connected SNMP agents (i.e. network core legacy switches) to the SNMP monitoring application
"snmp agent int"    -- For the list of interfaces of an SNMP agent along with mappings between SNMP ifindex and physical name
"snmp int mac"      -- For the MAC (i.e. physical) addresses of the SNMP agent interfaces
"uti thresh def"    -- For the SNMP based interface utilization thresholds definition/details
"uti thresh add"    -- For adding/changing SNMP based interface utilization thresholds
"snmp int events"   -- For the list of triggered network core - SNMP interface monitoring events (i.e. high utilizations and failures)
```

Figure A-7: List of REST API calls abstracted by the Main App in the NaaS Platform.

A-5 JSON based Data Store - NaaS Platform's NoSQL Database

An example network topology information in the JSON based Data Store of the NaaS platform

```
{
    "node_ip":{
        "interface_id":"connected_node_ip",
        "so_on":"so_on",
    },
    "192.0.2.4":{
        "te-1/1/13":"192.0.2.13"
    },
    "192.0.2.13":{
        "ge-0/3/0":"192.0.2.15",
        "ge-0/2/0":"192.0.2.11",
        "ge-0/0/0":"192.0.2.4"
    },
    "192.0.2.15":{
        "ge-1/3/0":"192.0.2.11",
        "ge-0/2/0":"192.0.2.14",
        "ge-0/3/0":"192.0.2.13"
    },
    "192.0.2.11":{
        "ge-0/2/0":"192.0.2.14",
        "ge-1/3/0":"192.0.2.15",
        "ge-0/3/0":"192.0.2.13"
    },
    "192.0.2.14":{
        "ge-0/2/0":"192.0.2.15",
        "ge-0/3/0":"192.0.2.11",
        "ge-0/0/0":"192.0.2.5"
    },
    "192.0.2.5":{
        "te-1/1/13":"192.0.2.14"
    }
}
```

An example ingress MPLS label to static LSP path bindings in the JSON based Data Store of the NaaS platform

```
{
    "ingress_mpls_label_for_static_lsp":[
        "first_node_ip_in_static_lsp",
        "second_node_ip_in_static_lsp",
        "so_on",
        "last_node_ip_in_static_lsp",
    ],
    "1000001":[
        "192.0.2.4",
        "192.0.2.13",
        "192.0.2.15",
        "192.0.2.14",
        "192.0.2.5"
    ],
    "1000002":[
        "192.0.2.4",
        "192.0.2.13",
        "192.0.2.15",
        "192.0.2.11",
        "192.0.2.14",
        "192.0.2.5"
    ],
    "1000003":[
        "192.0.2.4",
        "192.0.2.13",
        "192.0.2.11",
        "192.0.2.15",
        "192.0.2.14",
        "192.0.2.5"
    ],
    "1000004":[
        "192.0.2.4",
        "192.0.2.13",
        "192.0.2.11",
        "192.0.2.14",
        "192.0.2.5"
    ],
    "1000005":[
        "192.0.2.5",
        "192.0.2.14",
        "192.0.2.11",
        "192.0.2.13",
        "192.0.2.4"
    ],
    "1000006":[
        "192.0.2.5",
        "192.0.2.14",
        "192.0.2.11",
        "192.0.2.15",
        "192.0.2.13",
```

```
51          "192.0.2.4"
52      ],
53      "1000007":[
54          "192.0.2.5",
55          "192.0.2.14",
56          "192.0.2.15",
57          "192.0.2.11",
58          "192.0.2.13",
59          "192.0.2.4"
60      ],
61      "1000008":[
62          "192.0.2.5",
63          "192.0.2.14",
64          "192.0.2.15",
65          "192.0.2.13",
66          "192.0.2.4"
67      ]
68  }
```

A-6 JSON File Formats for Data Interchange through REST API Calls

An example JSON file format for installing MPLS push label flows in the underlying OpenFlow enabled devices through the OpenDaylight Controller's REST API

```
{
   "flow":{
      "flow-name":"push-mpls-action",
      "instructions":{
         "instruction":{
            "order":"4",
            "apply-actions":{
               "action":[
                  {
                     "push-mpls-action":{
                        "ethernet-type":"34887"
                     },
                     "order":"0"
                  },
                  {
                     "set-field":{
                        "protocol-match-fields":{
                           "mpls-label":"1000001"
                        }
                     },
                     "order":"1"
                  },
                  {
                     "set-field":{
                        "ethernet-match":{
                           "ethernet-destination":{
                              "address":"00:00:00:00:00:01"
                           }
                        }
                     },
                     "order":"2"
                  },
                  {
                     "output-action":{
                        "output-node-connector":"13"
                     },
                     "order":"3"
                  }
               ]
            }
         }
      },
      "strict":"false",
      "id":"100",
      "match":{
         "ethernet-match":{
```

```
47              "ethernet-type":{
48                 "type":"2048"
49              },
50              "ethernet-destination":{
51                 "address":"ff:ff:ff:ff:ff:ff"
52              },
53              "ethernet-source":{
54                 "address":"00:00:00:00:00:00"
55              }
56           },
57           "vlan-match":{
58              "vlan-id":{
59                 "vlan-id":"1",
60                 "vlan-id-present":"true"
61              },
62              "vlan-pcp":"3"
63           },
64           "ipv4-destination":"192.0.2.3",
65           "ipv4-source":"192.0.2.1",
66           "ip-match":{
67              "ip-protocol":"56"
68           },
69           "tcp-source-port":"25364",
70           "tcp-destination-port":"8080",
71           "udp-source-port":"25364",
72           "udp-destination-port":"8080",
73           "in-port":"1"
74        },
75        "idle-timeout":"0",
76        "cookie_mask":"0",
77        "cookie":"0",
78        "priority":"100",
79        "hard-timeout":"0",
80        "installHw":"true",
81        "table_id":"0"
82     }
83  }
```

An example JSON file format for installing static flows in the underlying OpenFlow enabled devices through the OpenDaylight Controller's REST API

```
{
"installInHw":"true",
"name":"flow1",
"node":{
"id":"openflow:2",
"type":"MD_SAL"
},
"priority":"1000",
"etherType":"0x800",
"vlanId":"100",
"vlanPriority":"7",
"nwDst":"192.0.2.4",
"nwSrc":"192.0.2.1",
"dlDst":"00:00:00:00:00:00",
"dlSrc":"00:00:00:00:00:00",
"ingressPort":"openflow:2:1",
"protocol":"6",
"tpSrc":"80",
"tpDst":"80",
"actions":["drop"]
}
```

A-7 NaaS CLI - Northbound Service Provisioning Platform

```
Enter a command to perform a manual NaaS operation...

For the list of available commands, enter: Command = "list" or "help" or "?"

At anytime, if you want to close this NaaS application, enter: command = "exit" (or) "close"



Command: list


Here is the list of available commands to perform NaaS operations:

"odl list conn"     -- For the list of connected OF/OVS switches to the ODL controller
"odl switch ip"     -- For the configured management IP addresses of the OF/OVS switches that are connected to the ODL controller
"edit switch ip"    -- For editing/changing above entries of the configured management IP addresses of the underlying OF/OVS switches
"odl switch alias"  -- For the assigned alias names of the OF/OVS switches that are connected to the ODL controller
"edit switch alias" -- For editing/changing above entries of the alias names of the underlying OF/OVS switches
"odl topo"          -- For the underlying OF/OVS topology with the list of interfaces and their properties
"odl switch prop"   -- For the list of all the underlying OF/OVS switches along with their properties
"odl list hosts"    -- For the list of all the connected end-user hosts along with their configurations
"odl flow stat"     -- For the list of all the installed flows in the underlying OF/OVS switches along with their statistics
"odl port stat"     -- For the list of all the available ports in the underlying OF/OVS switches along with their statistics
"odl table stat"    -- For the list of all the available flow tables in the underlying OF/OVS switches along with their statistics
"static flow inst"  -- For installing static flows in the underlying OF/OVS switch flow tables through the ODL controller's REST API
"static flow stat"  -- For the list of all the installed static flows in the underlying OF/OVS switch flow tables
"static flow del"   -- For deleting static flows in the underlying OF/OVS switch flow tables through the ODL controller's REST API
"mpls push inst"    -- For installing MPLS push flows in the OF/OVS switch flow tables through the ODL controller's RESTCONF API
"hyb mpls push"     -- For installing above MPLS push flows in the case of a hybrid MPLS network with legacy switches as its LSRs
"mpls push stat"    -- For the list of all the installed MPLS push flows in the underlying OF/OVS switch flow tables
"mpls push del"     -- For deleting MPLS push flows in the OF/OVS switch flow tables through the ODL controller's RESTCONF API
"sflow perform"     -- For the statistics describing sFlow-RT analyzer performance
"sflow agents"      -- For the list of connected sFlow agents to the sFlow-RT network analyzer, along with their statistics
"edge sflow alias"  -- For the assigned alias names of the edge sFlow agents that are connected to the edge sFlow-RT network analyzer
"edit edge alias"   -- For editing/changing above entries of the alias names of the underlying network edge sFlow agents
"core sflow alias"  -- For the assigned alias names of the core sFlow agents that are connected to the core sFlow-RT network analyzer
"edit core alias"   -- For editing/changing above entries of the alias names of the underlying network core sFlow agents
"sflow metrics"     -- For the list of currently active sFlow metrics
"sflow met val"     -- For the list of currently active sFlow metrics
"sflow ifs"         -- For the list of interfaces of an sFlow agent(s)
"sflow if met val"  -- For a sFlow metric value of a sFlow agent's interface
"sflow flowkeys"    -- For the list of currently active sFlow flow keys
"sflow flows"       -- For the list of sFlow flow definitions
"sflow flow def"    -- For a sFlow flow definition/details
"sflow flow add"    -- For defining/adding a sFlow flow
"sflow flow del"    -- For deleting a sFlow flow
"sflow activeflows" -- For the list of top active sFlow flows
"sflow comp flows"  -- For the list of completed sFlow flows
"sflow groups"      -- For the list of names of the defined/added sFlow address groups
"sflow group def"   -- For a sFlow address group definition/details
"sflow group add"   -- For defining/adding a sFlow address group to categorize network traffic
"sflow group del"   -- For deleting a sFlow address group
"sflow thresholds"  -- For the list of sFlow thresholds
"sflow thresh def"  -- For a sFlow threshold definition/details
"sflow thresh add"  -- For defining/adding a sFlow threshold
"sflow thresh_del"  -- For deleting a sFlow threshold
"sflow events"      -- For the list of sFlow events
"core snmp agents"  -- For the list of connected SNMP agents (i.e. network core legacy switches) to the SNMP monitoring application
"snmp agent int"    -- For the list of interfaces of an SNMP agent along with mappings between SNMP ifindex and physical name
"snmp int mac"      -- For the MAC (i.e. physical) addresses of the SNMP agent interfaces
"uti thresh def"    -- For the SNMP based interface utilization thresholds definition/details
"uti thresh add"    -- For adding/changing SNMP based interface utilization thresholds
"snmp int events"   -- For the list of triggered network core - SNMP interface monitoring events (i.e. high utilizations and failures)
"test 1 topo"       -- For the topology of testbed network 1 (i.e. network with open (i.e. OF/OVS) switches)
"test 2 topo"       -- For the topology of testbed network 2 (i.e. network core with legacy (i.e. vendor-specific) switches)
"test 1 lsps"       -- For the ingress MPLS push label to path bindings (i.e. static network core LSPs) of testbed network 1
"test 2 lsps"       -- For the ingress MPLS push label to path bindings (i.e. static network core LSPs) of testbed network 2
"load balancing"    -- For running a new instance of Network-as-a-Service (NaaS) platform's load balancing application
"edge firewall"     -- For running a new instance of Network-as-a-Service (NaaS) platform's edge firewall application
"basic_connectivity"-- For running a new instance of Network-as-a-Service (NaaS) platform's basic connectivity application
"clear"             -- For clearing the  NaaS application CLI/terminal at any time
"clear all"         -- For clearing the  NaaS application CLI/terminal at any time
"exit" (or) "close" -- For closing this NaaS application at any time


Command:
```

Figure A-8: NaaS CLI command prompt for service provisioning and management.

Appendix B

Physical Network Testbed Components

B-1 Pica8 P-3922 Open Switches as MPLS LERs

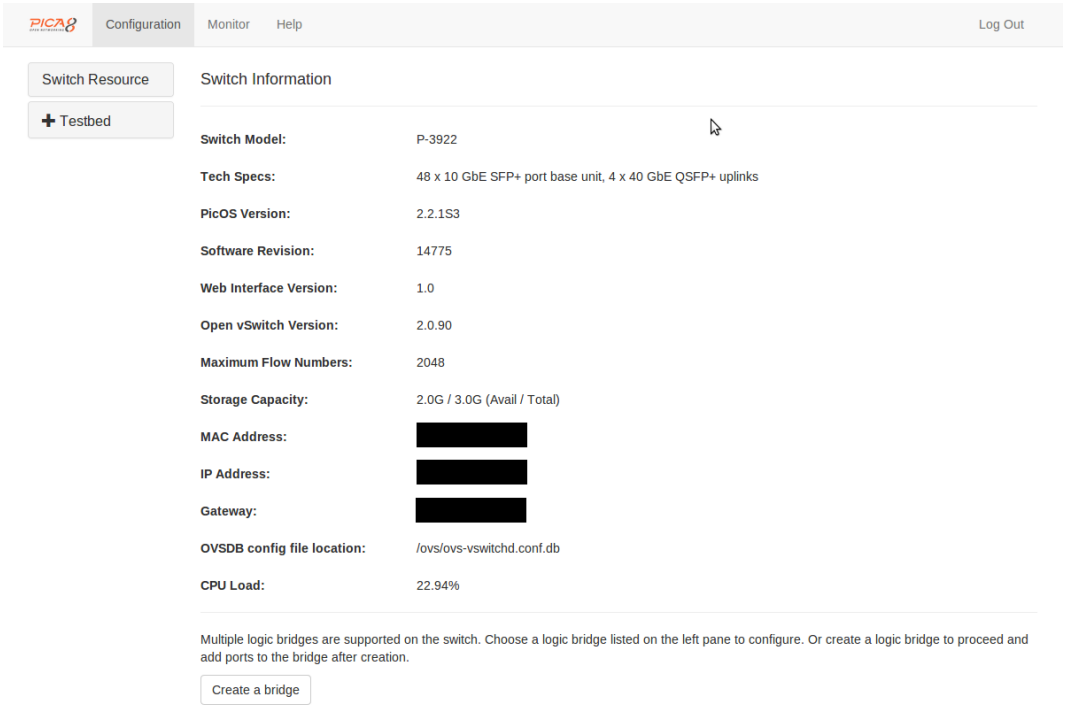


Figure B-1: Management GUI of a Pica8 P-3922 open switch.

Configuration of a Pica8 MPLS LER in the physical network testbed along with its Open vSwitch bridge information

```
root@PicOS-OVS$ovs-ofctl show Testbed

OFPT_FEATURES_REPLY (OF1.3) (xid=0x2): dpid:4d6a486e7300006a
n_tables:254, n_buffers:256
capabilities: FLOW_STATS TABLE_STATS PORT_STATS GROUP_STATS
OFPST_PORT_DESC reply (OF1.3) (xid=0x4):
 1(te-1/1/1): addr:xx:xx:xx:xx:xx:xx
     config:     0
     state:      LINK_UP
     current:    1GB-FD FIBER AUTO_NEG
     advertised: 1GB-FD 10GB-FD FIBER AUTO_NEG
     supported:  100MB-HD 100MB-FD 1GB-FD 10GB-FD FIBER AUTO_NEG
     speed: 1000 Mbps now, 10000 Mbps max
 5(te-1/1/5): addr:xx:xx:xx:xx:xx:xx
     config:     0
     state:      LINK_UP
     current:    1GB-FD FIBER AUTO_NEG
     advertised: 1GB-FD 10GB-FD FIBER AUTO_NEG
     supported:  100MB-HD 100MB-FD 1GB-FD 10GB-FD FIBER AUTO_NEG
     peer:       1GB-FD FIBER
     speed: 1000 Mbps now, 10000 Mbps max
 9(te-1/1/9): addr:xx:xx:xx:xx:xx:xx
     config:     0
     state:      LINK_UP
     current:    1GB-FD FIBER AUTO_NEG
     advertised: 1GB-FD 10GB-FD FIBER AUTO_NEG
     supported:  100MB-HD 100MB-FD 1GB-FD 10GB-FD FIBER AUTO_NEG
     peer:       1GB-FD FIBER
     speed: 1000 Mbps now, 10000 Mbps max
 13(te-1/1/13): addr:xx:xx:xx:xx:xx:xx
     config:     0
     state:      LINK_UP
     current:    1GB-FD FIBER AUTO_NEG
     advertised: 1GB-FD 10GB-FD FIBER AUTO_NEG
     supported:  100MB-HD 100MB-FD 1GB-FD 10GB-FD FIBER AUTO_NEG
     peer:       1GB-FD FIBER
     speed: 1000 Mbps now, 10000 Mbps max
 LOCAL(Testbed): addr:xx:xx:xx:xx:xx:xx
     config:     0
     state:      LINK_UP
     current:    10MB-FD COPPER
     supported:  10MB-FD COPPER
     speed: 10 Mbps now, 10 Mbps max
OFPT_GET_CONFIG_REPLY (OF1.3) (xid=0x6): frags=normal miss_send_len=0
```

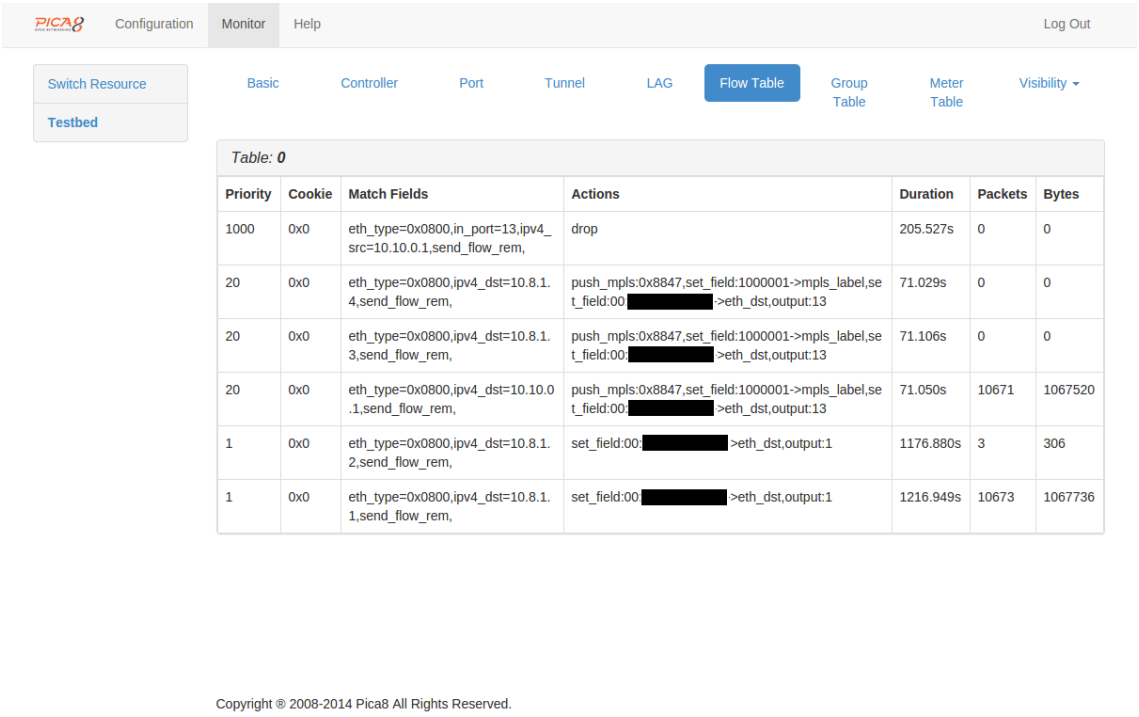


Figure B-2: Example installed flows in a Pica8 MPLS LER.

B-2 Juniper M10i series Legacy Routers as MPLS LSRs

Configuration of a Juniper M10i series MPLS LSR in Testbed Setup A

```
1  --- JUNOS 13.2R1.7 built 2013-08-24 07:03:38 UTC
2
3  admin@Face> show configuration
4
5  ## Last commit: 2014-07-30 16:38:19 UTC by admin
6  version 13.2R1.7;
7  system {
8      host-name Face;
9      root-authentication {
10         encrypted-password "..."; ## SECRET-DATA
11     }
12     login {
13         user admin {
14             uid 2004;
15             class super-user;
16             authentication {
17                 encrypted-password "..."; ## SECRET-DATA
18             }
19         }
20     }
21     services {
22         ftp;
23         ssh {
24             root-login allow;
25         }
26         telnet;
27     }
28     syslog {
29         user * {
30             any emergency;
31         }
32         file messages {
33             any notice;
34             authorization info;
35         }
36         file interactive-commands {
37             interactive-commands any;
38         }
39     }
40 }
41 interfaces {
42     ge-0/0/0 {
43         description "To OVS-1 te-1/1/13";
44         unit 0 {
45             family inet {
46                 address 10.1.0.25/30 {
47                     arp 10.1.0.26 mac xx:xx:xx:xx:xx:xx publish;
48                 }
```

```
49              }
50              family mpls;
51          }
52      }
53      ge-0/2/0 {
54          description "To Murdock ge-0/3/0";
55          unit 0 {
56              family inet {
57                  address 10.1.0.13/30;
58              }
59              family mpls;
60          }
61      }
62      ge-0/3/0 {
63          description "To BA  ge-0/3/0";
64          unit 0 {
65              family inet {
66                  address 10.1.0.10/30;
67              }
68              family mpls;
69          }
70      }
71      fxp0 {
72          unit 0 {
73              family inet {
74                  address 192.0.2.13/25;
75              }
76          }
77      }
78      lo0 {
79          unit 0 {
80              family inet {
81                  address 172.16.0.4/32;
82              }
83          }
84      }
85  }
86  snmp {
87      community public;
88  }
89  routing-options {
90      static {
91          route 192.0.2.0/16 {
92              next-hop 192.0.2.1;
93              no-readvertise;
94          }
95          route 192.0.2.0/16 {
96              next-hop 192.0.2.0;
97              no-readvertise;
98          }
99          route 10.8.1.0/24 next-hop 10.1.0.26;
100     }
101     router-id 172.16.0.4;
```

```
102     autonomous-system 65001;
103 }
104 protocols {
105     rsvp {
106         interface lo0.0;
107         interface ge-0/0/0.0;
108         interface ge-0/2/0.0;
109         interface ge-0/3/0.0;
110     }
111     mpls {
112         static-label-switched-path path1 {
113             transit 1000001 {
114                 next-hop 10.1.0.9;
115                 swap 1000011;
116             }
117         }
118         static-label-switched-path path2 {
119             transit 1000002 {
120                 next-hop 10.1.0.9;
121                 swap 1000021;
122             }
123         }
124         static-label-switched-path path3 {
125             transit 1000003 {
126                 next-hop 10.1.0.14;
127                 swap 1000031;
128             }
129         }
130         static-label-switched-path path4 {
131             transit 1000004 {
132                 next-hop 10.1.0.14;
133                 swap 1000041;
134             }
135         }
136         static-label-switched-path path5 {
137             transit 1000052 {
138                 next-hop 10.1.0.26;
139                 pop;
140             }
141         }
142         static-label-switched-path path6 {
143             transit 1000063 {
144                 next-hop 10.1.0.26;
145                 pop;
146             }
147         }
148         static-label-switched-path path7 {
149             transit 1000073 {
150                 next-hop 10.1.0.26;
151                 pop;
152             }
153         }
154         static-label-switched-path path8 {
```

```
155             transit 1000082 {
156                 next-hop 10.1.0.26;
157                 pop;
158             }
159         }
160         interface ge-0/0/0.0;
161         interface ge-0/2/0.0;
162         interface ge-0/3/0.0;
163     }
164     ospf {
165         traffic-engineering;
166         area 0.0.0.0 {
167             interface lo0.0;
168             interface ge-0/0/0.0;
169             interface ge-0/2/0.0;
170             interface ge-0/3/0.0;
171         }
172     }
173     lldp-med {
174         interface all;
175     }
176 }
```

Route table entries of a Juniper M10i series MPLS LSR in Testbed Setup A

```
admin@Face> show route

inet.0: 22 destinations, 22 routes (22 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

10.1.0.0/30        *[OSPF/10] 1w5d 23:33:34, metric 2
                    > to 10.1.0.9 via ge-0/3/0.0
10.1.0.4/30        *[OSPF/10] 1w6d 00:12:22, metric 2
                    > to 10.1.0.14 via ge-0/2/0.0
                      to 10.1.0.9 via ge-0/3/0.0
10.1.0.8/30        *[Direct/0] 1w6d 00:43:40
                    > via ge-0/3/0.0
10.1.0.10/32       *[Local/0] 1w6d 00:43:40
                      Local via ge-0/3/0.0
10.1.0.12/30       *[Direct/0] 1w6d 00:43:40
                    > via ge-0/2/0.0
10.1.0.13/32       *[Local/0] 1w6d 00:43:40
                      Local via ge-0/2/0.0
10.1.0.16/30       *[OSPF/10] 1w5d 23:33:35, metric 2
                    > to 10.1.0.14 via ge-0/2/0.0
10.1.0.20/30       *[OSPF/10] 02:04:37, metric 3
                      to 10.1.0.14 via ge-0/2/0.0
                    > to 10.1.0.9 via ge-0/3/0.0
10.1.0.24/30       *[Direct/0] 02:04:56
                    > via ge-0/0/0.0
10.1.0.25/32       *[Local/0] 1w6d 00:43:40
                      Local via ge-0/0/0.0
10.1.0.28/30       *[OSPF/10] 1w6d 00:24:29, metric 2
                    > to 10.1.0.9 via ge-0/3/0.0
10.1.0.32/30       *[OSPF/10] 1w6d 00:12:22, metric 2
                    > to 10.1.0.14 via ge-0/2/0.0
10.8.1.0/24        *[Static/5] 02:04:56
                    > to 10.1.0.26 via ge-0/0/0.0
192.0.2.0/16      *[Static/5] 1w5d 23:56:18
                    > to 192.0.2.1 via fxp0.0
192.0.2.0/16       *[Static/5] 1w5d 23:56:18
                    > to 192.0.2.1 via fxp0.0
192.0.2.0/25      *[Direct/0] 1w5d 23:56:18
                    > via fxp0.0
192.0.2.13/32    *[Local/0] 1w6d 01:12:45
                      Local via fxp0.0
172.16.0.2/32      *[OSPF/10] 1w6d 00:12:22, metric 1
                    > to 10.1.0.14 via ge-0/2/0.0
172.16.0.3/32      *[OSPF/10] 1w6d 00:24:29, metric 1
                    > to 10.1.0.9 via ge-0/3/0.0
172.16.0.4/32      *[Direct/0] 1w6d 01:12:45
                    > via lo0.0
172.16.0.5/32      *[OSPF/10] 1w3d 01:05:13, metric 2
                      to 10.1.0.14 via ge-0/2/0.0
                    > to 10.1.0.9 via ge-0/3/0.0
224.0.0.5/32       *[OSPF/10] 1w6d 01:02:09, metric 1
```

```
52                      MultiRecv
53
54 mpls.0: 16 destinations, 16 routes (16 active, 0 holddown, 0 hidden)
55 + = Active Route, - = Last Active, * = Both
56
57 0                 *[MPLS/0] 1w6d 00:50:42, metric 1
58                     Receive
59 1                 *[MPLS/0] 1w6d 00:50:42, metric 1
60                     Receive
61 2                 *[MPLS/0] 1w6d 00:50:42, metric 1
62                     Receive
63 13                *[MPLS/0] 1w6d 00:50:42, metric 1
64                     Receive
65 1000001           *[MPLS/6] 1w1d 07:22:22, metric 1
66                     > to 10.1.0.9 via ge-0/3/0.0, Swap 1000011
67 1000002           *[MPLS/6] 1w1d 07:22:22, metric 1
68                     > to 10.1.0.9 via ge-0/3/0.0, Swap 1000021
69 1000003           *[MPLS/6] 1w1d 07:22:22, metric 1
70                     > to 10.1.0.14 via ge-0/2/0.0, Swap 1000031
71 1000004           *[MPLS/6] 1w1d 07:22:22, metric 1
72                     > to 10.1.0.14 via ge-0/2/0.0, Swap 1000041
73 1000052           *[MPLS/6] 02:04:56, metric 1
74                     > to 10.1.0.26 via ge-0/0/0.0, Pop
75 1000052(S=0)      *[MPLS/6] 02:04:56, metric 1
76                     > to 10.1.0.26 via ge-0/0/0.0, Pop
77 1000063           *[MPLS/6] 02:04:56, metric 1
78                     > to 10.1.0.26 via ge-0/0/0.0, Pop
79 1000063(S=0)      *[MPLS/6] 02:04:56, metric 1
80                     > to 10.1.0.26 via ge-0/0/0.0, Pop
81 1000073           *[MPLS/6] 02:04:56, metric 1
82                     > to 10.1.0.26 via ge-0/0/0.0, Pop
83 1000073(S=0)      *[MPLS/6] 02:04:56, metric 1
84                     > to 10.1.0.26 via ge-0/0/0.0, Pop
85 1000082           *[MPLS/6] 02:04:56, metric 1
86                     > to 10.1.0.26 via ge-0/0/0.0, Pop
87 1000082(S=0)      *[MPLS/6] 02:04:56, metric 1
88                     > to 10.1.0.26 via ge-0/0/0.0, Pop
```

ARP table entries of a Juniper M10i series MPLS LSR in Testbed Setup A

```
admin@Face> show arp

(MAC Address - Address - Name - Interface - Flags)

xx:xx:xx:xx:xx:xx - 10.1.0.9 - 10.1.0.9 - ge-0/3/0.0 - none

xx:xx:xx:xx:xx:xx - 10.1.0.14 - 10.1.0.14 - ge-0/2/0.0 - none

xx:xx:xx:xx:xx:xx - 10.1.0.26 - 10.1.0.26 - ge-0/0/0.0 - permanent
                                                          published

xx:xx:xx:xx:xx:xx - 128.0.0.2 - 128.0.0.2 - fxp1.0 - none

xx:xx:xx:xx:xx:xx - 192.0.2.1 - 192.0.2.1 - fxp0.0 - none

xx:xx:xx:xx:xx:xx - 192.0.2.9 - 192.0.2.9 - fxp0.0 - none

Total entries: 6
```

B-3 Pica8 P-3290 Open Switches as MPLS LSRs

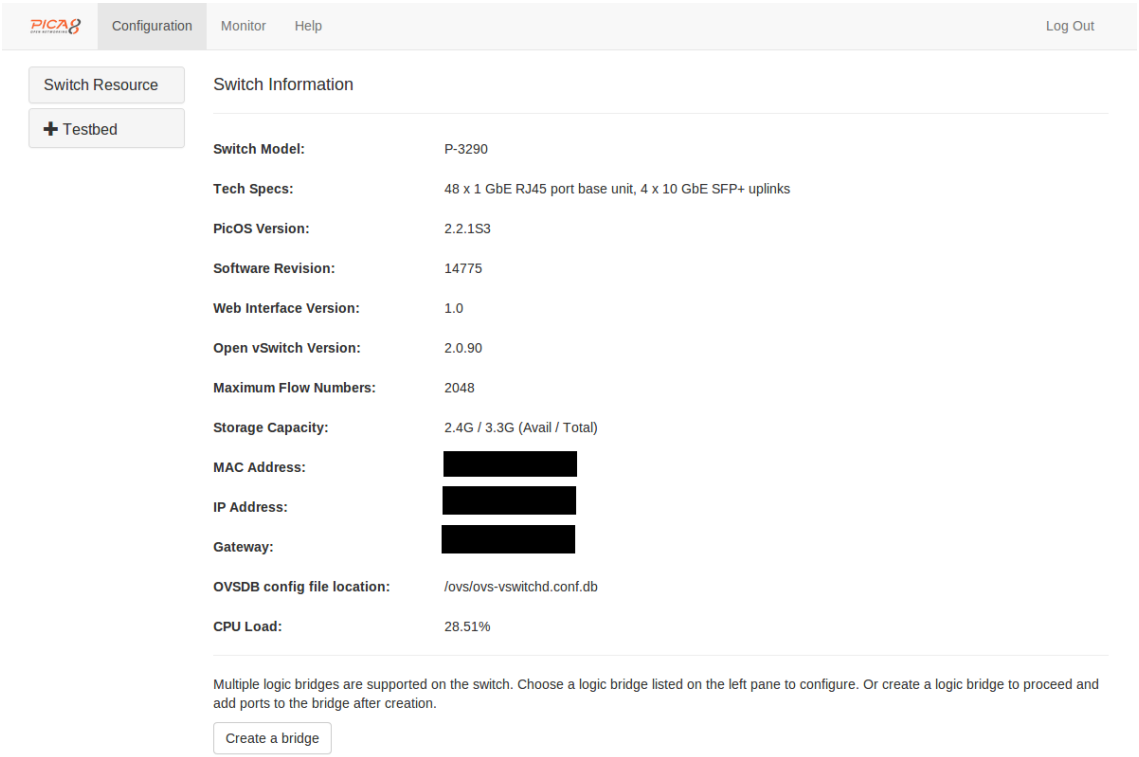


Figure B-3: Management GUI of a Pica8 P-3290 open switch.

Configuration of a Pica8 MPLS LSR in Testbed Setup B along with its Open vSwitch bridge information

```
root@PicOS-OVS$ovs-ofctl show Testbed

OFPT_FEATURES_REPLY (OF1.3) (xid=0x2): dpid:4d6a089e01e9950d
n_tables:254, n_buffers:256
capabilities: FLOW_STATS TABLE_STATS PORT_STATS GROUP_STATS
OFPST_PORT_DESC reply (OF1.3) (xid=0x4):
 49(te-1/1/49): addr:xx:xx:xx:xx:xx:xx
     config:     0
     state:      LINK_UP
     current:    1GB-FD FIBER AUTO_NEG
     advertised: 1GB-FD FIBER AUTO_NEG
     supported:  1GB-FD 10GB-FD FIBER AUTO_NEG
     peer:       1GB-FD FIBER
     speed: 1000 Mbps now, 10000 Mbps max
 50(te-1/1/50): addr:xx:xx:xx:xx:xx:xx
     config:     0
     state:      LINK_UP
     current:    1GB-FD FIBER AUTO_NEG
     advertised: 1GB-FD FIBER AUTO_NEG
     supported:  1GB-FD 10GB-FD FIBER AUTO_NEG
     peer:       1GB-FD FIBER
     speed: 1000 Mbps now, 10000 Mbps max
 51(te-1/1/51): addr:xx:xx:xx:xx:xx:xx
     config:     0
     state:      LINK_UP
     current:    1GB-FD FIBER AUTO_NEG
     advertised: 1GB-FD FIBER AUTO_NEG
     supported:  1GB-FD 10GB-FD FIBER AUTO_NEG
     peer:       1GB-FD FIBER
     speed: 1000 Mbps now, 10000 Mbps max
 52(te-1/1/52): addr:xx:xx:xx:xx:xx:xx
     config:     0
     state:      LINK_UP
     current:    1GB-FD FIBER AUTO_NEG
     advertised: 1GB-FD FIBER AUTO_NEG
     supported:  1GB-FD 10GB-FD FIBER AUTO_NEG
     peer:       1GB-FD FIBER
     speed: 1000 Mbps now, 10000 Mbps max
 LOCAL(Testbed): addr:xx:xx:xx:xx:xx:xx
     config:     0
     state:      LINK_UP
     current:    10MB-FD COPPER
     supported:  10MB-FD COPPER
     speed: 10 Mbps now, 10 Mbps max
```

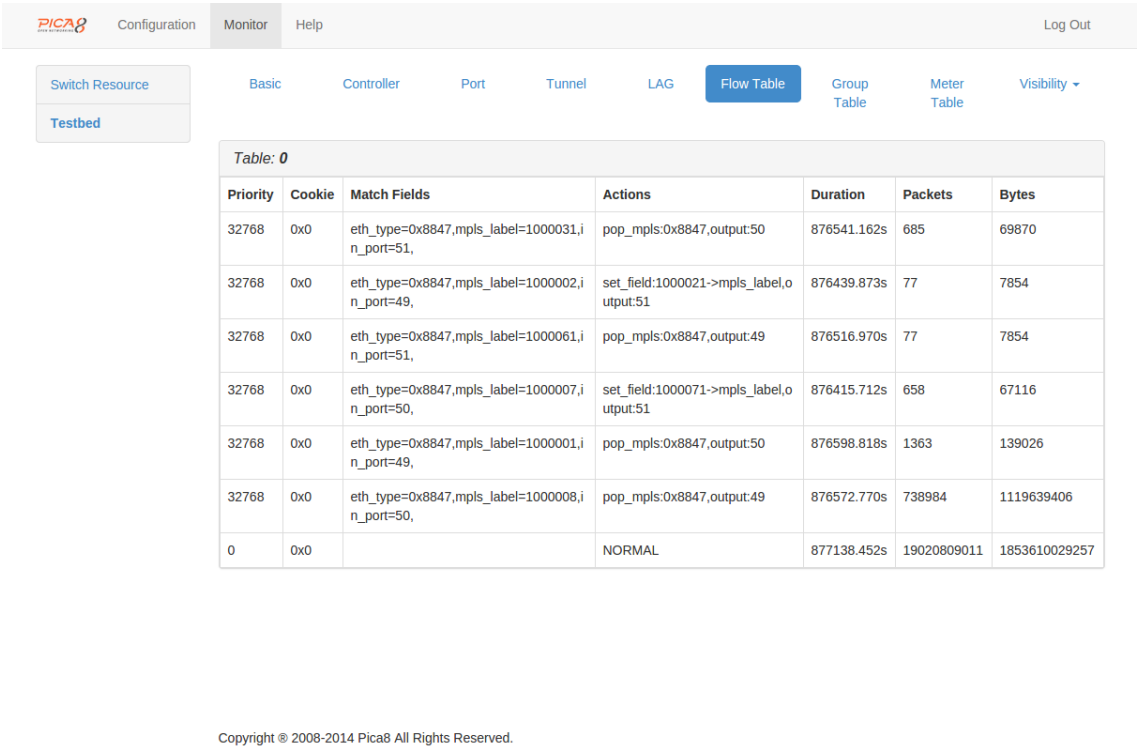


Figure B-4: Installed proactive MPLS flows in a Pica8 MPLS LSR, which describe the pre-installed full-mesh static LSPs in the network core of Testbed Setup B.

Appendix C

Basic Network Connectivity Services Key Enabling Components

C-1 sFlow based Edge Flow Monitoring App

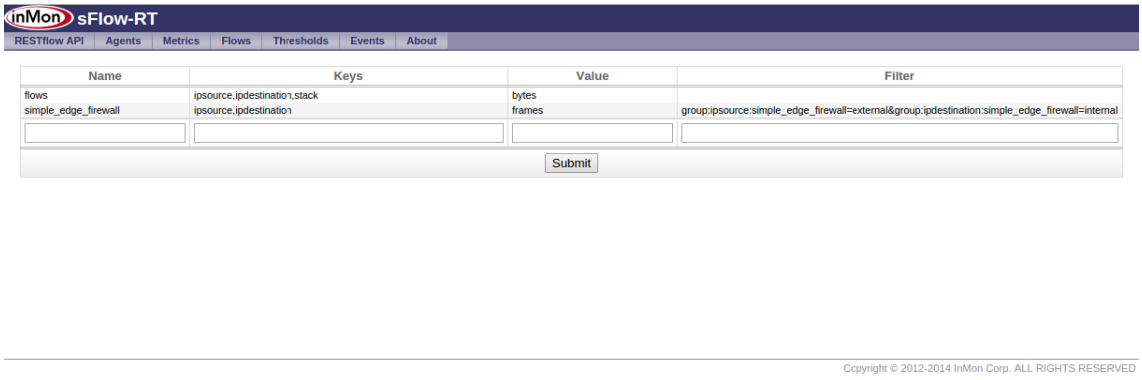


Figure C-1: Example monitoring flows configured in the sFlow-RT network analyzer.

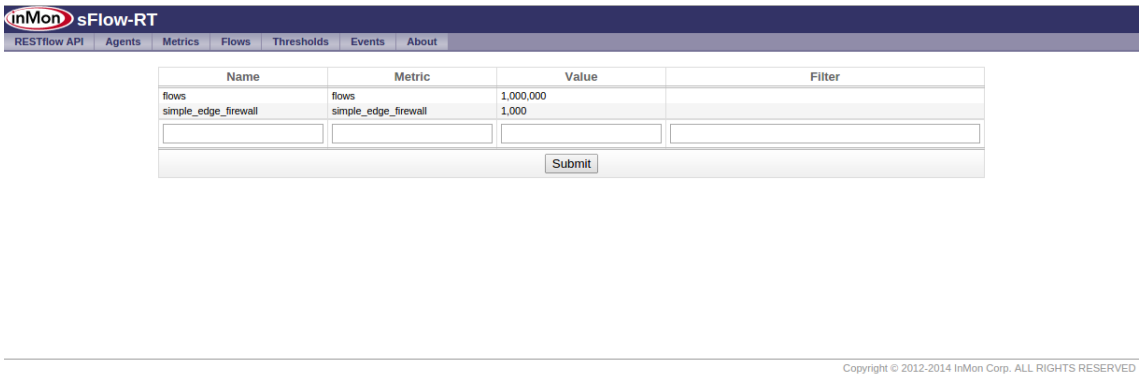


Figure C-2: Example monitoring flow thresholds configured in the sFlow-RT network analyzer.

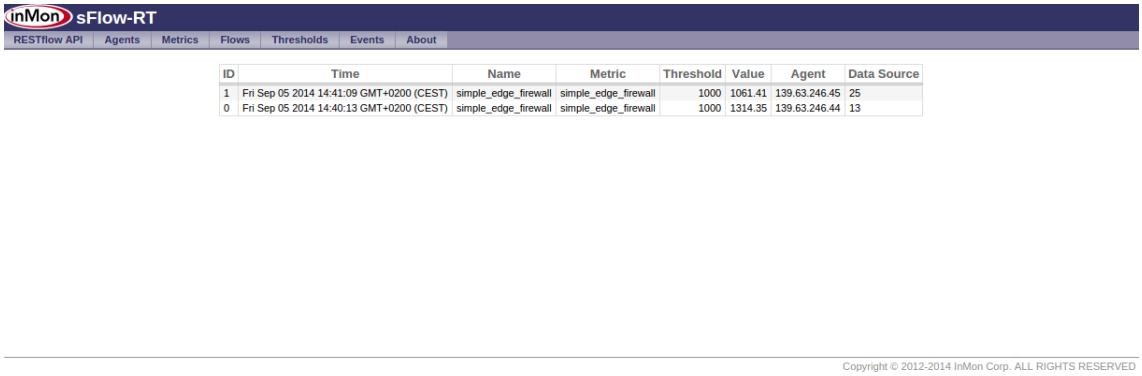


Figure C-3: Example monitoring flow events triggered by the sFlow-RT network analyzer.

C-2 sFlow/SNMP based Core Interface Monitoring App

C-2-1 Testbed Setup A

Example link failure events triggered by the custom built SNMP web application.


```
{
    "1":{
        "Agent":"192.0.2.1",
        "Interface ID":"501",
        "Metric":"ifoperstatus",
        "Value":"down",
        "Last Updated":"123456789"
    },
    "2":{
        "Agent":"192.0.2.2",
        "Interface ID":"502",
        "Metric":"ifoperstatus",
        "Value":"down",
        "Last Updated":"123456789"
    }
}
```


Example link high utilization events triggered by the custom built SNMP web application.


```
{
    "1":{
        "Agent":"192.0.2.3",
        "Interface ID":"503",
        "Metric":"ifinutilization",
        "Value":"25",
        "Last Updated":"123456789"
    },
    "2":{
        "Agent":"192.0.2.3",
        "Interface ID":"504",
        "Metric":"ifoututilization",
        "Value":"25",
        "Last Updated":"123456789"
    }
}
```

C-2-2 Testbed Setup B

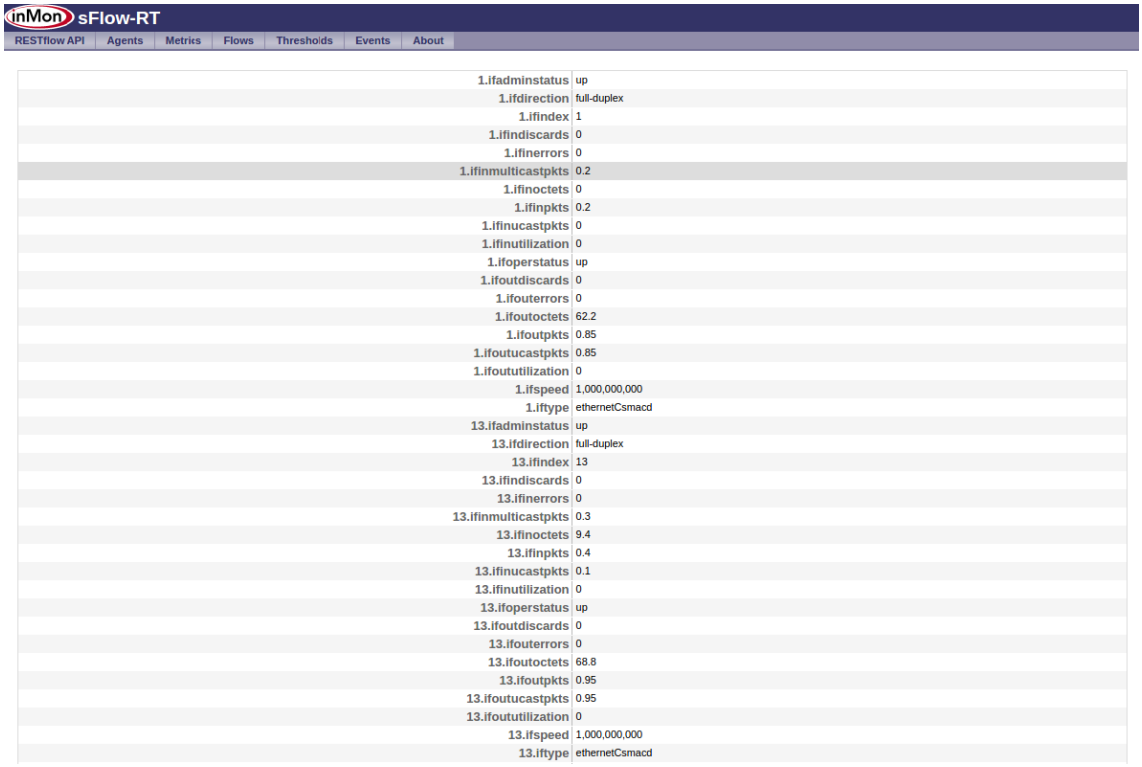


Figure C-4: Example interface monitoring statistics in the sFlow-RT network analyzer.

C-3 Optimal Path Computation App

C-3-1 Testbed Setup A

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}$$

Figure C-5: Network connectivity matrix of Testbed Setup A.

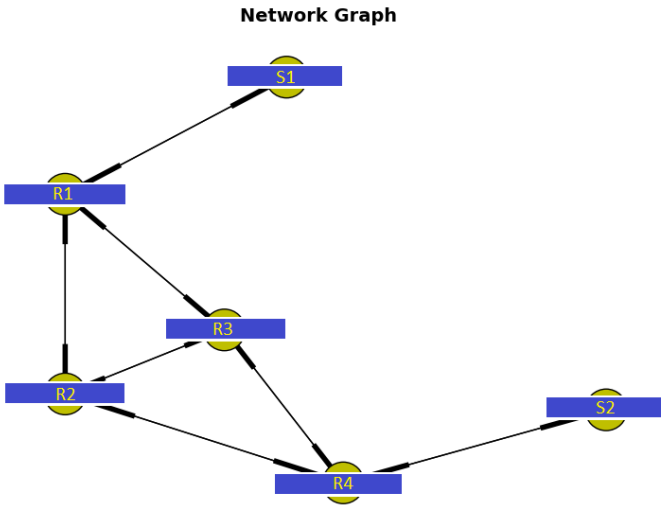


Figure C-6: Network graphical visualization of Testbed Setup A.

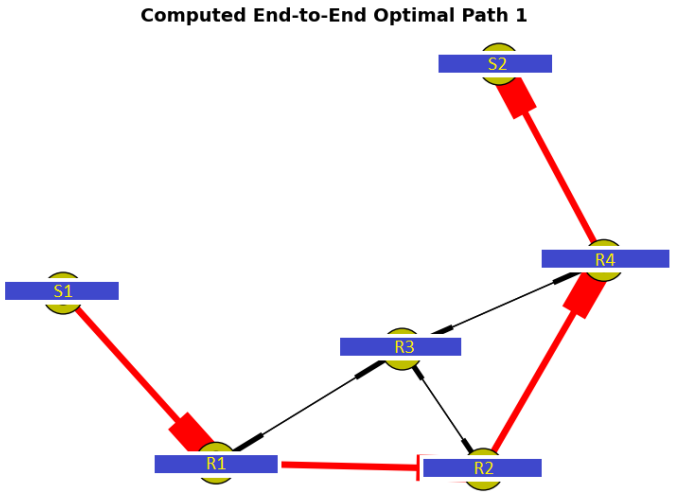


Figure C-7: An example end-to-end computed optimal path in Testbed Setup A - 1.

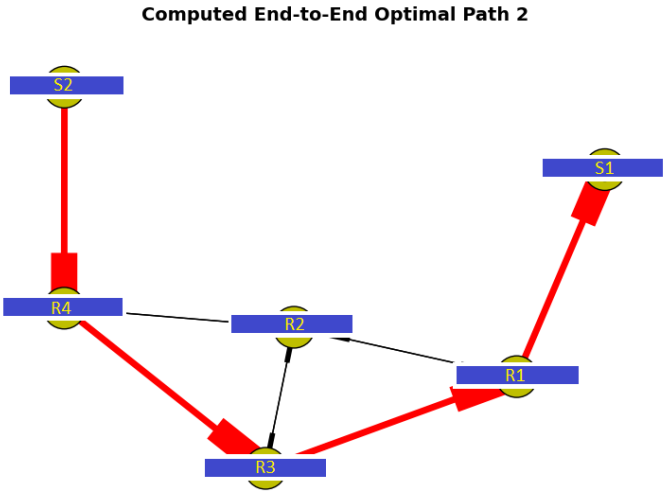


Figure C-8: An example end-to-end computed optimal path in Testbed Setup A - 2.

C-3-2 Testbed Setup B

$$\begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix}$$

Figure C-9: Network connectivity matrix of Testbed Setup B.

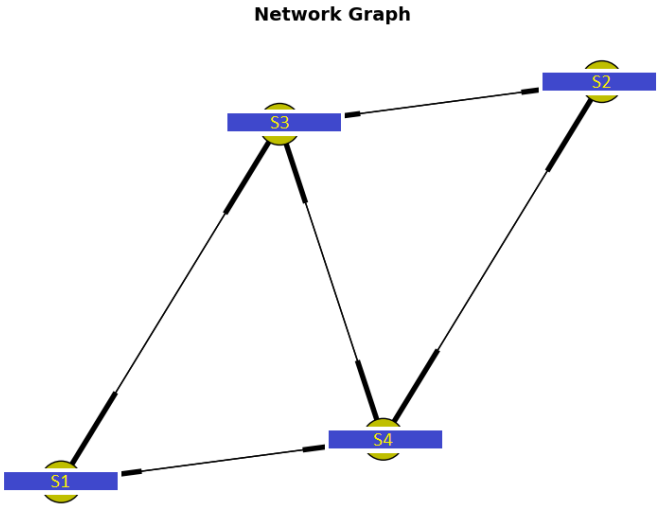


Figure C-10: Network graphical visualization of Testbed Setup B.

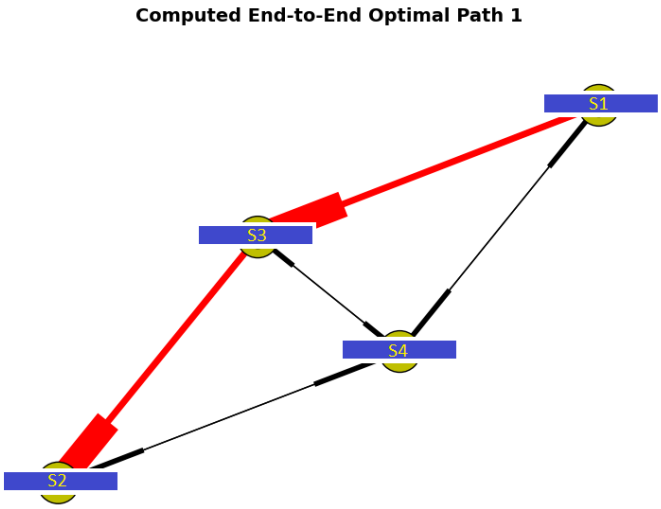


Figure C-11: An example end-to-end computed optimal path in Testbed Setup B - 1.

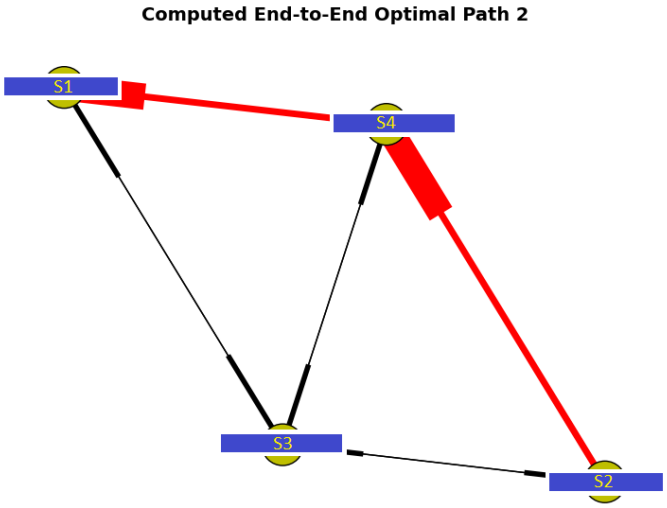


Figure C-12: An example end-to-end computed optimal path in Testbed Setup B - 2.

Appendix D

Experimental Performance Analysis Data and Plots

D-1 Network Control Overhead

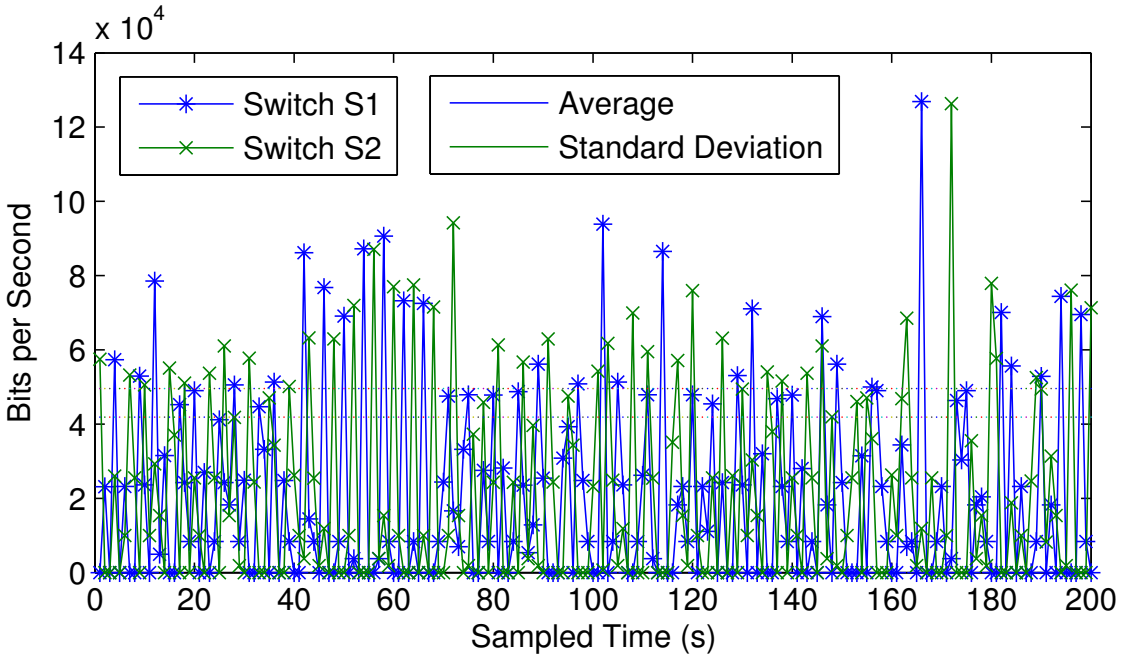


Figure D-1: First sample of the OpenFlow protocol traffic load at the OpenDaylight Controller in Testbed Setup A.

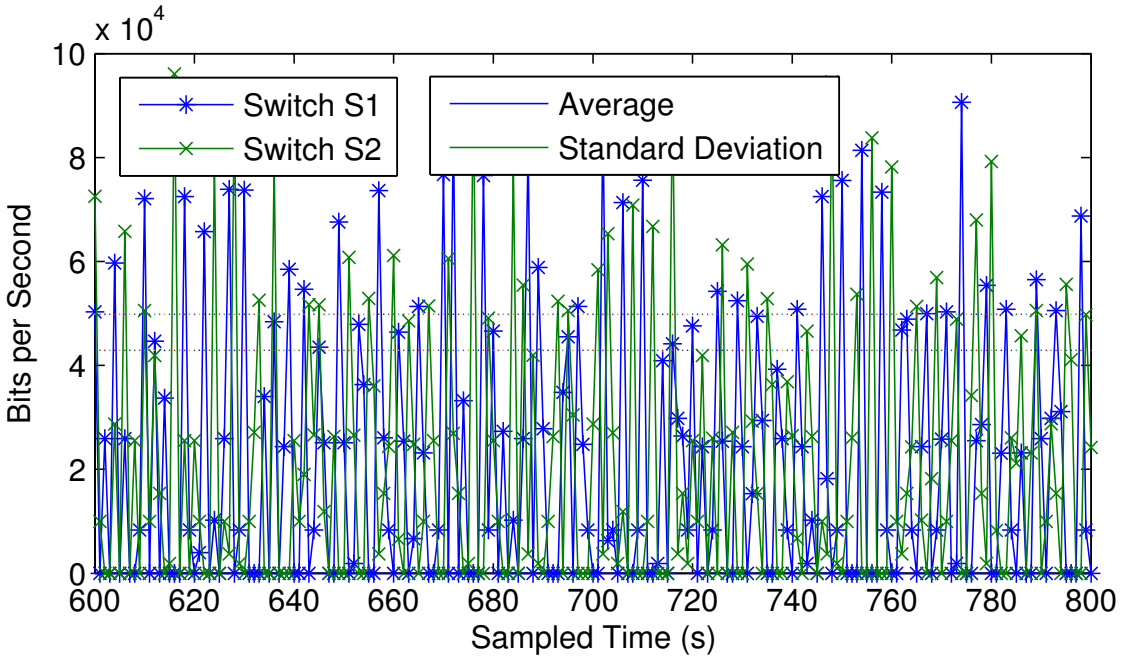


Figure D-2: Second sample of the OpenFlow protocol traffic load at the OpenDaylight Controller in Testbed Setup A.

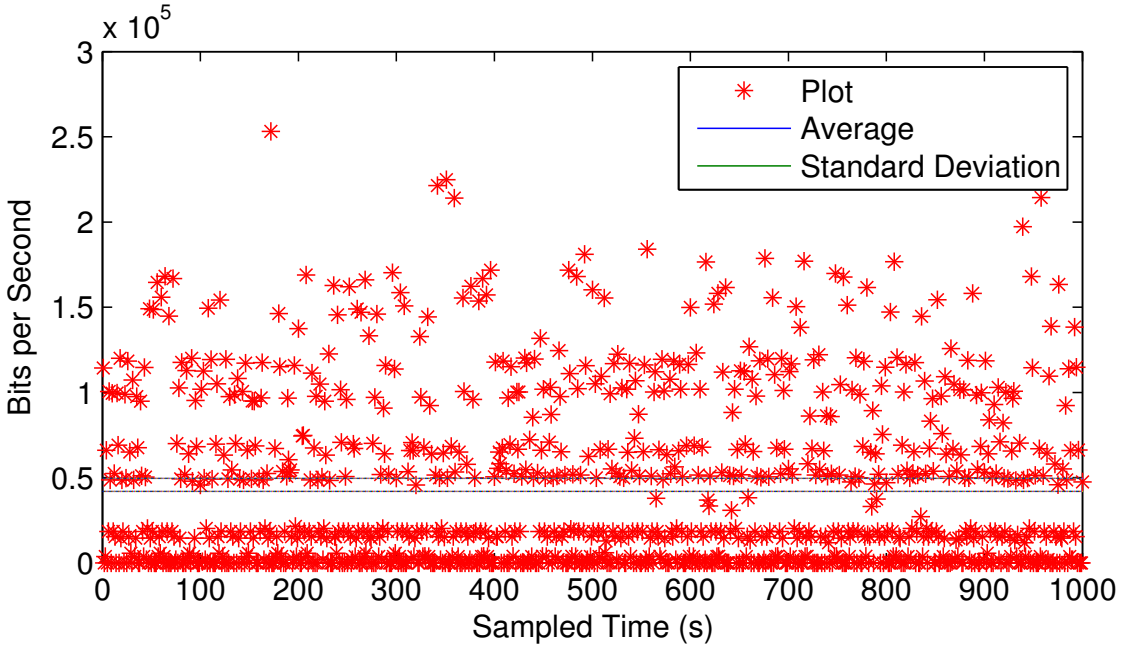


Figure D-3: Total sample of the OpenFlow protocol traffic load at the OpenDaylight Controller in Testbed Setup A.

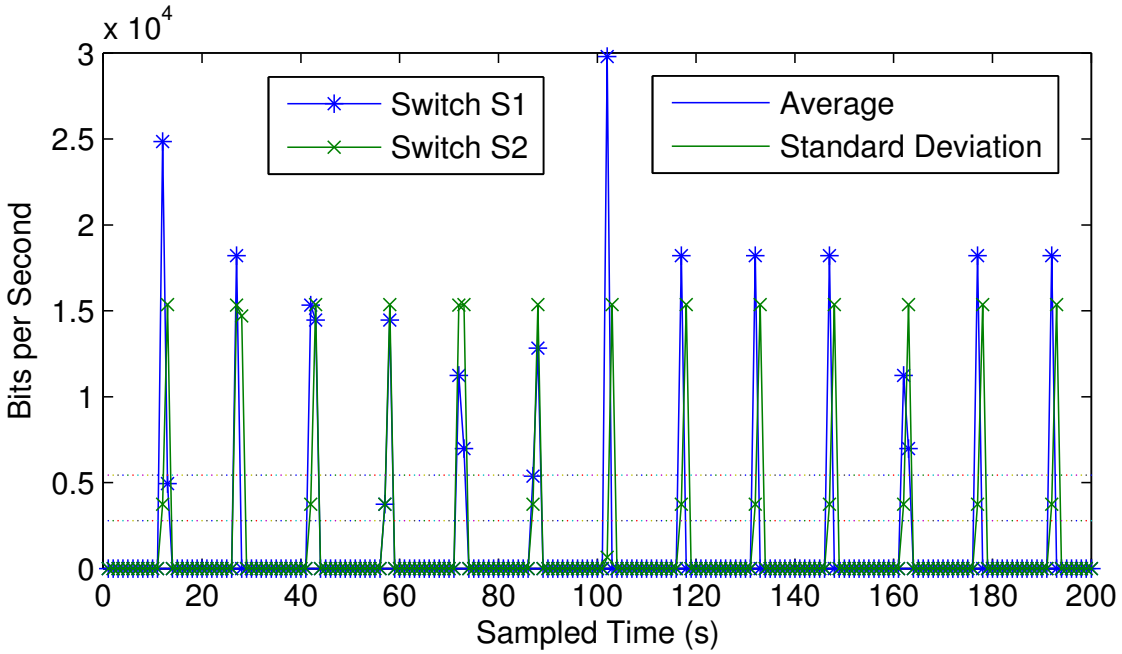


Figure D-4: First sample of the OpenFlow protocol traffic load at the OpenDaylight Controller in Testbed Setup B.

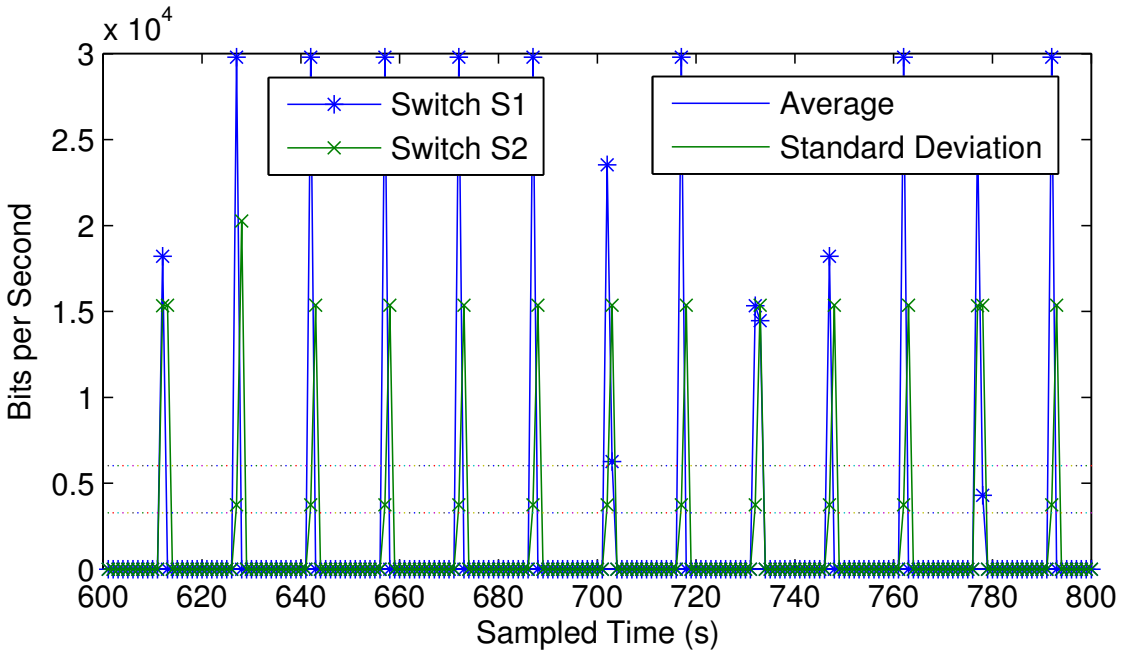


Figure D-5: Second sample of the OpenFlow protocol traffic load at the OpenDaylight Controller in Testbed Setup B.

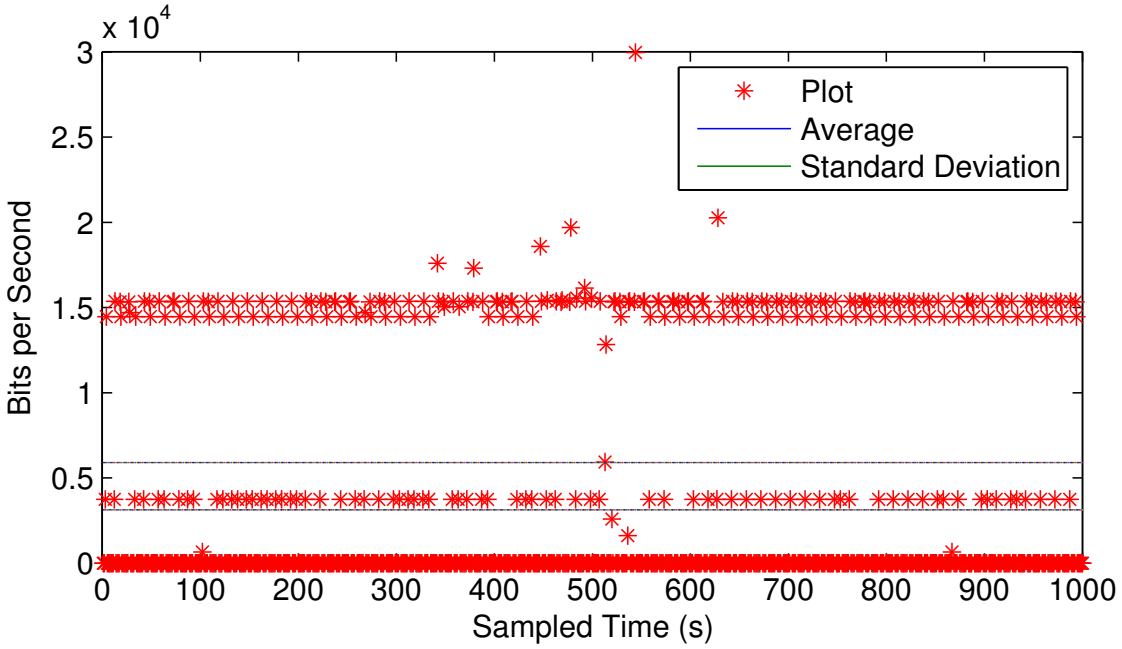


Figure D-6: Total sample of the OpenFlow protocol traffic load at the OpenDaylight Controller in Testbed Setup B.

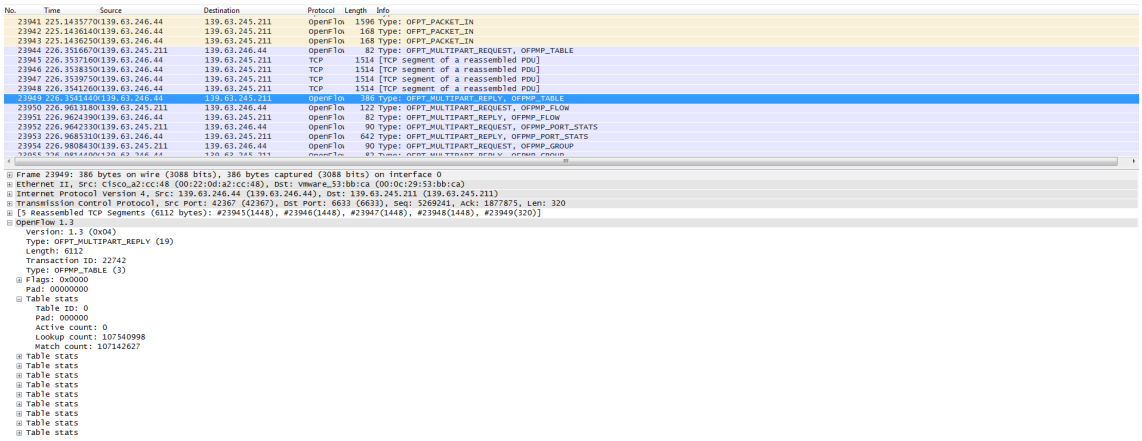


Figure D-7: Wireshark capture of OpenFlow statistics polling by the OpenDaylight Controller via the OpenFlow protocol in the PoC physical network testbed.

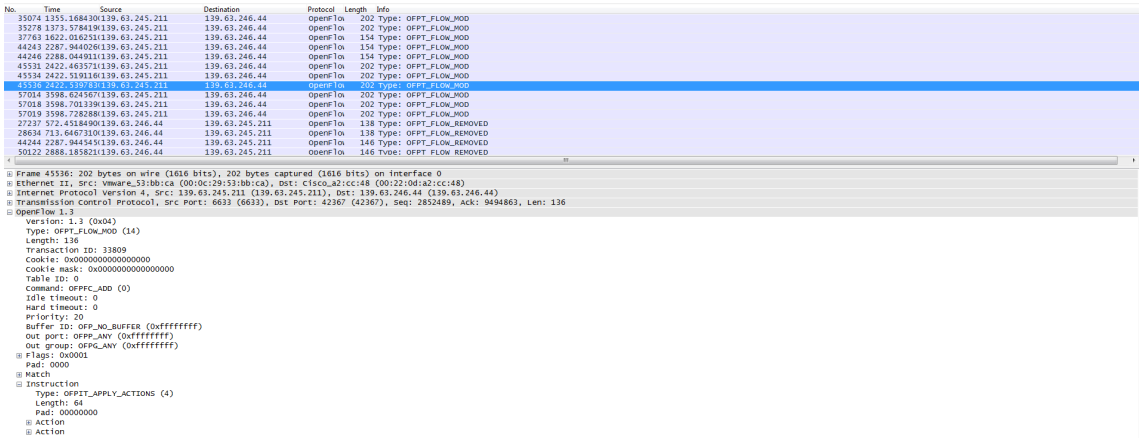


Figure D-8: Wireshark capture of OpenFlow flow modification by the OpenDaylight Controller via the OpenFlow protocol in the PoC physical network testbed.

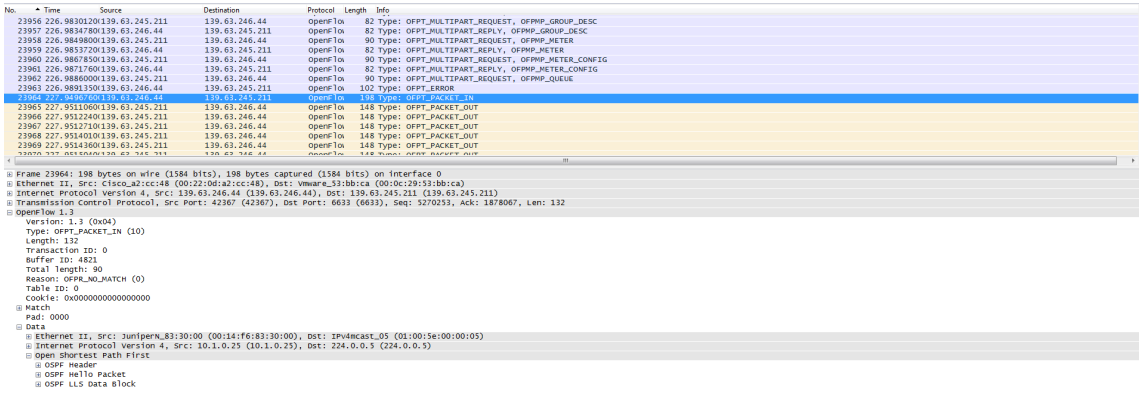


Figure D-9: Wireshark capture of a multicast OSPF "Hello" received by the OpenDaylight Controller via the OpenFlow protocol in Testbed Setup A.

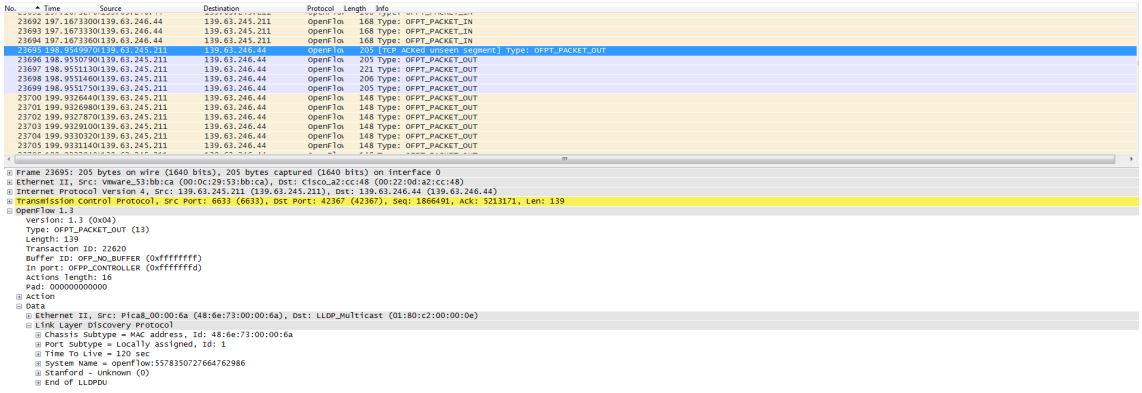


Figure D-10: Wireshark capture of a multicast LLDP request received by the OpenDaylight Controller via the OpenFlow protocol in Testbed Setup A.

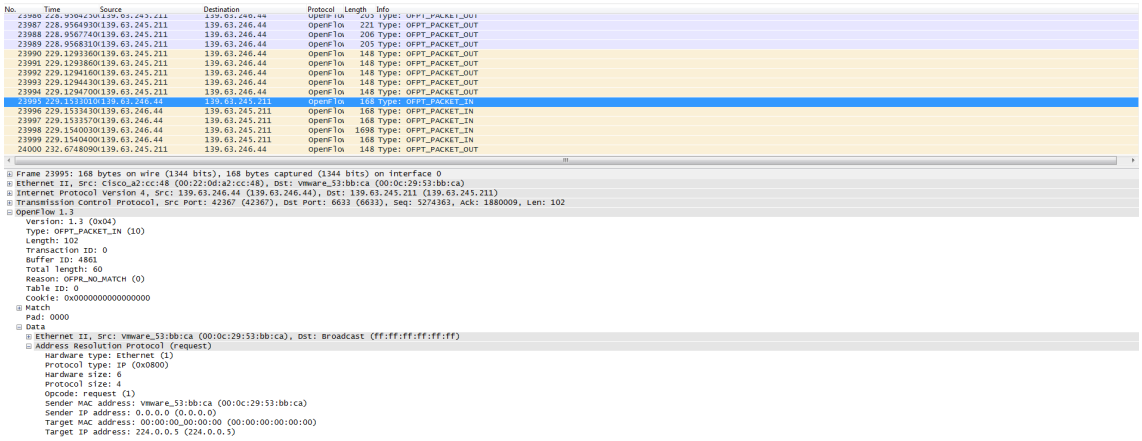


Figure D-11: Wireshark capture of a multicast ARP request received by the OpenDaylight Controller via the OpenFlow protocol in Testbed Setup A.

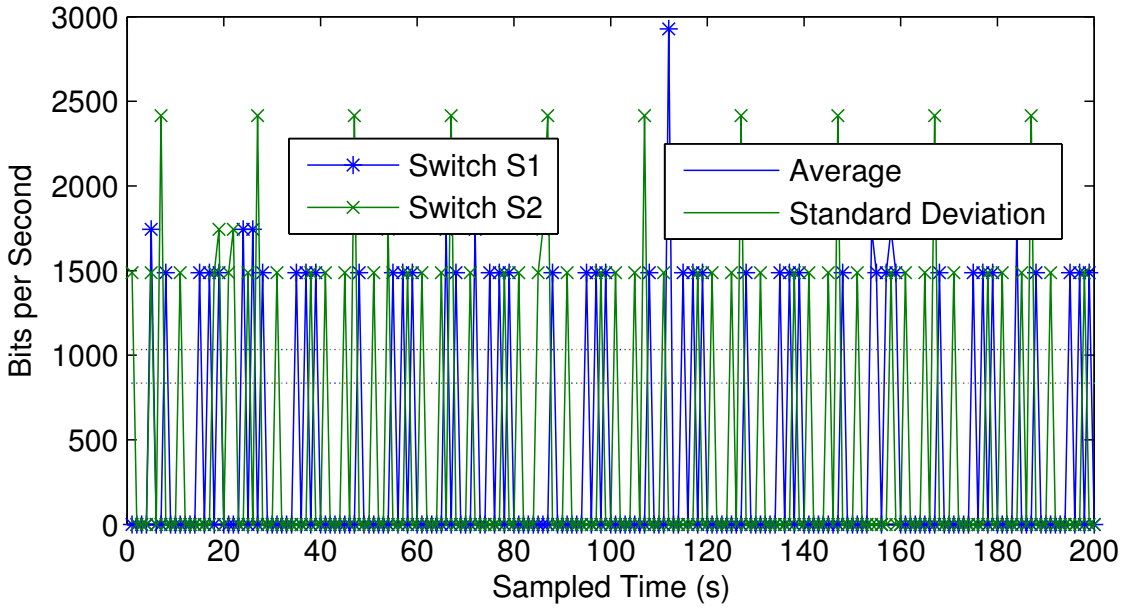


Figure D-12: First sample of the sFlow protocol based edge flow monitoring traffic load at the edge sFlow-RT network analyzer in the PoC physical network testbed.

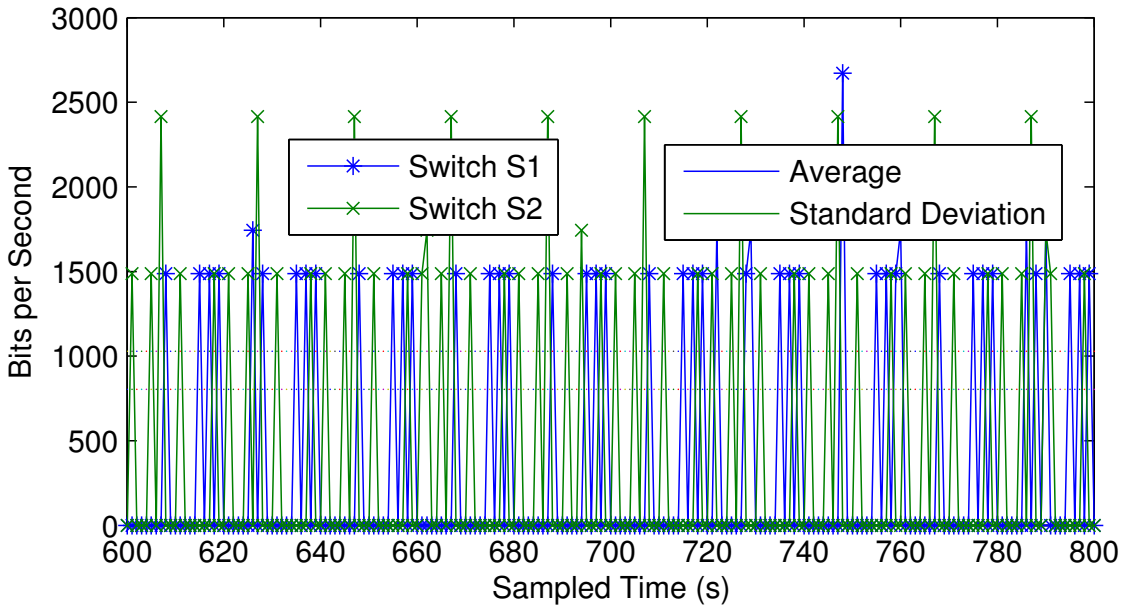


Figure D-13: Second sample of the sFlow protocol based edge flow monitoring traffic load at the edge sFlow-RT network analyzer in the PoC physical network testbed.

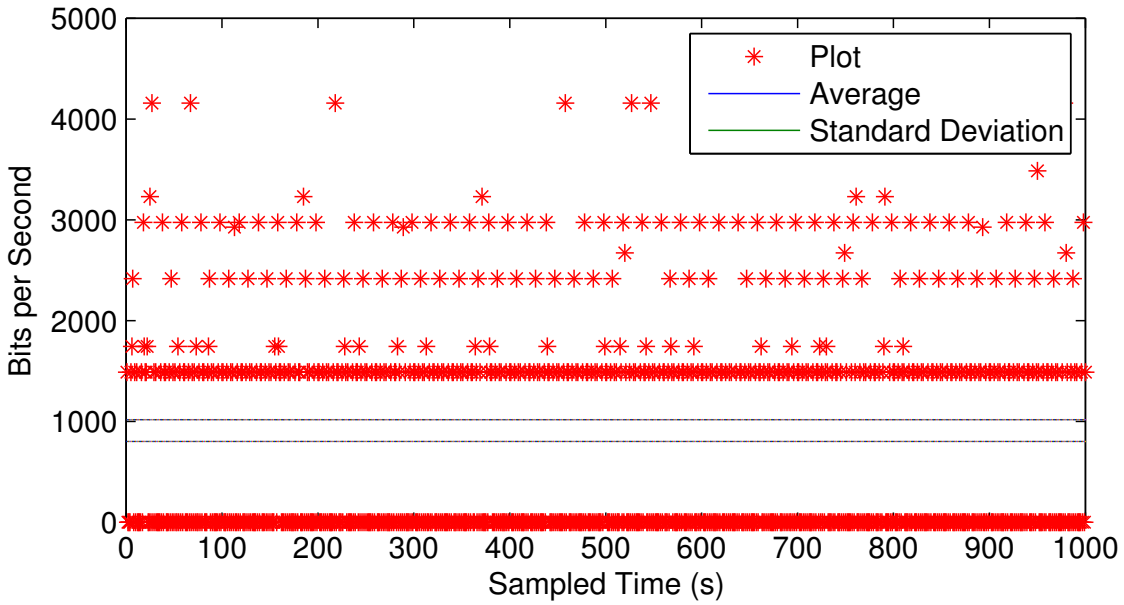


Figure D-14: Total sample of the sFlow protocol based edge flow monitoring traffic load at the edge sFlow-RT network analyzer in the PoC physical network testbed.

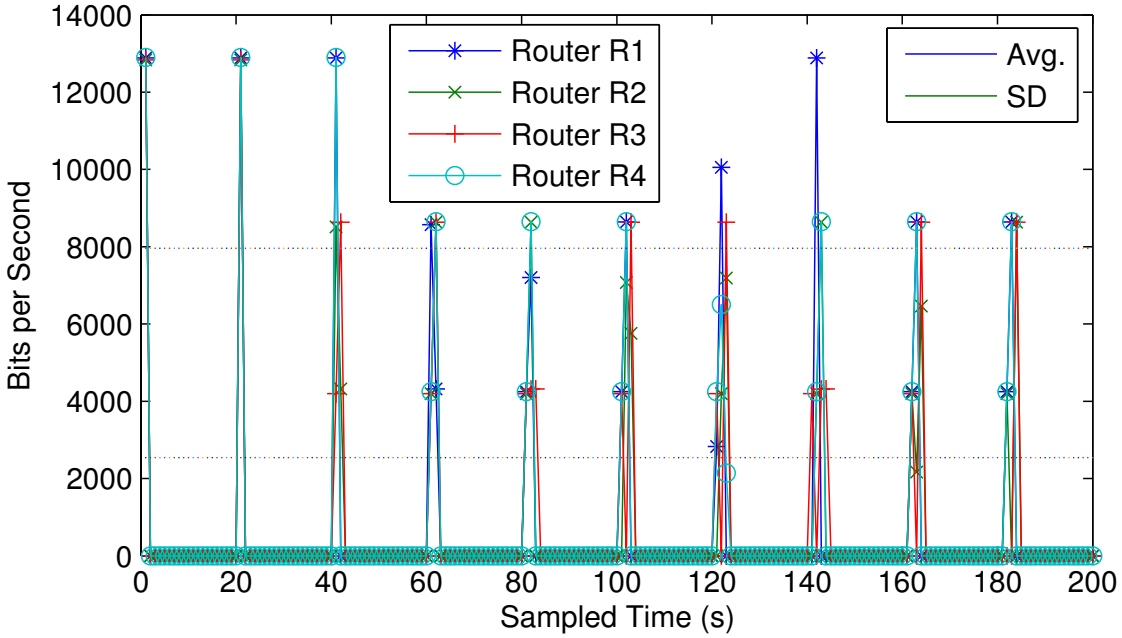


Figure D-15: First sample of the SNMP protocol based network core interface monitoring traffic load at the SNMP web application in Testbed Setup A.

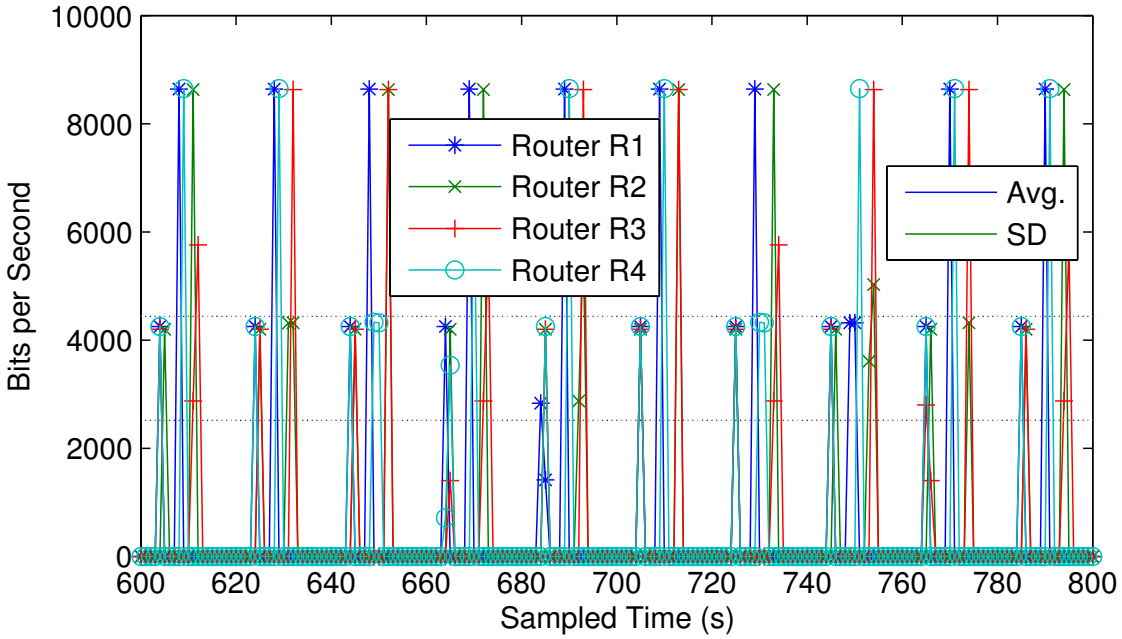


Figure D-16: Second sample of the SNMP protocol based network core interface monitoring traffic load at the SNMP web application in Testbed Setup A.

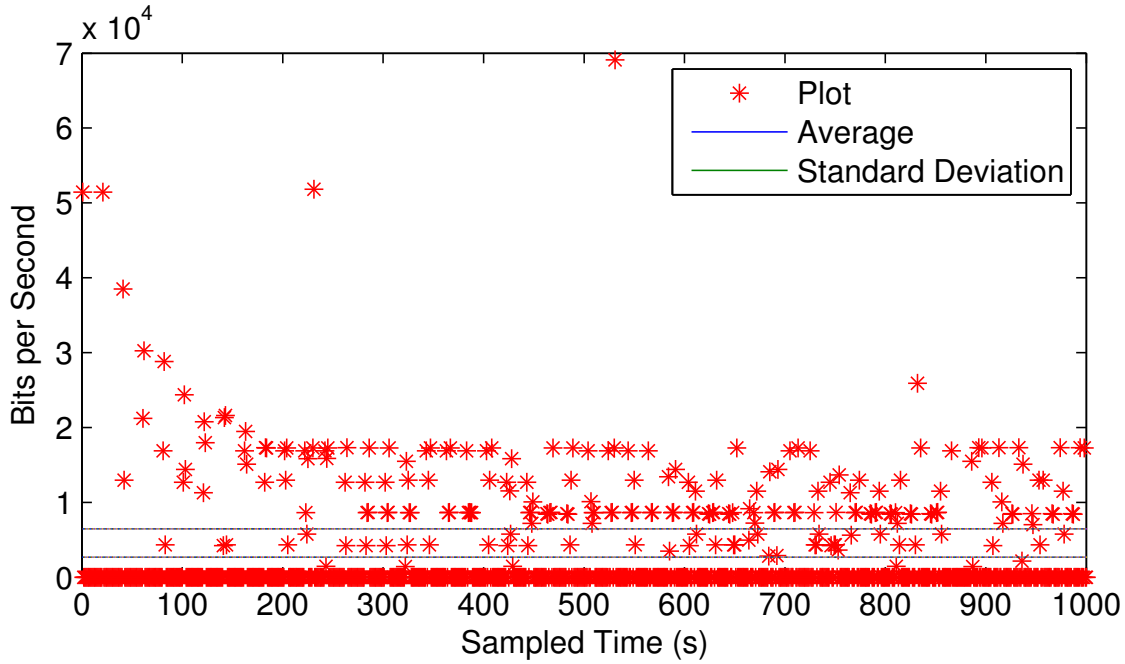


Figure D-17: Total sample of the SNMP protocol based network core interface monitoring traffic load at the SNMP web application in Testbed Setup A.

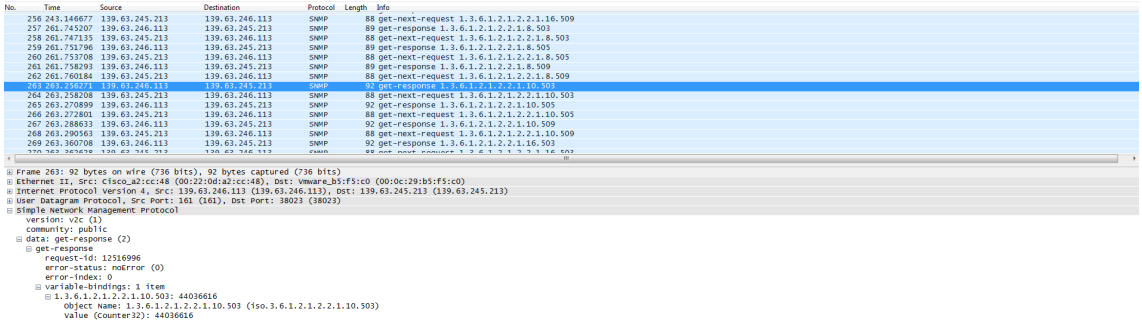


Figure D-18: Wireshark capture of SNMP interface counters polling by the SNMP web application via the SNMP protocol in Testbed Setup A.

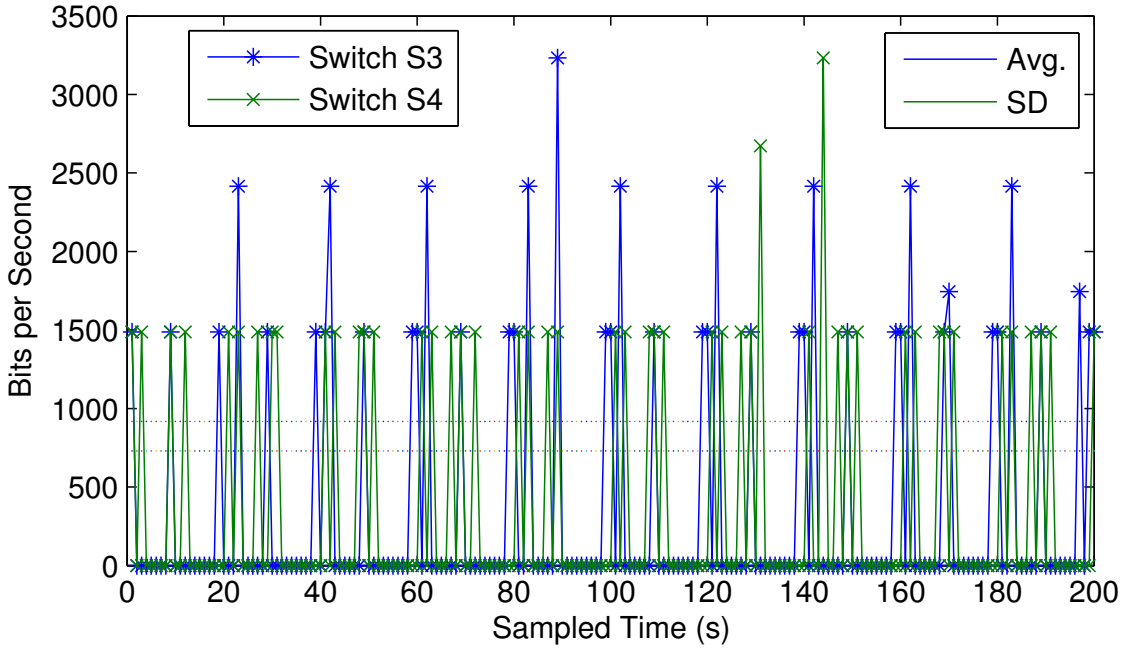


Figure D-19: First sample of the sFlow protocol based network core interface monitoring traffic load at the core sFlow-RT network analyzer in Testbed Setup B.

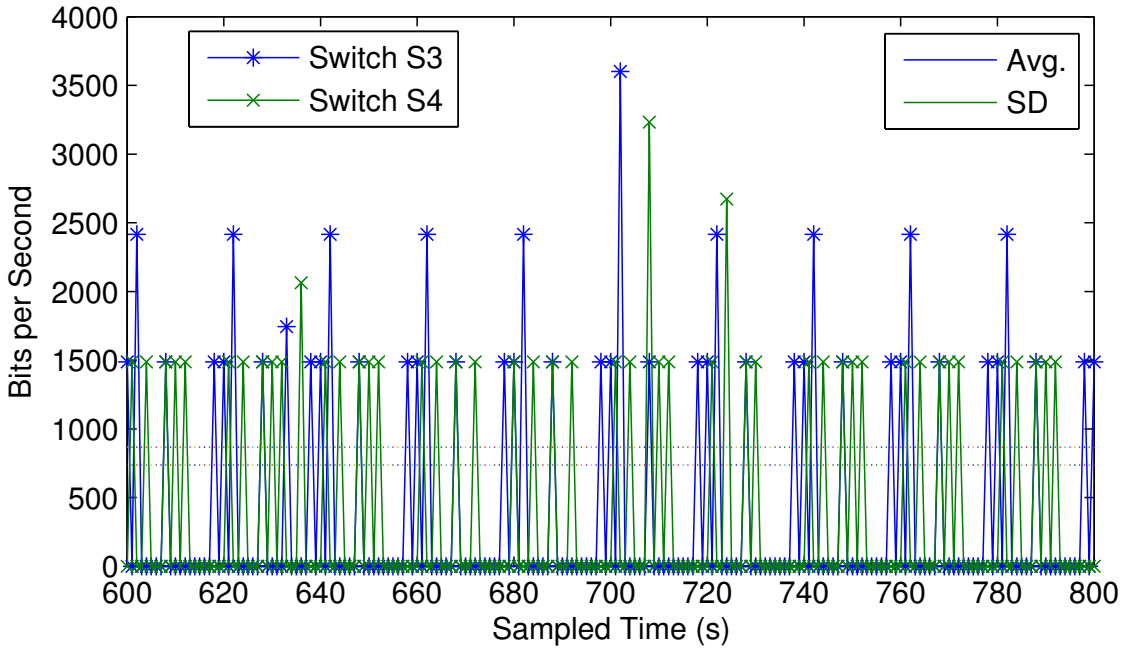


Figure D-20: Second sample of the sFlow protocol based network core interface monitoring traffic load at the core sFlow-RT network analyzer in Testbed Setup B.

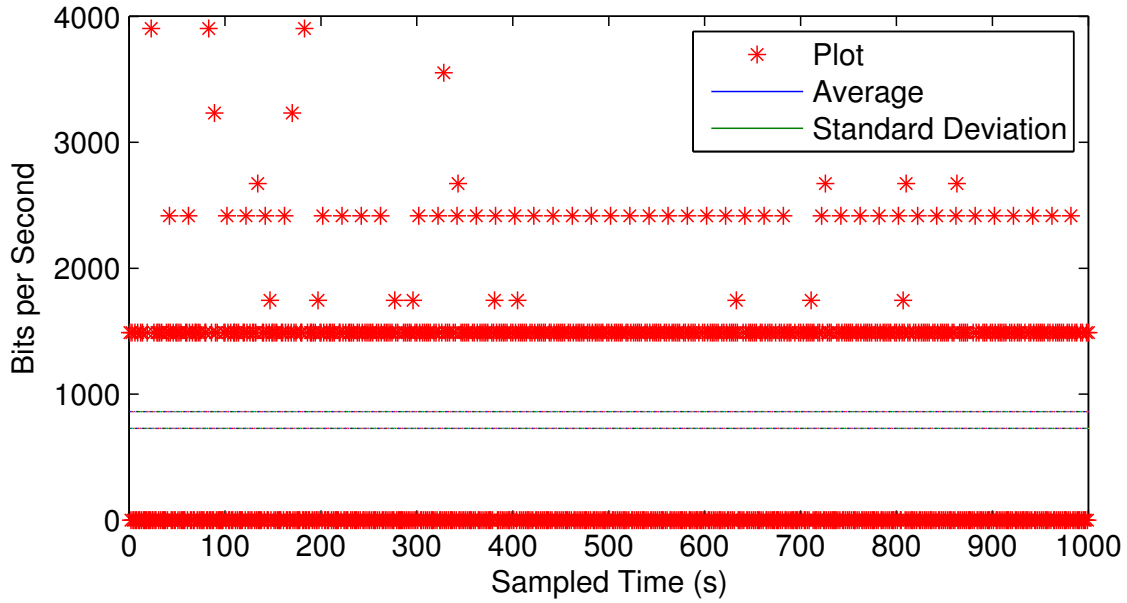


Figure D-21: Total sample of the sFlow protocol based network core interface monitoring traffic load at the core sFlow-RT network analyzer in Testbed Setup B.

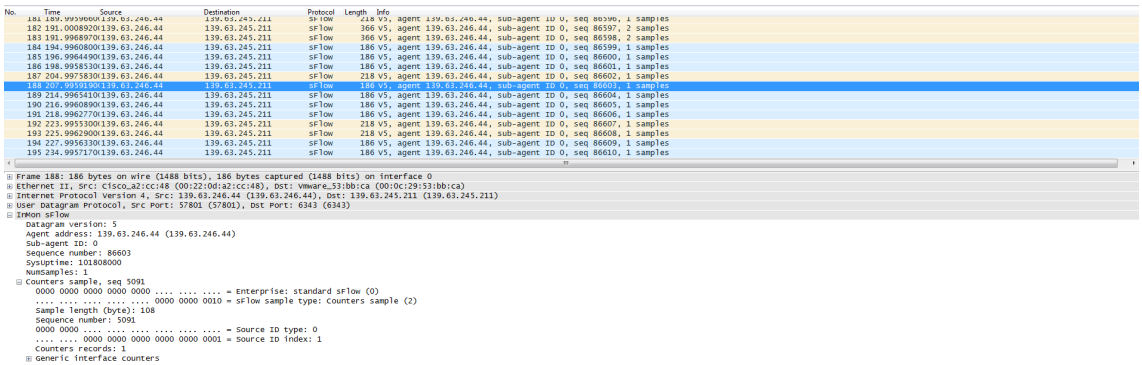


Figure D-22: Wireshark capture of sFlow interface counters polling at the sFlow-RT network analyzer via the sFlow protocol in the PoC physical network testbed.

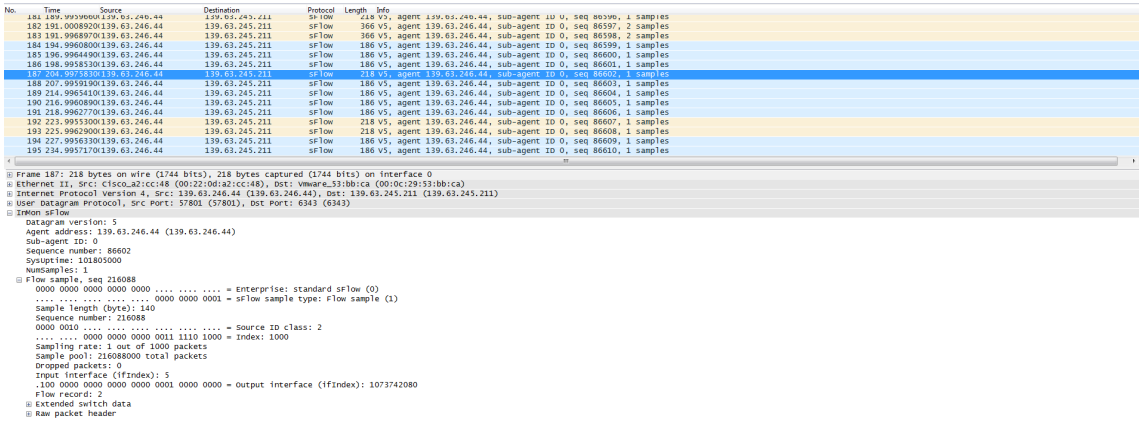


Figure D-23: Wireshark capture of sFlow sampled flows polling at the sFlow-RT network analyzer via the sFlow protocol in the PoC physical network testbed.

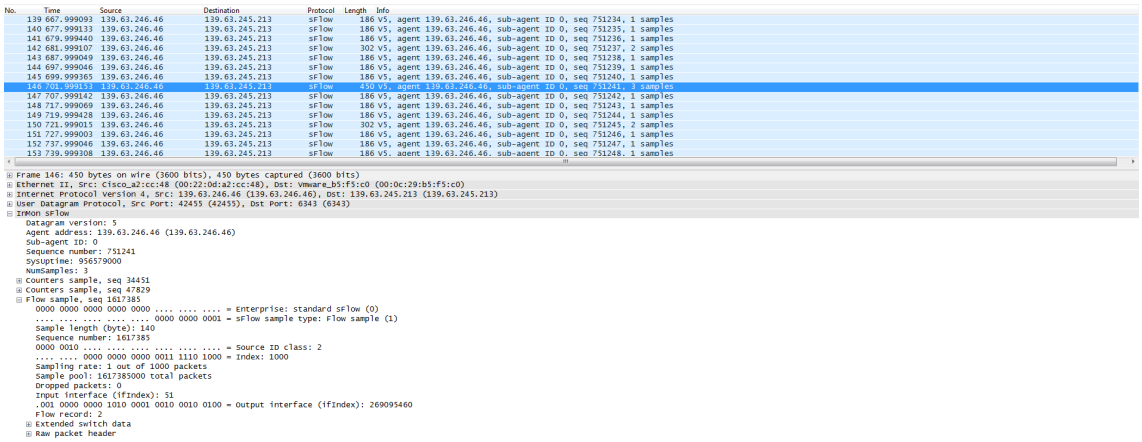


Figure D-24: Wireshark capture of combined sFlow interface counters and sampled flows polling at the sFlow-RT network analyzer via the sFlow protocol in the PoC physical network testbed.

D-2 Basic Network Connectivity Services

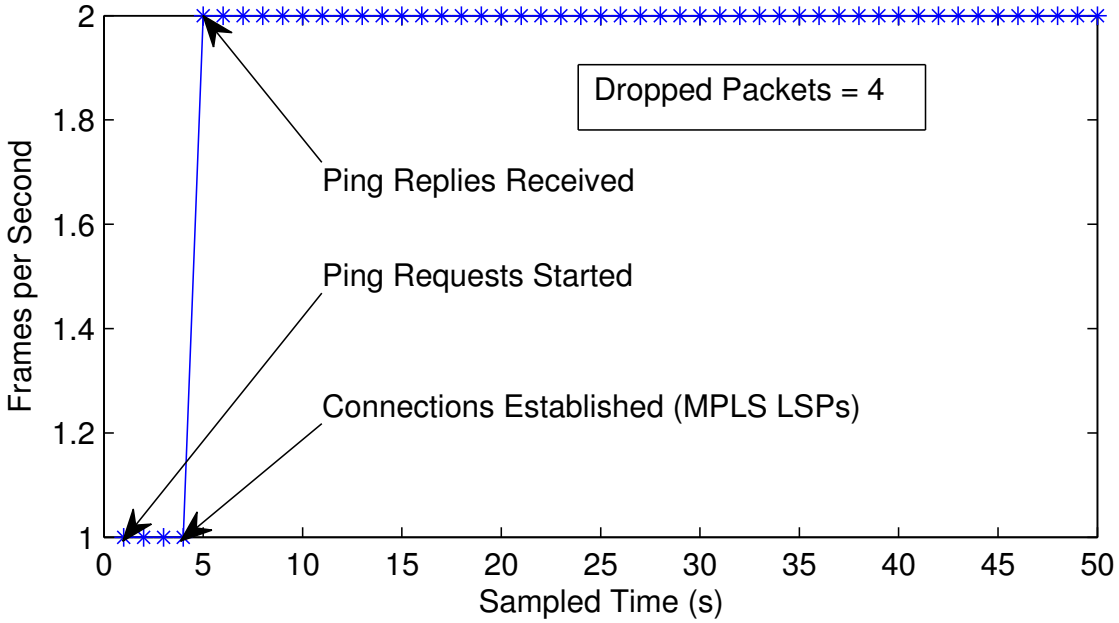


Figure D-25: Reactive basic connectivity service performance analysis - frames per second.

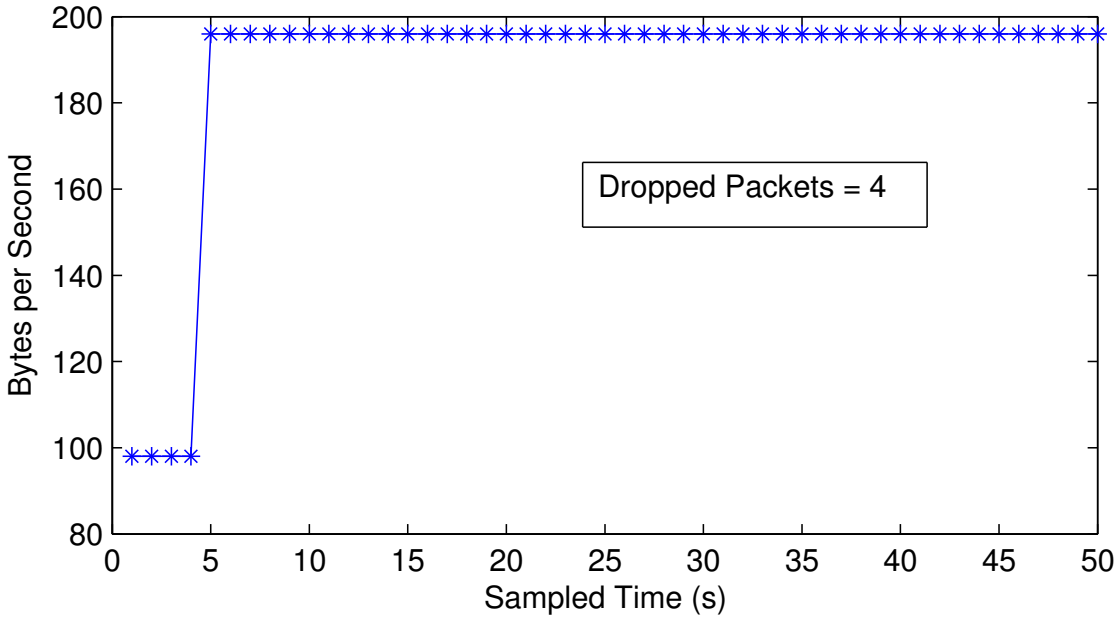


Figure D-26: Reactive basic connectivity service performance analysis - bytes per second.

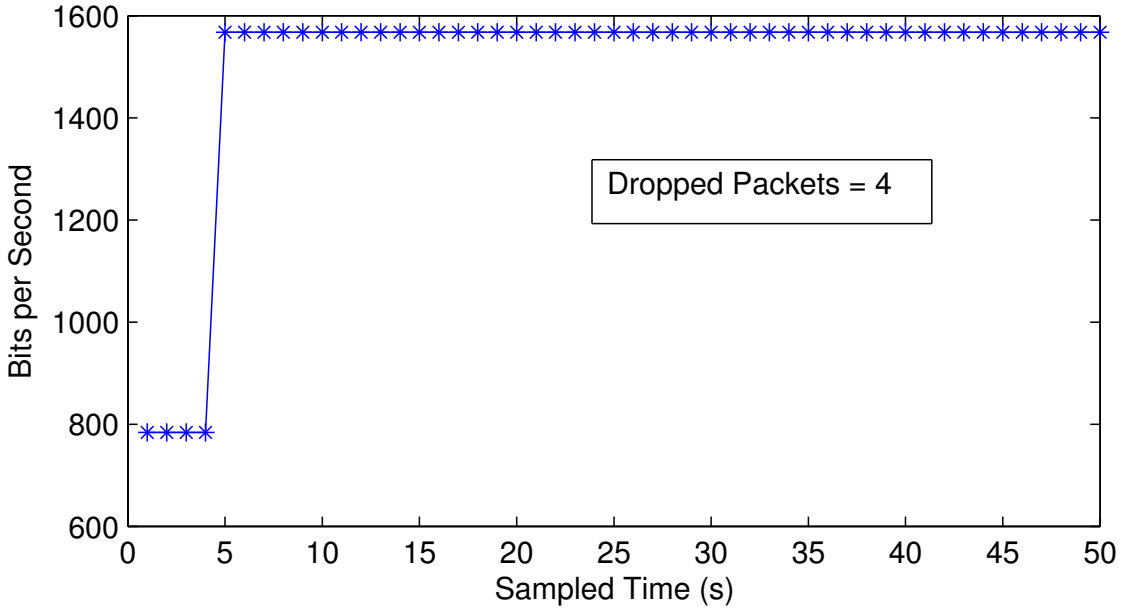


Figure D-27: Reactive basic connectivity service performance analysis - bits per second.

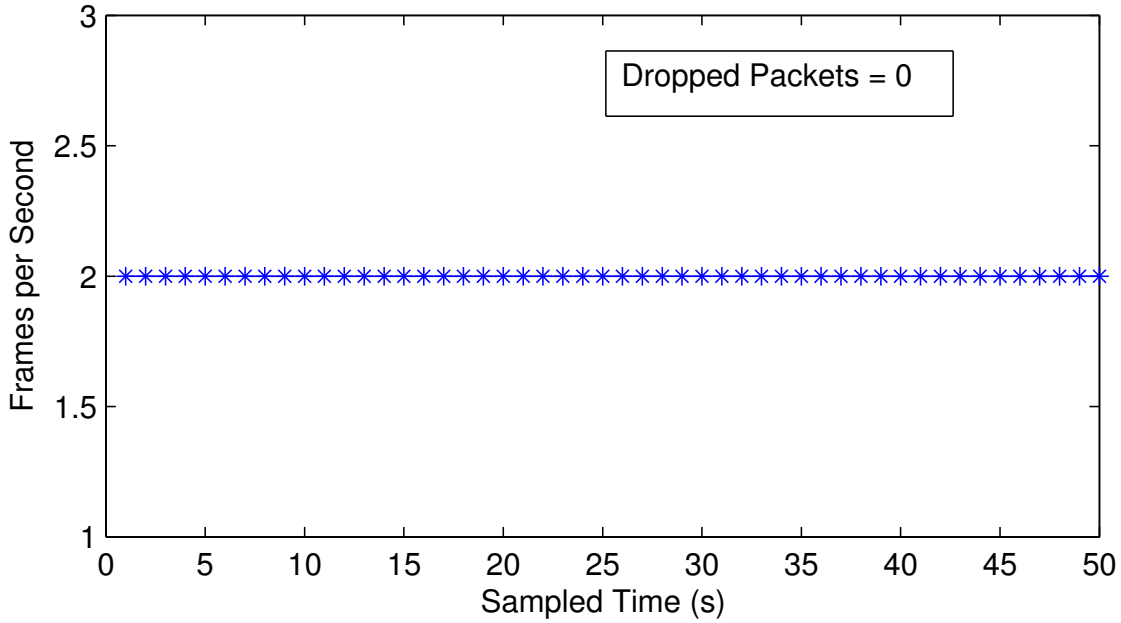


Figure D-28: Proactive basic connectivity service performance analysis - frames per second.

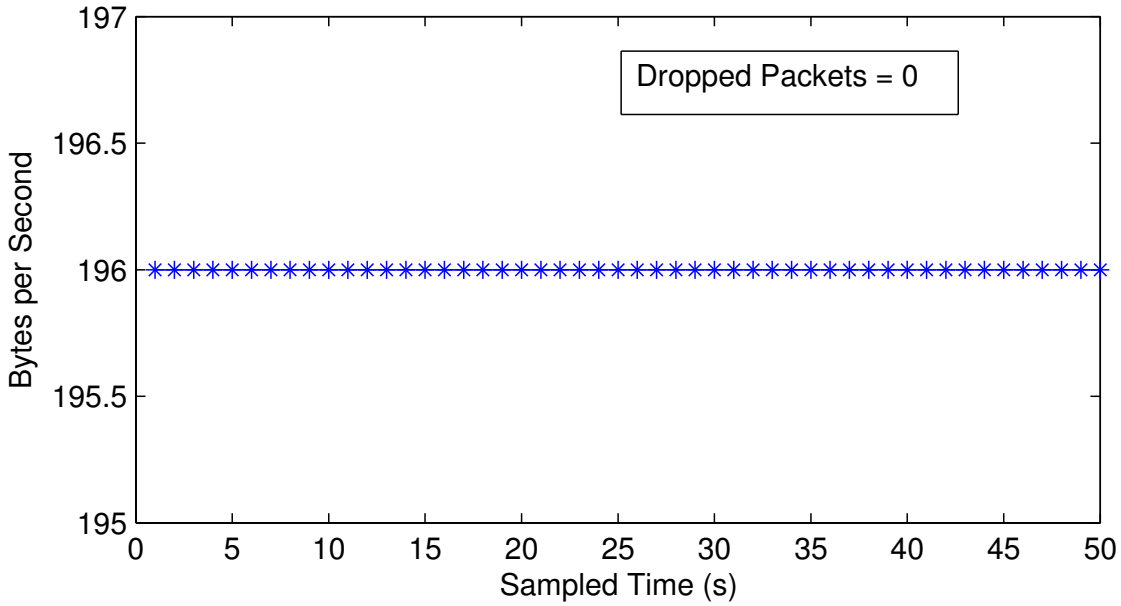


Figure D-29: Proactive basic connectivity service performance analysis - bytes per second.

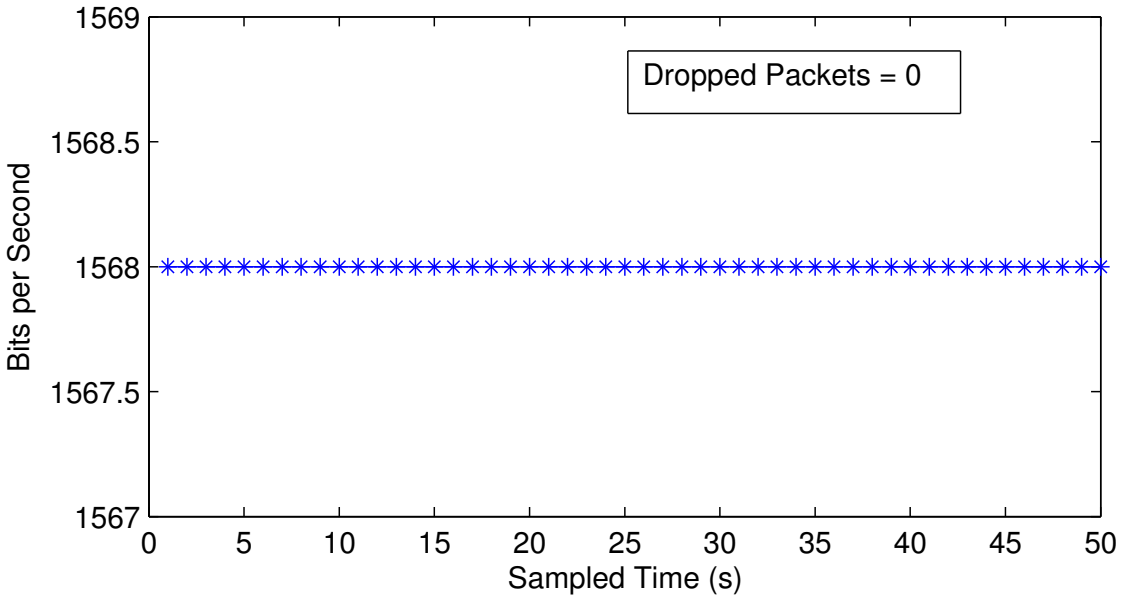


Figure D-30: Proactive basic connectivity service performance analysis - bits per second.

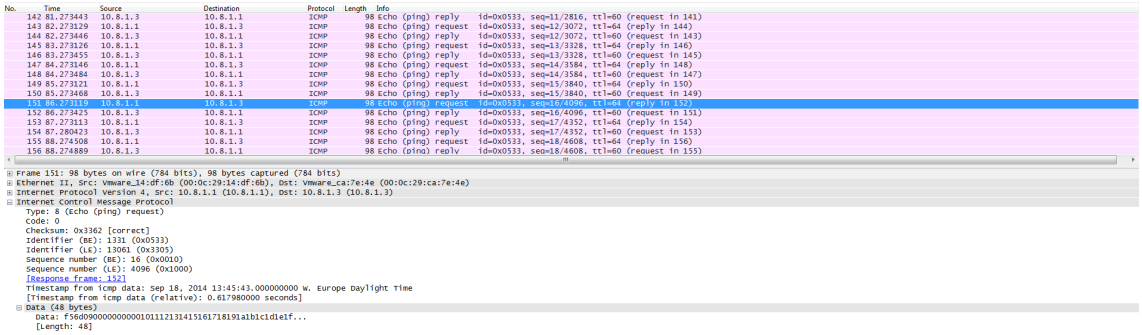


Figure D-31: Wireshark capture of ping echo request during basic network connectivity service in the PoC physical network testbed.

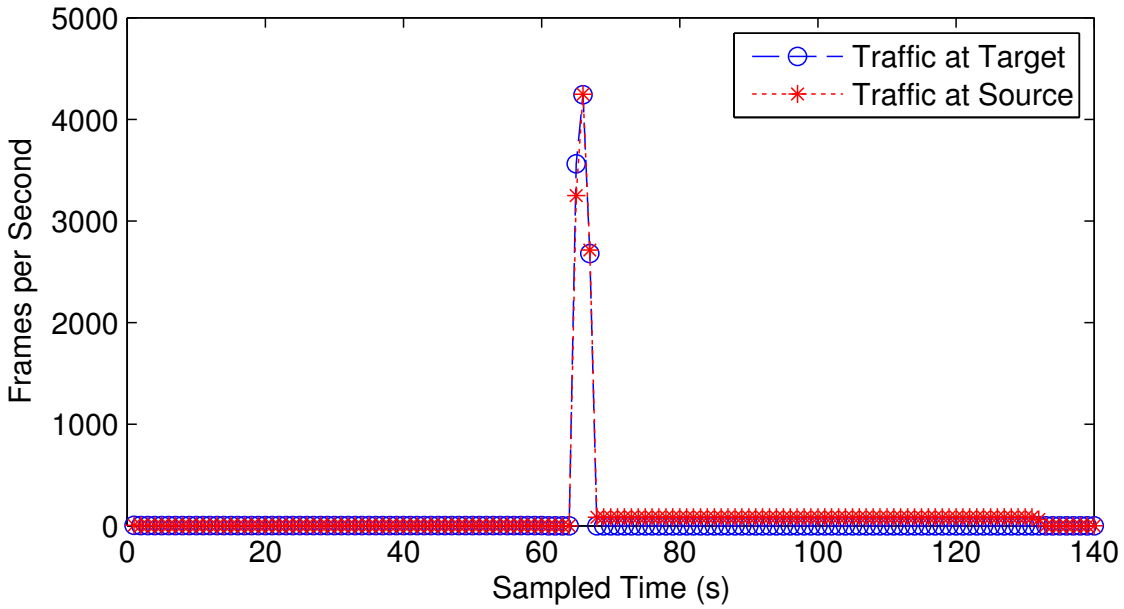


Figure D-32: Basic ping flood based DoS attack and its mitigation.

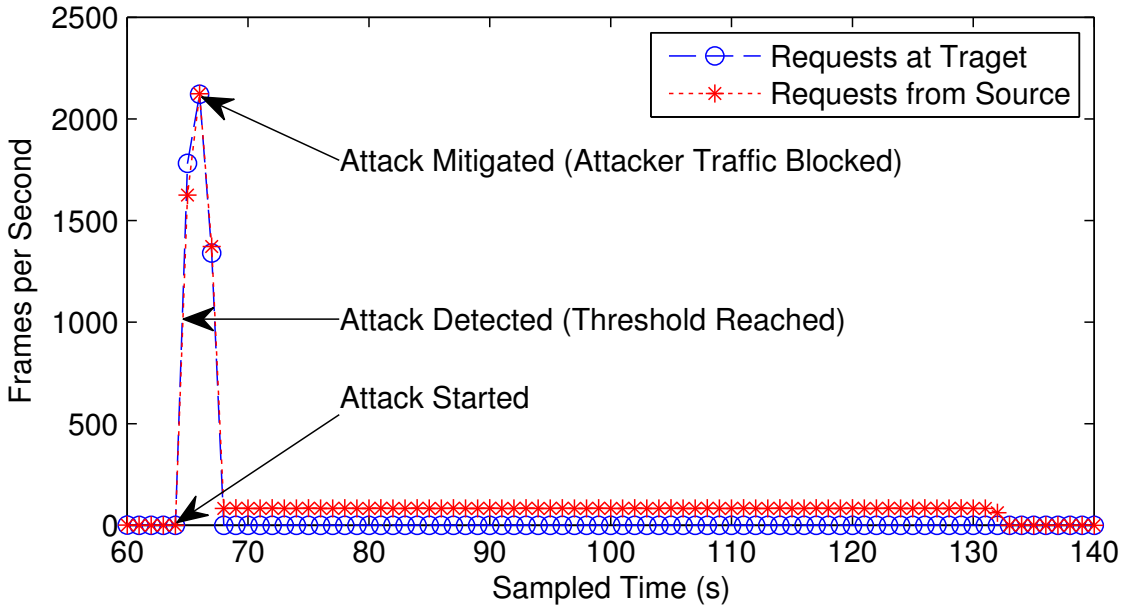


Figure D-33: Basic ping flood based DoS attack and its mitigation - ping flood requests.

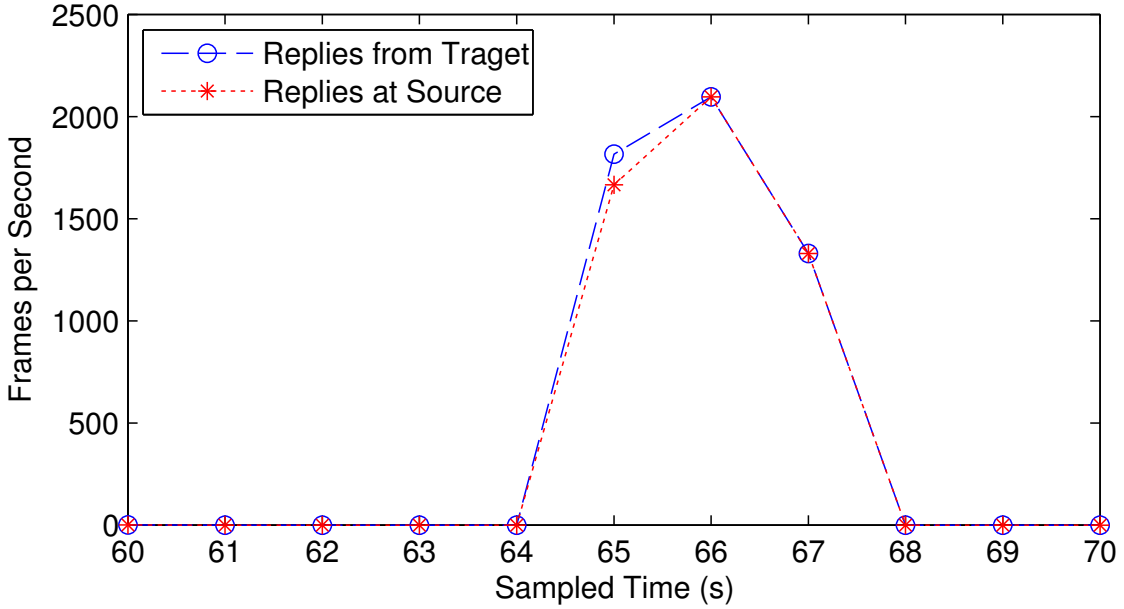


Figure D-34: Basic ping flood based DoS attack and its mitigation - ping flood replies.

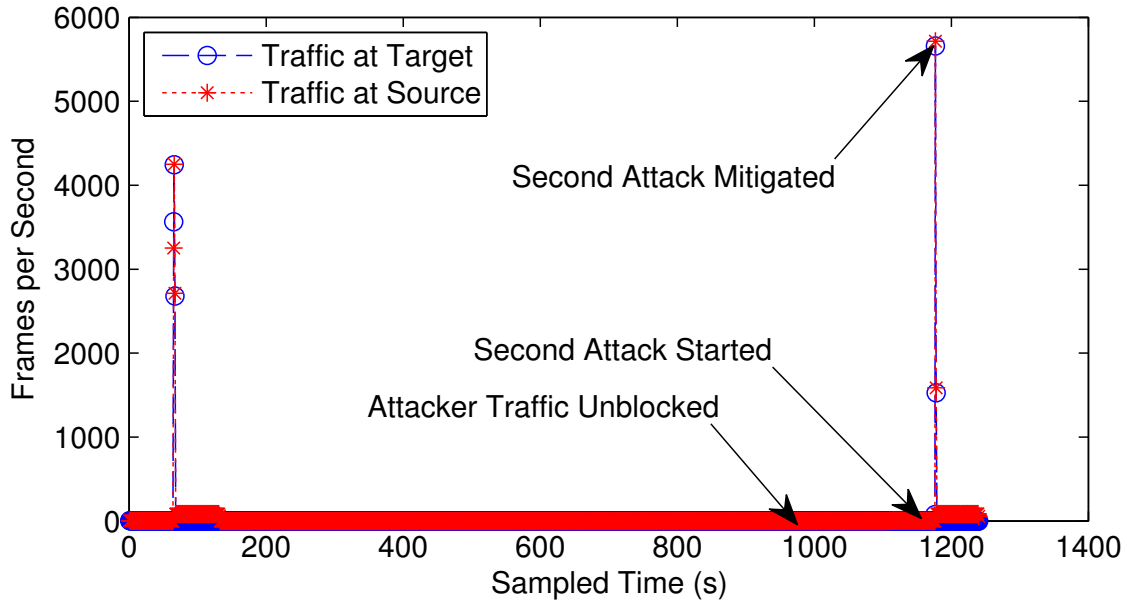


Figure D-35: Second basic ping flood based DoS attack and its mitigation.

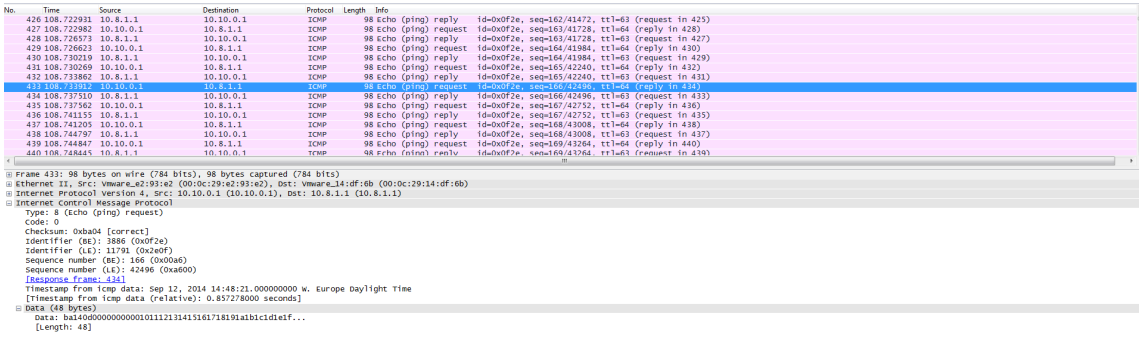


Figure D-36: Wireshark capture of ping flood echo request during basic network connectivity service in the PoC physical network testbed.

Bibliography

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

[2] Q. Duan, Y. Yan, and A. V. Vasilakos, "A survey on service-oriented network virtualization toward convergence of networking and cloud computing," *Network and Service Management, IEEE Transactions on*, vol. 9, no. 4, pp. 373–392, 2012.

[3] Open Networking Foundation, "Software-defined networking: the new norm for networks," ONF Paper, Open Networking Foundation, Apr. 2012. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf

[4] ——, "Openflow switch specification: version 1.3.0 (wire protocol 0x04)," OpenFlow, Open Networking Foundation, June 2012. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf

[5] AT&T et al., "Network functions virtualisation (nfv): an introduction, benefits, enablers, challenges & call for action," Network Functions Virtualisation - Introductory White Paper, Oct. 2012. [Online]. Available: https://portal.etsi.org/nfv/nfv_white_paper.pdf

[6] ——, "Network functions virtualisation (nfv): network operator perspectives on industry progress," Network Functions Virtualisation - Update White Paper, Oct. 2013. [Online]. Available: http://portal.etsi.org/NFV/NFV_White_Paper2.pdf

[7] R. Braden, "Requirements for internet hosts – communication layers," RFC 1122 (INTERNET STANDARD), Internet Engineering Task Force, Oct. 1989, updated by RFCs 1349, 4379, 5884, 6093, 6298, 6633, 6864. [Online]. Available: http://www.ietf.org/rfc/rfc1122.txt

[8] ——, "Requirements for internet hosts – application and support," RFC 1123 (INTERNET STANDARD), Internet Engineering Task Force, Oct. 1989, updated by RFCs 1349, 2181, 5321, 5966. [Online]. Available: http://www.ietf.org/rfc/rfc1123.txt

[9] E. Rosen, A. Viswanathan, R. Callon, "Multiprotocol label switching architecture," RFC 3031 (PROPOSED STANDARD), Internet Engineering Task Force, Jan. 2001, updated by RFCs 6178, 6790. [Online]. Available: http://www.ietf.org/rfc/rfc3031.txt

[10] "How to port open vswitch to new software or hardware," GitHub Project, Open vSwitch. [Online]. Available: https://github.com/homework/openvswitch/blob/master/PORTING

[11] "Opendaylight "hydrogen" base edition," Linux Foundation Collaborative Project, OpenDaylight. [Online]. Available: https://wiki.opendaylight.org/view/Release/Hydrogen/Base/User_Guide

[12] ICT Data and Statistics Division - Telecommunication Development Bureau, "The world in 2014: ict facts and figures," Informational Brochure, International Telecommunication Union, Apr. 2014. [Online]. Available: http://www.itu.int/en/ITU-D/Statistics/Documents/facts/ICTFactsFigures2014-e.pdf

[13] D. Evans, "The internet of things: how the next evolution of the internet is changing everything," White Paper, Cisco Systems, Inc, Apr. 2011. [Online]. Available: http://www.cisco.com/web/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf

[14] Cisco Visual Networking Index (VNI), "The zettabyte era: trends and analysis," White Paper, Cisco Systems, Inc, June 2013. [Online]. Available: http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/VNI_Hyperconnectivity_WP.pdf

[15] S. Shenker, "Fundamental design issues for the future internet," *Selected Areas in Communications, IEEE Journal on*, vol. 13, no. 7, pp. 1176–1188, 1995.

[16] J. S. Turner and D. E. Taylor, "Diversifying the internet," in *Global Telecommunications Conference, 2005. GLOBECOM'05. IEEE*, vol. 2. IEEE, 2005, pp. 6–pp.

[17] C. Jinzhou, W. Chunming, J. Ming, and Z. Dong, "A review of future internet research programs and possible trends," in *Wireless Communications Networking and Mobile Computing (WiCOM), 2010 6th International Conference on*. IEEE, 2010, pp. 1–4.

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

[19] "Openflow: an open standard for sdn southbound interface," SDN Resources, Open Networking Foundation. [Online]. Available: https://www.opennetworking.org/sdn-resources/openflow

[20] Q. Duan, "Network-as-a-service in software-defined networks for end-to-end qos provisioning," in *Wireless and Optical Communication Conference (WOCC), 2014 23rd*, May 2014, pp. 1–5.

[21] I. Bueno, J. I. Aznar, E. Escalona, J. Ferrer, and J. A. Garcia-Espin, "An opennaas based sdn framework for dynamic qos control," in *Future Networks and Services (SDN4FNS), 2013 IEEE SDN for*. IEEE, 2013, pp. 1–7.

[22] R. Gouveia, J. Aparicio, J. Soares, B. Parreira, S. Sargento, and J. Carapinha, "Sdn framework for connectivity services," in *Communications (ICC), 2014 IEEE International Conference on*. IEEE, 2014, pp. 3058–3063.

[23] A. R. Sharafat, S. Das, G. Parulkar, and N. McKeown, "Mpls-te and mpls vpns with openflow," in *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4. ACM, 2011, pp. 452–453.

[24] R. Muñoz, R. Casellas, R. Martínez, and R. Vilalta, "Pce: what is it, how does it work and what are its limitations?," *Journal of Lightwave Technology*, vol. 32, no. 4, pp. 528–543, 2014.

[25] M. Zimmerman, D. Allan, M. Cohn, N. Damouny, C. Kolias, J. Maguire, S. Manning, D. McDysan, E. Roch, and M. Shirazipou, "Openflow-enabled sdn and network functions virtualization," ONF Solution Brief, Open Networking Foundation, Feb. 2014. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/solution-briefs/sb-sdn-nvf-solution.pdf

[26] J. Batalle, J. Ferrer Riera, E. Escalona, and J. A. Garcia-Espin, "On the implementation of nfv over an openflow infrastructure: routing function virtualization," in *Future Networks and Services (SDN4FNS), 2013 IEEE SDN for*. IEEE, 2013, pp. 1–6.

[27] H. Masutani, Y. Nakajima, T. Kinoshita, T. Hibi, H. Takahashi, K. Obana, K. Shimano, and M. Fukui, "Requirements and design of flexible nfv network infrastructure node leveraging sdn/openflow," in *Optical Network Design and Modeling, 2014 International Conference on*. IEEE, 2014, pp. 258–263.

[28] G. Hampel, M. Steiner, and T. Bu, "Applying software-defined networking to the telecom domain," in *Computer Communications Workshops (INFOCOM WKSHPS), 2013 IEEE Conference on*. IEEE, 2013, pp. 133–138.

[29] J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, A. R. Curtis, and S. Banerjee, "Devoflow: cost-effective flow management for high performance enterprise networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. ACM, 2010, p. 1.

[30] N. M. K. Chowdhury and R. Boutaba, "Network virtualization: state of the art and research challenges," *Communications Magazine, IEEE*, vol. 47, no. 7, pp. 20–26, 2009.

[31] N. Feamster, J. Rexford, and E. Zegura, "The road to sdn: an intellectual history of programmable networks," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 2, pp. 87–98, 2014.

[32] L. Yang, R. Dantu, T. Anderson, R. Gopal, "Forwarding and control element separation (forces) framework," RFC 3746 (INFORMATIONAL), Internet Engineering Task Force, Apr. 2004. [Online]. Available: https://tools.ietf.org/rfc/rfc3746.txt

[33] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and J. van der Merwe, "Design and implementation of a routing control platform," in *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*. USENIX Association, 2005, pp. 15–28.

[34] A. Farrel, J.-P. Vasseur, J. Ash, "A path computation element (pce)-based architecture," RFC 4655 (INFORMATIONAL), Internet Engineering Task Force, Aug. 2006. [Online]. Available: http://tools.ietf.org/rfc/rfc4655.txt

[35] "Nsx: the network virtualization and security platform for the software-defined data center," Products, VMware. [Online]. Available: https://www.vmware.com/products/nsx/

[36] B. Pfaff, J. Pettit, K. Amidon, M. Casado, T. Koponen, and S. Shenker, "Extending networking into the virtualization layer," in *Hotnets*, 2009.

[37] "Open vswitch: an open virtual switch," GitHub Web Front-End and Project, Open vSwitch. [Online]. Available: http://git.openvswitch.org

[38] "Cisco application centric infrastructure," Data Ceneter and Virtualization Solutions, Cisco. [Online]. Available: http://www.cisco.com/c/en/us/solutions/data-center-virtualization/application-centric-infrastructure/index.html

[39] "Onrc," Research Program, Open Networking Research Center. [Online]. Available: http://onrc.net/

[40] "Openflow switch technical specifications," Technical Library, Open Networking Foundation. [Online]. Available: https://www.opennetworking.org/sdn-resources/technical-library

[41] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "Nox: towards an operating system for networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 3, pp. 105–110, 2008.

[42] "Floodlight," Open SDN Controller, Big Switch Networks. [Online]. Available: http://www.projectfloodlight.org/floodlight/

[43] "Opendaylight: the project," Linux Foundation Collaborative Project, OpenDaylight. [Online]. Available: http://www.opendaylight.org/project

[44] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. ACM, 2010, p. 19.

[45] "Mininet: an instant virtual network on your laptop," Network Emulator, Mininet. [Online]. Available: http://mininet.org/

[46] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, "Flowvisor: A network virtualization layer," *OpenFlow Switch Consortium, Tech. Rep*, 2009.

[47] "Flowvisor: a transparent proxy between openflow switches and multiple openflow controllers," OpenFlow Controller, Open Networking Lab. [Online]. Available: https://openflow.stanford.edu/display/DOCS/Flowvisor

[48] M. Jammal, T. Singh, A. Shami, R. Asal, and Y. Li, "Software-defined networking: state of the art and research challenges," *arXiv preprint arXiv:1406.0124*, 2014.

[49] B. J. van Asten, N. L. van Adrichem, and F. A. Kuipers, "Scalability and resilience of software-defined networking: an overview," *arXiv preprint arXiv:1408.6760*, 2014.

[50] D. Harrington, R. Presuhn, B. Wijnen, "An architecture for describing simple network management protocol (snmp) management frameworks," RFC 3411 (INTERNET STANDARD), Internet Engineering Task Force, Dec. 2002, updated by RFCs 5343, 5590 and Obsoletes RFC 2571. [Online]. Available: http://www.ietf.org/rfc/rfc3411.txt

[51] B. Claise, B. Trammell, P. Aitken, "Specification of the ip flow information export (ipfix) protocol for the exchange of flow information," RFC 7011 (INTERNET STANDARD), Internet Engineering Task Force, Sep. 2013, obsoletes RFC 5101. [Online]. Available: http://www.ietf.org/rfc/rfc7011.txt

[52] B. Claise, "Cisco systems netflow services export version 9," RFC 3954 (INFORMATIONAL), Internet Engineering Task Force, Oct. 2004. [Online]. Available: http://www.ietf.org/rfc/rfc3954.txt

[53] P. Phaal, S. Panchen, N. McKee, "Inmon corporation's sflow: a method for monitoring traffic in switched and routed networks," RFC 3176 (INFORMATIONAL), Internet Engineering Task Force, Sep. 2001. [Online]. Available: https://www.ietf.org/rfc/rfc3176.txt

[54] N. Van Adrichem, C. Doerr, and F. Kuipers, "Opennetmon: Network monitoring in openflow software-defined networks," in *Network Operations and Management Symposium (NOMS), 2014 IEEE*, May 2014, pp. 1–8.

[55] R. Enns, M. Bjorklund, J. Schoenwaelder, A. Bierman, "Network configuration protocol (netconf)," RFC 6241 (PROPOSED STANDARD), Internet Engineering Task Force, June 2011, obsoletes RFC 4741. [Online]. Available: https://www.ietf.org/rfc/rfc6241.txt

[56] B. Pfaff, B. Davie, "The open vswitch database management protocol," RFC 7047 (INFORMATIONAL), Internet Engineering Task Force, Dec. 2013, independent Submission. [Online]. Available: https://www.ietf.org/rfc/rfc7047.txt

[57] Open Networking Foundation, "Of-config 1.2: openflow management and configuration protocol," OpenFlow Config, Open Networking Foundation, 2014. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow-config/of-config-1.2.pdf

[58] "sflow-rt: sflow based network analyzer," Products, InMon Corp. [Online]. Available: http://www.inmon.com/products/sFlow-RT.php

[59] "sflow: sdn analytics and control using sflow standard," Web Blog, InMon Corp. [Online]. Available: http://blog.sflow.com/

[60] "Bottle: python web framework," Python Module, Bottle. [Online]. Available: http://bottlepy.org/docs/dev/index.html

[61] "Pysnmp: snmp library for python," Python Module, The PySNMP Project. [Online]. Available: http://pysnmp.sourceforge.net/

[62] "Requests: http library for humans," Python Module, Requests. [Online]. Available: http://docs.python-requests.org/en/latest/

[63] T. Bray, "The javascript object notation (json) data interchange format," RFC 7159 (PROPOSED STANDARD), Internet Engineering Task Force, Mar. 2014, obsoletes RFCs 4627, 7158. [Online]. Available: http://www.ietf.org/rfc/rfc7159.txt

[64] "Delft university of technology - network architectures and services," GitHub Profile. [Online]. Available: https://github.com/TUDelftNAS/

[65] "Pica8: open networking," 1 GbE / 10 GbE / 40 GbE Open Switches, Pica8. [Online]. Available: http://www.pica8.com/open-switching/1gbe-10gbe-40gbe-open-switches.php

[66] "Juniper networks: m10i router description," TechLibrary, Juniper Networks. [Online]. Available: http://www.juniper.net/techpubs/en_US/release-independent/junos/information-products/pathway-pages/m-series/m10i/

[67] "vsphere: the world's leading server virtualization platform," Products, VMware. [Online]. Available: http://www.vmware.com/products/vsphere

[68] "Networkx: high-productivity software for complex networks," GitHub Web Front-End and Python Software Package, NetworkX. [Online]. Available: https://networkx.github.io/

[69] "matplotlib: python plotting," Python 2D Plotting Library, matplotlib. [Online]. Available: http://matplotlib.org/

[70] P. Van Mieghem and F. A. Kuipers, "Concepts of exact qos routing algorithms," *Networking, IEEE/ACM Transactions on*, vol. 12, no. 5, pp. 851–864, 2004.

[71] "Wireshark v1.12.0," Release Notes, Wireshark. [Online]. Available: https://www.wireshark.org/docs/relnotes/wireshark-1.12.0.html

[72] "Matlab: the language of technical computing," High-Level Language and Interactive Environment, MathWorks. [Online]. Available: http://nl.mathworks.com/products/matlab/

[73] V. Bernat, "lldpd: implementation of lldp for various unixes," GitHub Web Front-End and Project. [Online]. Available: http://vincentbernat.github.io/lldpd/

[74] "ping, ping6 - send icmp datagrams to network hosts," Ubuntu Manpage, Ubuntu. [Online]. Available: http://manpages.ubuntu.com/manpages/trusty/man8/ping.8.html

Glossary

List of Abbreviations

ARP Address Resolution Protocol

App Application

API Application Programming Interface

ASIC Application-Specific Integrated Circuit

ATM Asynchronous Transfer Mode

BGP Border Gateway Protocol

BSS Business Support System

CAPEX Capital Expenditure

CPU Central Processing Unit

CLI Command Line Interface

CDN Content Delivery Network

CE Customer Edge

DiffServ Integrated Services

EMS Element Management System

DoS Denial of Service

DSL Digital Subscriber Line

DDoS Distributed Denial of Service

DNS Domain Name System

ETSI European Telecommunications Standards Institute

FPGA Field-Programmable Gate Array

FTP File Transfer Protocol

ForCES Forwarding and Control Element Separation

FEC Forwarding Equivalence Class

FIB Forwarding Information Base

4G Fourth Generation

GRE Generic Routing Encapsulation

Gb Gigabit

GbE Gigabit Ethernet

GUI Graphical User Interface

HTTP Hypertext Transfer Protocol

ICT Information and Communications Technology

IT Information Technology

IaaS Infrastructure as a Service

IEEE Institute of Electrical and Electronics Engineers

IntServ Integrated Services

IPC Inter-Process Communication

IF-MIB Interfaces-Management Information Bases

ICMP Internet Control Message Protocol

IETF Internet Engineering Task Force

IGMP Internet Group Management Protocol

IoT Internet of Things

IP Internet Protocol

IPFIX Internet Protocol Flow Information Export

IPv6 Internet Protocol version 6

JSON Java Script Object Notation

KVM Kernel-based Virtual Machine

Kbps Kilobits per second

LDP Label Distribution Protocol

LER Label Edge Router

LFIB Label Forwarding Information Base

LIB Label Information Base

LSP Label Switched Path

LSR Label Switching Router

L2/L3 Layer 2 or Layer 3

L2-L4 Layer 2 to Layer 4

L2TP Layer 2 Tunneling Protocol

L4-L7 Layer 4 to Layer 7

LACP Link Aggregation Control Protocol

LLDP Link Layer Discovery Protocol

LAN Local Area Network

LTE Long Term Evolution

M2M Machine-to-Machine

MIB Management Information Base

MAC Medium Access Control

MPLS Multiprotocol Label Switching

NAT Network Address Translation

NETCONF Network Configuration Protocol

NFV Network Function Virtualization

NFV ISG Network Functions Virtualisation Industry Specification Group

NFVI Network Functions Virtualisation Infrastructure

NFV M&O Network Functions Virtualisation Management and Orchestration

NVP Network Virtualization Platform

NaaS Network-as-a-Service

NoSQL Not Only SQL

ONF Open Networking Foundation

ONRC Open Networking Research Center

OSPF Open Shortest Path First

OVS Open vSwitch

OVSDB Open vSwitch Database Management Protocol

OF-CONFIG OpenFlow Management and Configuration Protocol

OS Operating System

OPEX Operational Expenditure

OSS Operations support systems

OTT Over-The-Top

PCE Path Computational Element

PHP Penultimate Hop Popping

PC Personal Computer

PaaS Platform as a Service

PPTP Point-to-Point Tunneling Protocol

PoC Proof-of-Concept

P Provider

PE Provider Edge

QoS Quality of Service

RSPAN Remote Switched Port ANalyzer

REST REpresentational State Transfer

RFC Request for Comments

RSVP Resource Reservation Protocol

RSVP-TE Resource Reservation Protocol-Traffic Engineering

ROI Return On Investment

RTD Round Trip Delay

RTT Round Trip Time

RCP Routing Control Platform

SAL Service Abstraction Layer

SLA Service Level Agreement

SOA Service-Oriented Architecture

SMTP Simple Mail Transfer Protocol

SNMP Simple Network Management Protocol

SaaS Software as a Service

SDN Software-Defined Networking

SCTP Stream Control Transmission Protocol

SPAN Switched Port ANalyzer

TTL Time To Live

TC Traffic Class

TCP Transmission Control Protocol

TCP/IP Transmission Control Protocol/Internet Protocol

TLS/SSL Transport Layer Security/Secure Sockets Layer

UDP User Datagram Protocol

VXLAN Virtual Extensible Local Area Network

VLAN Virtual Local Area Network

VM Virtual Machine

VPN Virtual Private Network

VNF Virtualised Network Function

WAN Wide Area Network

WWW World Wide Web