PUCRS

FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
MESTRADO EM CIÊNCIA DA COMPUTAÇÃO

RAFAEL ANTON EICHELBERGER

SFC PATH TRACER: A TROUBLESHOOTING TOOL FOR SERVICE FUNCTION CHAINING

Porto Alegre

2017

PÓS-GRADUAÇÃO - *STRICTO SENSU*

Pontifícia Universidade Católica
do Rio Grande do Sul

PONTIFICAL CATHOLIC UNIVERSITY OF RIO GRANDE DO SUL
FACULTY OF INFORMATICS
GRADUATE PROGRAM IN COMPUTER SCIENCE

SFC PATH TRACER: A TROUBLESHOOTING TOOL FOR SERVICE FUNCTION CHAINING

RAFAEL ANTON EICHELBERGER

Dissertation presented as partial requirement for obtaining the degree of Master in Computer Science at Pontifical Catholic University of Rio Grande do Sul.

Advisor: Prof. Tiago Ferreto

Porto Alegre
2017

Ficha Catalográfica

E34 s Eichelberger, Rafael Anton

SFC Path Tracer : a troubleshooting tool for service function chaining / Rafael Anton Eichelberger . – 2017.
75 f.
Dissertação (Mestrado) – Programa de Pós-Graduação em Ciência da Computação, PUCRS.

Orientador: Prof. Dr. Tiago Coelho Ferreto.

1. Service Function Chaining. 2. SFC. 3. NFV. 4. SDN. 5. Networking. I. Ferreto, Tiago Coelho. II. Título.

Elaborada pelo Sistema de Geração Automática de Ficha Catalográfica da PUCRS com os dados fornecidos pelo(a) autor(a).

Rafael Anton Eichelberger

SFC Path Tracer: A Troubleshooting Tool for Service Function Chaining

This Dissertation/Thesis has been submitted in partial fulfillment of the requirements for the degree of Doctor/Master of Computer Science, of the Graduate Program in Computer Science, School of Computer Science of the Pontifícia Universidade Católica do Rio Grande do Sul.

Sanctioned on March, 2017.

COMMITTEE MEMBERS:

Prof. Dr. Luciano Paschoal Gaspary (Institution 1)

Prof. Dr. Fernando Luis Dotti (Institution 2)

Prof. Dr. Tiago Coelho Ferreto (PPGCC/PUCRS - Advisor)

SFC PATH TRACER: A TROUBLESHOOTING TOOL FOR SERVICE FUNCTION CHAINING

RESUMO

Service Function Chaining (SFC) é um importante campo de pesquisa na área de redes de computadores, com várias propostas de diferentes métodos de encapsulamento e encaminhamento de pacotes. Os métodos de encaminhamento de pacotes usados para implementar SFC podem inviabilizar o uso de ferramentas tradicionais de depuração de rede, o que dificulta a detecção de erros de configuração ou possíveis degradações de desempenho em ambientes SFC. Este trabalho apresenta o SFC Path Tracer, uma ferramenta para detecção de problemas no domínio SFC em ambientes NFV/SDN. Essa ferramenta permite a identificação de problemas no domínio SFC, através da geração de *trace* de pacotes e medição de atrasos *intra-hop* a partir de um *SFC Path* específico. SFC Path Tracer é agnóstico em relação aos mecanismos de encapsulamento e encaminhamento usados para implementar SFC, sendo eficaz na detecção de grande parte dos problemas em um ambiente SFC.

Palavras Chave: Encadeamento de funções de rede, SFC, NFV, SDN, Rede de Computadores.

SFC PATH TRACER: A TROUBLESHOOTING TOOL FOR SERVICE FUNCTION CHAINING

ABSTRACT

Service Function Chaining (SFC) is an important research field in networking area with many encapsulation and forwarding mechanisms being proposed. To implement SFC, non-standard forwarding methods are used which break the mechanism of regular network troubleshooting tools, challenging the detection of SFC misconfiguration or performance degradation. This work presents the SFC Path Tracer, a tool for troubleshooting SFC in NFV/SDN environments. This tool enables the identification of problems in the SFC environment by generating packet trace and computing intra-hop delays from a specific SFC path. SFC Path Tracer is agnostic regarding the SFC encapsulation and forwarding mechanisms being effective to detect most problems in an SFC environment.

Keywords: Service Function Chaining, SFC, NFV, SDN, Networking.

LIST OF FIGURES

CONTENTS

1. INTRODUCTION

The development of Software-Defined Networking (SDN) and Network Function Virtualization (NFV) technologies have changed the way administrators and operators manage network configuration and service deployment. SDN decouples the control and data plane of switches and delegates forwarding decisions to a controller entity that has a more accurate view of the network [22]. NFV replaces traditional middleboxes [15], deployed in specific dedicated hardware, with virtualized network functions that run on general purpose hardware, establishing a network infrastructure that has the same dynamicity and flexibility of virtual computing environments [18].

Middlebox is defined as any intermediary device performing functions other than the normal, standard functions of packets routing. Common network topologies contain several middleboxes, proportional to the number of switches and routers [53]. These network devices perform a critical role in the deployment of network services, such as: firewalls, IDS (Intrusion Detection Systems), DPI (Deep Packet Inspection), Proxies, Gateways, NAT (Network Address Translation), and WAN optimizers. Managing these devices is complex and require specialized technical support to reach the best performance among all aggregated functionalities. Networking requires an efficient and flexible way to control and route network packets, with middleboxes being as important as switches and router devices.

The features delivered by SDN and NFV have leveraged the on-the-fly configuration of network functions interconnections known as Service Function Chaining (SFC), enabling the addition and removal of those functions according to network requirements. SFC is a prominent research field in networking [5, 8, 20] with many proposals from industry [14, 17, 58] and also from working groups [9, 24]. It is possible to implement SFC using a variety of technologies including SDN/NFV or even a new protocol stack.

Given the dynamic nature of SFC, its deployment may involve multiple configuration steps as functions might reside in different hosts or networks. Hence, configuration mistakes lead the systems to several erroneous behaviors ranging from wrong forwarding decisions to packet drop. Even when packets reach their final destination, the system might have latent problems such as traversing suboptimal paths. Moreover, a single overload hop from a chain may degrade the performance of the whole chain. Besides, the lack of troubleshooting tools makes operators spend great efforts to ensure that the network meets its intended behavior [21]. To overcome this problem, researchers and practitioners proposed the SFC Operation, Administration and Maintenance Framework (SFC-OAM) [12], an Internet draft that discusses and proposes tools to aid SFC operation.

SFC-OAM discussions evolve around trace, connectivity and performance functions. In traditional networks, these functions assist the detection of misbehavior such as packet drops, unwanted network hops, lower paths, etc. However, these current troubleshooting tools are unable to give precise information in SFC environments because these environments employ ad-hoc forwarding mechanisms that partially breaks the tools mechanism. Related works consider just SDN

networks [11] which are not suitable enough for an SFC environment or present a solution for trace regarding only a specific SFC mechanism [45].

Therefore, the main contribution of this work is the design of SFC Path Tracer, a troubleshooting tool for SFC environments that enables the visualization and characterization of the network paths in the SFC domain. SFC Path Tracer is able to generate packet traces given a proof of transit of network packets in the SFC domain, using probe packets. Moreover, SFC Path Tracer can also work with the real traffic being able to monitor a traffic flow and compute latency measurements of each chain hop. SFC Path Tracer is agnostic regarding the SFC mechanism employed. The second contribution of this work is an SFC configuration framework that emulates SFC topologies and can be used to evaluate other SFC aspects.

The remainder of this work is organized as follows. Chapter 2 describes related concepts to SFC, presents the SFC architecture and common problems debugging an SFC environment; Chapter 3 shows related work on tracing network packets and detecting middleboxes interferences that might be leveraged on SFC environments. Chapter 4 describes the architecture and implementation of the SFC Path Tracer; Chapter 5 evaluates the tool in different scenarios using the SFC configuration framework. Finally, Chapter 6 concludes this work and proposes possible extensions for future work.

2. BACKGROUND

The necessity of a flexible architecture to control and manage the functions of network services emerges with the proliferation of configurable data-planes and network function deployed in virtual machines. This chapter describes proposed standards to enhance network infrastructures. They consist in centralizing network forwarding control (SDN) and bringing virtualization to the network infrastructure (NFV) in order to facilitate the operation and management of network services. These new standards leverage the dynamic chaining of network function (SFC) in the deployment of specific services.

2.1 Software-Defined Networking

Software-Defined Networking (SDN) [22] is a network architecture which aims at decoupling data and control planes. Network control is performed by a central SDN controller, which is responsible for computing and configuring network switches. Therefore, switches are responsible just to forward data packets, following the configurations sent by the controller, which has a global view of network topology. In a pure SDN environment, switches have no understanding regarding network topology. In other words, switches have no default protocol and the SDN controller must install forwarding policies in the switches' flow tables. Network configuration can be performed automatically and remotely, simplifying the installation and modification of network topologies.

2.1.1 Motivation

The SDN motivation is to provide open user-controlled management of the forwarding hardware of a network element. SDN operates on the idea of centralizing control-plane intelligence while keeping the data plane separate. Therefore, the network hardware devices keep their switching fabric, but hand over their intelligence (switching and routing functionalities) to the controller. SDN enables a series of benefits such as: programmability of the network, the rise of network functions virtualization, devices configuration and troubleshooting [31].

2.1.2 Architecture

Figure 2.1 shows the SDN architecture. The control layer is located in the center, where the SDN controller is located. The control plane interfaces with the infrastructure layer through standard protocols, known as Southbound (SB) API. Network switches are placed in the infrastructure layer. The most common Southbound protocol is OpenFlow [39], which allows the dynamic configuration of policies in switches flow tables. Policies are composed by a set of fields (MAC, IP, port, etc) that

are matched against network packets to execute defined actions. For example, the forwarding of a network packet with a specific source MAC address to a determined switch port.

The application layer is where business applications operate. This layer interfaces with the control layer through protocols known as Northbound (NB) API. SDN applications may implement a variety of network functions such as: security systems (IDS, IPS), traffic shaping, traffic steering, load balancing, etc. Currently, there is no standard Northbound API. However, many SDN controllers provide a REST-based interface for applications, such as OpenDayLight controller [42].

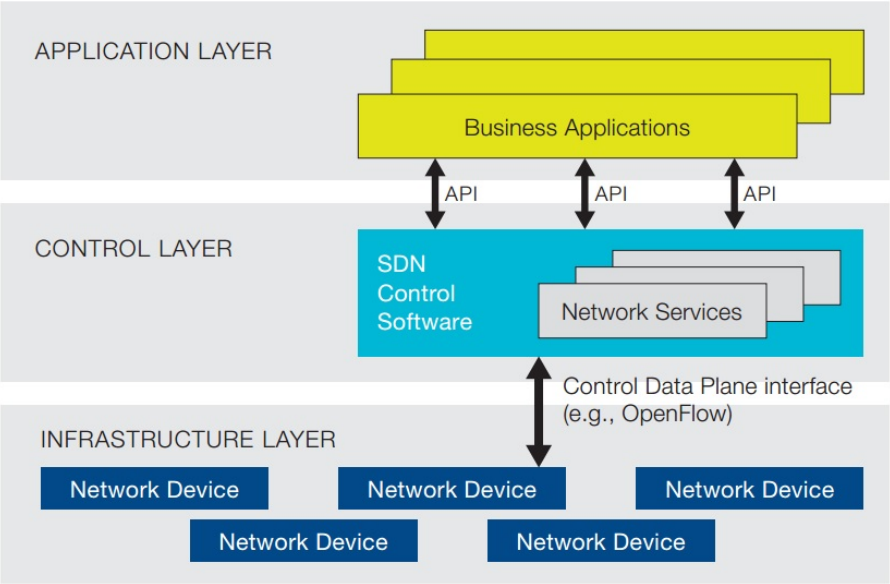


Figure 2.1: Software-Defined Networking Architecture [22].

2.2 Network Functions Virtualization

Network Functions Virtualization (NFV) is a standard specified by ETSI (European Telecommunication Standards Institute) [18]. It intends to transform the way that network operators build networks, by leveraging standard IT (Information Technology) virtualization technology. The goal is to use virtualization techniques to consolidate diverse network equipment types onto industry general purpose platforms, which could be located in Datacenters, network nodes, or on the end user premises [18]. Hence, network functions are implemented in software on top of a virtualized infrastructure, instead of being deployed in a complex appliance with dedicated hardware. Virtualized Network Functions (VNF) run on a range of industry standard servers, known as COTS (Commercial Off-The-Shelf). It provides a flexible network environment, where network functions can be instantiated and replicated when required, to reach service level agreements (SLA).

2.2.1 Motivation

NFV emerges as a new standard to migrate traditional middlebox devices (IPS, DPI, NAT, etc), from specific vendor platforms, to generic virtualized environments, with defined layers and communication standards. Some specific device vendors may require another device from the same vendor to properly work or to reach the expected performance. This leads to complex and static network topologies, with high costs and proprietary technical support. The lack of middlebox mobility may conduct to significant problems in common network deployments. The main motivations for NFV [52]:

- Decrease the expenditure in specific network infrastructure equipments (CAPEX - Capital Expenditures). This is reached by deploying network functions in general-purpose platforms using virtualization techniques. Virtualized network functions (VNF) might also share resources with other VNFs.
- Increase the flexibility and availability of network functions at different infrastructure regions. This can be achieved with standardization of a generic infrastructure to run any network function, called NFVI (Network Function Virtualization Infrastructure). The VNFs can be instantiated in the NFVI dynamically when required.
- Decrease time to implement and provide new features in the network services. Unlike hardware appliances, a software implementation of network functions brings the possibility to rapidly update or change network functionalities, as well as to add new network functions to a service.
- Decrease the expenditure in network operational and technical support (OPEX - Operational Expenditure) since the generic hardware does not require specialized operators.

NFV aims to standardize management and orchestration network layers in order to allow transparent communication between network devices from different vendors, unifying network management.

2.2.2 Architecture

The goal of NFV is to virtualize network functions. Virtualized Network Functions, named as VNF, run on top of NFVI infrastructure. NFV management and orchestration module is responsible for managing the hardware and software resources, as well as managing VNFs [52]. NFV reference architecture, defined by ETSI, is shown in Figure 2.2, with NFV functional modules and Reference Points. Reference points are interface points between components or architectural layers. NFV architecture focuses on functionalities that are necessary for virtualization. It does not specify which network functions should be virtualized. In other words, NFV architecture does not restrict

the use of a specific virtualization solution, rather it expects to use virtualization layers with standard features and open execution reference points towards VNFs and hardwares [19].

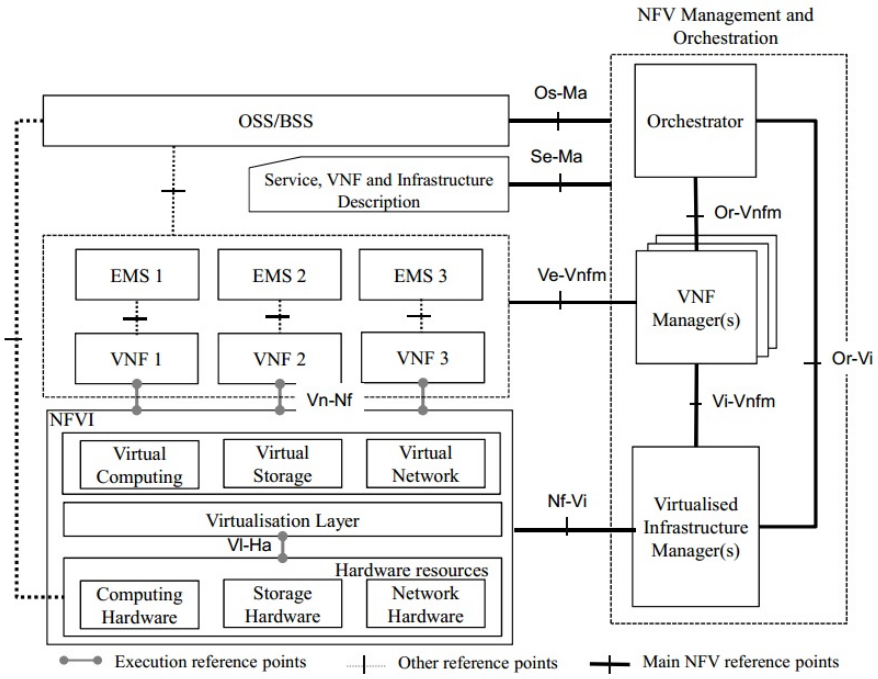


Figure 2.2: NFV reference architectural framework [19].

NFV must also support Physical Network Functions (PNF), which are the traditional middlebox appliances. This is essential to the NFV standard in order to maintain compatibility with current network environments. Therefore, NFV can be adopted gradually by companies, requiring small changes in the infrastructure.

2.3 SDN and NFV

NFV and SDN do not depend on each other to be adopted. However, they are complementary, enhancing their benefits together on a system. An OpenFlow-based SDN controller provides a flexible framework that can be used by the NFV orchestrator layer to manage network functions [41]. Figure 2.3 shows an example of SDN and NFV concepts working together on the same network topology. NFV orchestration communicates with an SDN controller through the Northbound API. NFV orchestration is able to manage the network topology while the SDN controller interacts with switches through OpenFlow. Switches can be virtual or physical. Figure 2.3 also shows virtualized network functions (VNF) running in virtual machines on top of Hypervisors hosted by general-purpose servers. Theses VNFs are managed by the NFV orchestration and management layer.

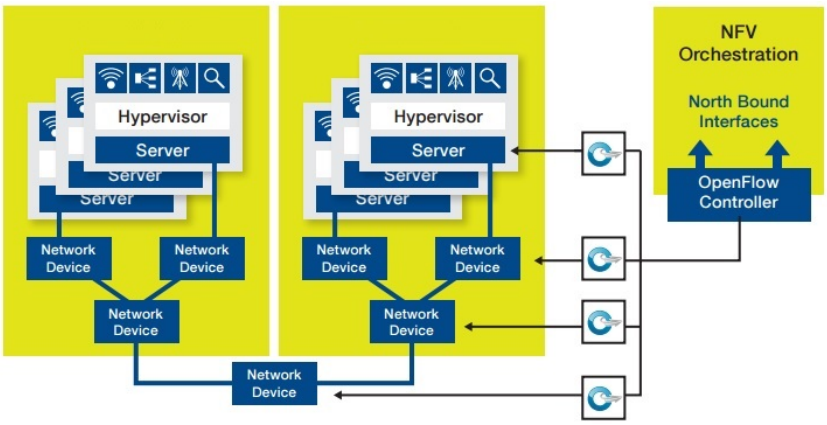


Figure 2.3: SDN and NFV concepts on the same deployment [41].

2.4 Service Function Chaining

Network services commonly require different kinds of processing when its packets traverse the network infrastructure. This processing is performed by middleboxes, also called network functions. Middleboxes manipulate traffic for different purposes, and it is defined as any intermediary device performing functions other than forwarding packets on the datagram path between a source and destination hosts [15]. The datagram is a self-contained and independent entity of data carrying sufficient information to be routed, defined in RFC 1594 [57], and the datagram path is the pathway formed by network devices for datagram traffic. Middleboxes play a critical role on network infrastructures, with functions such as: firewalls, traffic shapers, load balancers, Intrusion Detection Systems (IDS), Intrusion Prevention Systems (IPS), and application enhancement boxes. These functions, usually performed by specific hardware appliances, are characterized by the lack of mobility and its high cost. Some initiatives have been proposed to virtualize such functions to operate in commodity platforms [25, 53]. These functions virtualization represent the VNFs, in the NFV architecture.

Usually, network services require a set of network functions, which are mostly topological dependent due to its hardware dependencies. Some common deployments insert network functions on the data-forwarding path between communicating peers [20], which restrict infrastructure changes and adoption of new network functionalities. The static nature of such deployments limits the ability of an operator to modify existing services or introduce new network functions. The introduction of a new network function can be difficult, in both technical and organizational spaces, requiring topology changes and even manual configurations [51].

Deployment of network services is changing with SDN and NFV. Instead of requiring the placement of service functions along the direct data path, the traffic is steered through network functions, wherever they are deployed. Therefore, emerges the possibility of network services being deployed and changed dynamically.

Figure 2.4 shows an example of a network service deployed with network function chaining in the NFV architecture. The ETSI working group defines the VNF-forwarding graph, which is responsible for configuring a forward path for network packets. NFVI provides a virtualization layer through the N-Pop (Network Point Of Presence), where VNFs are instantiated. VNFs are the network functions that the service requires (firewall, IDS, NAT, etc). The dotted line shows a logical path where packets will be forwarded through VNFs. The solid line shows a physical link, which represents appliances where VNFs are instantiated or the PNFs are located.

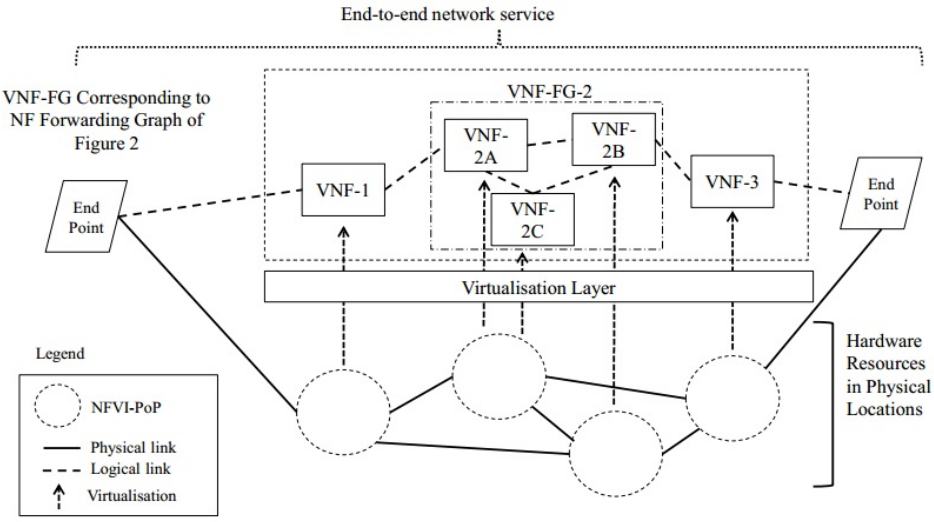


Figure 2.4: A network service example with VNFs forming a forwarding graph [19].

Similarly to ETSI VNF-Forwarding Graphs, IETF (Internet Engineering Task Force) defines the Service Function Chaining (SFC) [20], as an abstract view of network functions and the order in which they need to be applied. SFC forms traffic chains among service functions regarding a specific network service. SFC is instantiated from a set of network functions, which are placed in specific locations forming a forwarding graph. SDN and NFV play an essential role in the SFC adoption, so the network topologies could become more flexible and malleable lowering the time to market for new features.

2.4.1 Motivation

The following points describe aspects of existing service deployments that are problematic and that the SFC proposals aims to address [51]:

- *Topological dependency*, deployment of network services are often coupled with network topology. Such dependency can restrict network operators from optimally utilizing service resources, reduces flexibility, as well as the deployment of new services. This dependency also limits the placement and selection of service functions.

- *Configuration complexity* is a direct consequence of service functions being topological dependent. Once an SF is configured, operators may hesitate to change configurations, which leads to static network service deployments.
- *High availability*, since traffic reaches many SFs based on network topology, redundant service functions must be placed in the same topology as the primary service. Therefore, topological dependency affects the complexity of SFs availability.
- *Consistent ordering of Service Functions*, SFs are independent regarding ordering. However, many SFs must keep a restricted ordering in the service deployment perspective.
- *Application of Service Policy*, SFs depend on topology information to determine service policy selection, i.e., the SF action taken. Topology information often does not give adequate information to SFs, then SFs are forced to individually perform a more granular classification and its semantics may be overloaded.
- *Transport dependency*, SFs are deployed in a network with a variety of transport protocols and tunneling. SFs topological dependencies may require the service functions to support transport protocols.
- *Elastic service delivery*, network services changes, such as adding and removing SFs, commonly lead to VLAN or routing changes. Rapid changes to the deployed service capacity can be hard to perform, due to the risk and complexity of VLANs or routing modifications.
- *Traffic Selection Criteria*, network traffic on a particular segment traverses all service functions, whether the traffic requires service enforcement or not. In some deployments, more granular traffic selection is achieved using policy routing or access control filtering, which is operationally complex.
- *Classification/reclassification per Service Function*, classification functionality often differs between service functions, and SFs may not leverage the classification results from other service functions.
- *Symmetric traffic flows*, SFC may be unidirectional or bidirectional depending on the service deployment requirements. Existing service deployment models provide a static approach to create chains, with complex topological configurations. On a bidirectional chain, it would be necessary the same complex configuration in both directions.
- *Multi-vendor Service Functions*, deploying service functions from multiple vendors often requires a vendor specialized support, hence standards are needed to ensure interoperability.

In modern network deployments, the number of middleboxes almost follows the large number of forwarding elements such as routers and switches. For instance, on the enterprise middlebox survey [54], a deployment is mentioned with 2850 L3 routers have a total of 1946 middleboxes.

The management of this variety of Middleboxes is complicated and typically they are physically inserted on the data-forwarding path between communicating peers. The network traffic is directed via virtual LANs (VLANs) and policy-based routing techniques. Consequently, services are tightly coupled to the physical network topology [49]. Virtual LANs techniques might be considered as an early SFC technique.

2.4.2 Architecture

The basic concepts of SFC are defined by IETF (Internet Engineering Task Force) [26]. It defines all SFC elements and its functions. In this work the taxonomy used is based on this IETF definition, where network functions are named as Service Functions (SF). Other terms used in SFC architecture are defined in Table 2.1.

SFC enables the creation of composite network services that consist of an ordered set of SFs (SF1 $\Rightarrow$ SF2 $\Rightarrow$ SF3) that must be applied to packets/frames/flows selected as a result of classification. Each SF is referenced using an identifier that is unique within an SFC-enabled domain. SFC describes a method for deploying SFs in a way that enables dynamic ordering and topological independence of SFs, as well as the exchange of metadata between its components [26]. Figure 2.5 shows the SFC architecture, where the SCF, Service Classification Function, performs packet classifications and encapsulation for an SFC-enable domain. The encapsulation contains SFP for service functions of a particular chain.

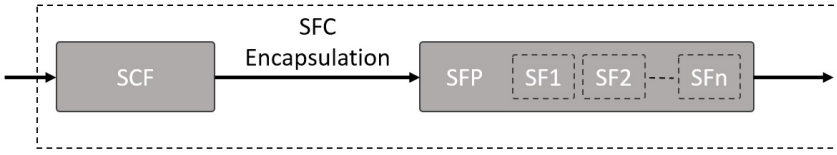


Figure 2.5: Service Function Chain Architecture [26].

The SFC elements shown on Table 2.1 can be seen in Figure 2.6. The figure shows a generic example of an SFC deployment, and where the SFC elements are placed in a network topology. In this example, a client sends a request to remote servers with standard protocols such as TCP/HTTP. SCF is responsible for classifying network traffic and adding the SFC-Encapsulation to the chain traffic then forwarding to the SFC-Enabled domain. In the SFC-Enabled domain, SFC-Encapsulation packets are forwarded to SFs (SF1, SF2 and SF3) by SFFs (SFF1 and SFF2) following the SFP. SFC defines the ordered set o SFs of a chain, which afterward is translated to an SFP, the actual path of the chain, that is used to configure and install rules in SFFs.

In order to support legacy SFs, witch are not aware of SFC-Encapsulation, an SFC-proxy element is defined. SFC-proxy is responsible for removing the SFC-Encapsulation for unaware SFs and add it back when forwarding back to SFFs. Important to notice that this proxy function is placed between SFF and SF elements and it can be achieved by multiple approaches. SFC-proxy can be performed by an external box or inside the SFF or SF spaces. Once SFP is finished, the CTF

Table 2.1: SFC taxonomy used [26].

Taxonomy	Name	Description
SF	Service Function	A function that is responsible for the specific processing of received packets. SF can be aware or unaware. An aware SF knows it belongs to a chain and is able to treat SFC metadata. When the SF has no knowledge of the SFC layer, it is called an unaware SF. Traditional middleboxes might be an example of unaware SF, requiring an SFC-proxy to be part of a chain.
SFC	Service Function Chain	Defines an ordered set of abstract Service Functions (SFs).
SCF	Service Classifier Function	Defines the packet classification that will be forwarded to a specific chain. The classifier locally matches network traffic flows against policies for subsequent application and then forwards to the required set of network service functions.
SFF	Service Function Forwarder	It is responsible for forwarding traffic to one or more connected SFs according to information carried in the SFC encapsulation. SFF is usually composed of one or more network switches.
SFC-proxy	Service Function Chaining Proxy	Removes and inserts SFC encapsulation on behalf of an unaware SF.
SFP	Service Function Path	It is a constrained specification of the specific service function instances and network nodes chosen in which the packets will pass through.
SFC-Encapsulation	Service Function Chaining Encapsulation	It provides a minimum SFP identification, that will be used by SFFs and SFs.
SFC-Enabled Domain	Service Function Chaining Enable Domain	A network or region of a network that implements SFC.
CTF	Chain Termination Function	The termination point of the chain, where the SFC-Enabled Domain ends. This element should remove any SFC-Encapsulation.

removes the SFC-Encapsulation from packets and forwards the traffic to its original destination. CTF can be SCF switches or the last SFF of the chain.

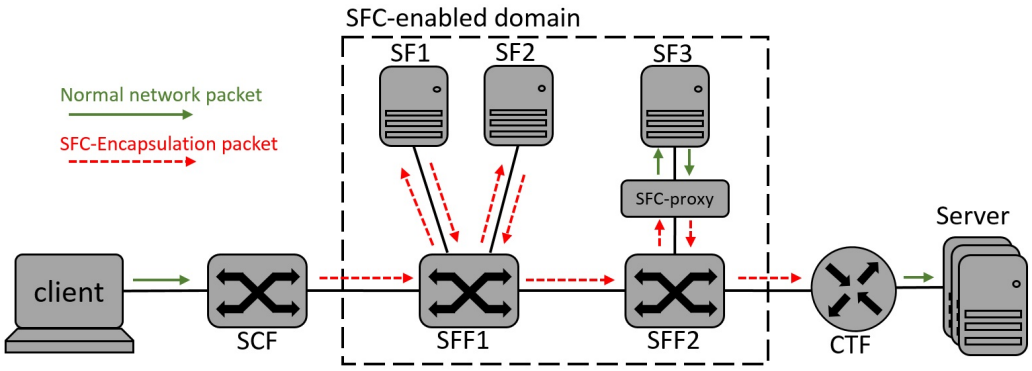


Figure 2.6: Generic SFC deployment example and its elements.

2.5 SFC encapsulation proposals

SFC is a prominent research field in network area [1, 5, 8]. There are several SFC proposals [14, 24, 30, 58] from industry and academia. The following sections present SFC proposals based on IETF drafts. SFC techniques may use current network protocols, leverage encapsulation protocols or even add new layers in the protocol stack. For instance, most of these techniques can be implemented using OpenFlow rules to configure the data plane to forward packets through service chains.

2.5.1 Network Service Header

Network Service Header (NSH) [24] is one of the most complete solutions of SFC. It aims to implement a service function chaining as it is defined on the SFC architecture [26]. NSH contains service path information and optional metadata that are added to a packet and used to create a service plane. The original packets preceded by NSH are encapsulated in an outer header for transport.

NSH header is added by a network classifier and removed by the last SFF or SF in the chain. NSH is composed of a *Base Header*, a *Service Path Header*, and *Context Header*, as shown in Figure 2.7. The base header provides configuration fields, such as metadata type (*MD Type*). The service path header provides the chain and its service functions identifiers. Context Header provides a space for a context metadata specified by MD Type field. Context header can be mandatory, which means that metadata bytes, with a fixed length, must be added immediately following the Service Path Header. Context header can also contain variable metadata length, following MD Type definition.

Service Path ID indicates a path that is formed by a set of SFs identified by a service index field. The first classifier in the NSH function path should set the *Service Index* (SI) to 255. SI is

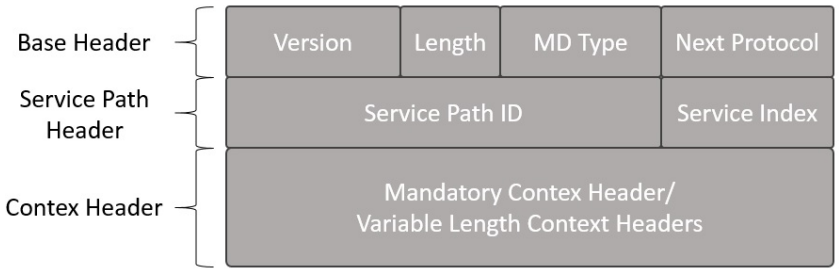


Figure 2.7: Network Service Header [24].

decremented along the chain by SFs or proxy nodes to reach the next SF on the chain. It is possible to reclassify traffic in the middle of the path chain, which will change the NSH header and set SI to 255 again.

NSH can be formed by a NSH-aware service function, that may alter the content of the NSH headers, and a NSH-proxy, which is used to remove the NSH header for an unaware SF. NSH-proxy and classifiers can add and remove NSH headers, as well as the SFF, which is also responsible for selecting the service function path.

NSH emerges as the leading SFC approach even with the requirement of an NSH-enabled switch. Open vSwitch [43] supports NSH, and can be used to perform tests and evaluations. OpenDayLight [42] controller has an SFC module that also supports the NSH protocol. These softwares can be used in a virtualized environment to build a testbed for NSH evaluation. However, there is no easy-to-use environment to exercise NSH.

2.5.2 Ethernet MAC Chaining

MAC Chaining is based on the current IEEE 802 Ethernet header for physical and virtualized environments. The basic mechanism is to use MAC addresses in the Ethernet header for both identifying chains and forwarding packets along the MAC chain [14]. These assigned Ethernet addresses are called Chain Segment MAC (*CS-MAC*). The CS-MACs allows MAC chaining to be implemented on existing Ethernet infrastructure. Therefore, no extra network header is necessary to perform the service function chaining.

Figure 2.8 illustrates a chain example, where a classification (SFC) is performed and the MAC destination field of the packet is changed to the chain enter point identifier, *CS-MAC A*. In the SFF, a flow rule will forward the packet from *A* to the first hop of the chain, and increment the chain segment. When back from *SF1*, the packets will be matched by another SFF flow rule, *CS2* and then forward to *SF2*. A final chain rule will forward packets to the Chain Termination Function (CTF) that will perform any de-encapsulation and operations required to continue forwarding to the final destination.

MAC Chaining uses Ethernet MAC address bits to encapsulate CS-ID information, as well as a branch decision for aware SFs. SFF matches the source MAC fields, increments the Chain ID

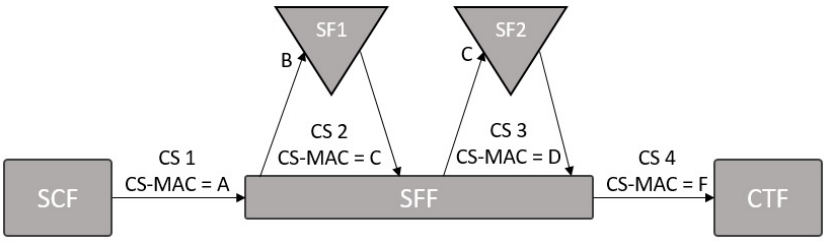


Figure 2.8: MAC Chain Segments Addressing [14].

and forwards the packets to the next hop of the chain. Therefore, another match will be performed against the incremented Chain ID for subsequent chain hops. MAC Chaining supports unaware SFs with no SFC-proxy, as it does not add an extra network header.

Currently, Ethernet MAC Chaining has no public implementation nor proof of concept. However, it is being developed a first proof of concept under OpenDayLight controller. This SFC approach does not require specific switch modification and works with any OpenFlow-based switch.

2.5.3 An IPv6 Extension Header for Service Function Chaining

SFC-IPv6 leverages the IPv6 extension header to perform Service Function Chaining [30]. The IPv6 extension header is used by SFC data plane elements to make forwarding decisions in an IPv6-enabled SFC domain and it conveys metadata that are processed by SFC-aware nodes.

Based on the selection of Next Header from IPv6 header and following the uniform format of IPv6 extension headers (RFC 6564), SFC extension header is used as shown in Figure 2.9. The fields *Next Header* and *Hdr Ext Len* follows the RFC 6564 and identifies the type of header immediately following the extension header and the length of SFC extension header. The *Flags* field comprises a set of 8 flags where it is informed if optional field are present and other bits reserved for future implementations. *SF Index* field is decremented by 1 and used to detect SFC loops and *SFC ID* identifies the SFC associated to the IPv6 packet. The optional *SFP ID* is used to convey an identifier of a path that is bound to a given SFC. *Information Elements* field may be used to conveys one or multiple optional metadata that may be supplied within an SFC-Enabled Domain.

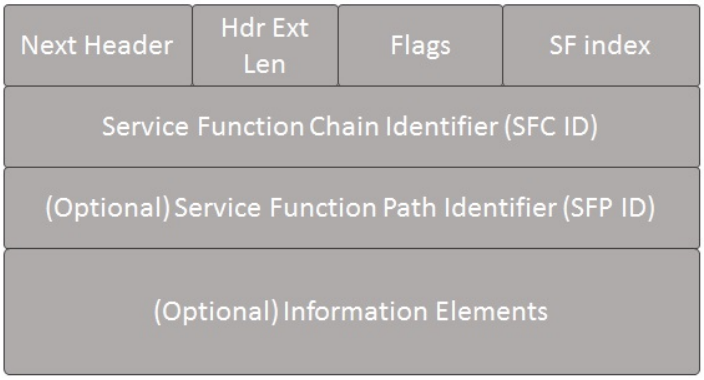


Figure 2.9: IPv6 Extension Header for SFC [30].

An IPv6 network packet, in IPv6-enabled SFC domain, has the advantage to using any specific transport encapsulation scheme when forwarding packets between nodes that are connected to the same subnet. A classifier typically inserts the SFC Extension Header to incoming packets that matches SFC classification policies. SFFs are responsible for decapsulating the packet, and process the SFC information carried in the SFC Extension Header. The SFF will use this information to position itself in the forwarding path, determine which SF instances need to be invoked next and make its forwarding decision according to the SFC instructions carried in the SFC Extension Header.

2.6 OpenDayLight

The OpenDaylight (ODL) project is a commercial, collaborative, open-source and java based platform to accelerate the adoption and innovation of SDN and NFV. ODL architecture is developed based on the Open Services Gateway Initiative (OSGi) which is a modular development framework where loosely coupled modules construct the entire platform. The modules can be built independently with the ability to import and export data from one another [29]. The innovation of Model-Driven Service Abstraction Layer (MD-SAL) in the architecture leads to developing models for automatic management and configuration of the network. MD-SAL provides ODL with the ability to support any protocol talking to the network elements as well as any network application. The flexibility is a strong characteristic of ODL, where multiple plugins might be connected performing several functionalities. Figure 2.10 shows the ODL architecture from 5th release named Boron.

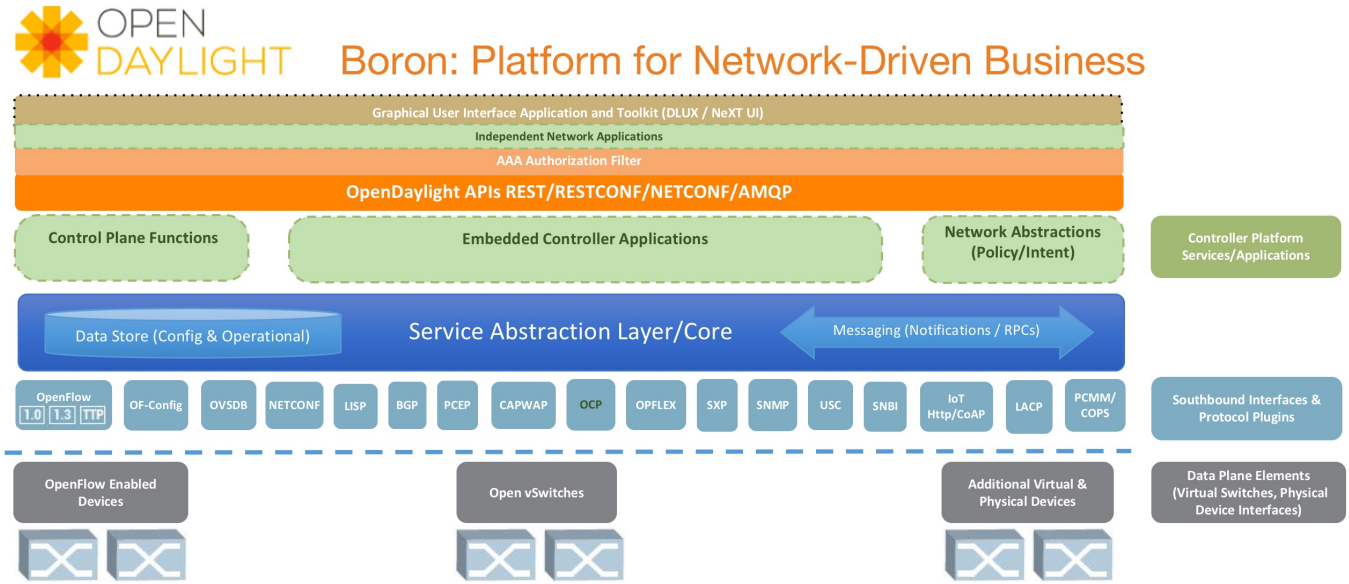


Figure 2.10: OpenDayLight modular architecture [2].

ODL provides a rest API (NorthBound API) for external applications be able to configure internal plugins which might control and manage network elements through protocols (SouthBound API) such as, OpenFlow, BGP, Netconf, etc. OpenDayLight features are composed by Base Network

Service Functions, Platform Network Service Functions, and the Service Abstraction Layer [29], highlighted in Figure 2.11 .

- *Base Network Service Functions* are responsible for collecting and providing information about network topology, statistics, switches and hosts information, etc. They expose NorthBound (NB) APIs for applications as well as to any other internal plugin.
- *Platform Network Service Functions* are pluggable oriented services performing specific network tasks. Two examples of platform functions are L2 Switch and SFC. L2 switch provides L2 switch functionality and creates reusable services such as address tracking, basic spanning tree protocol, event-driven packet handling and basic path computation. On the other hand, SFC provides an SFC modeling and the ability to define a chain of network functions. This SFC function is used in this work to deploy chains that will be evaluated with SFC Path Tracer.
- *Service Abstraction Layer (SAL)* is the central element of ODL and is responsible for providing connectivity among ODL plugins. Most SAL services are built based on SouthBound plugins features. ODL implements Model-Driven SAL, detailed in Section 2.6.1.

The southbound (SB) interface and protocols plugins enable communication between the controller and the network devices. ODL support multiples SB protocols. These SB protocols enable ODL to support heterogeneous networks and ensure interoperability with other technologies and between other vendors [29]. ODL provides SB plugins that implement protocols such as, OpenFlow, OVSDB, NETCONF, BGP, etc. Figure 2.11 shows ODL architecture with network basic function, and the service abstraction layer providing communication between functions either NB or SB plugins.

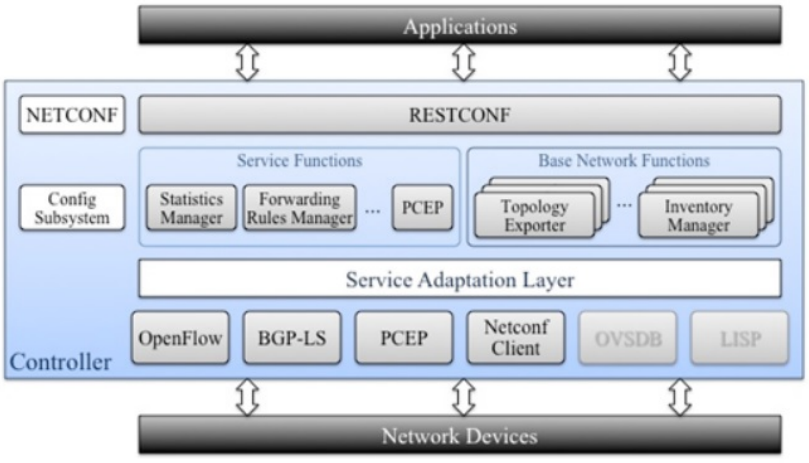


Figure 2.11: OpenDayLight plugins and service abstraction layer [23].

Generally, ODL plugins are built based on YANG models, detailed in Section 2.6.2. An ODL plugin should implement listeners for data modification on defined YANG model. This implementation should take actions based on data added, removed or changed. A plugin can also

listen to other plugins data models. User applications are able to interact with plugins through auto generated RESTCONF APIs. Plugins can interact with each other by adding or modifying data of other plugins model. The communication among plugins or provided RESTCONF APIs are routed by the service abstraction layer.

The controller plugins can be either data/service provider or data/service consumer. A provider provides data/services to the data stores through its APIs. A consumer consumes services/data located in the data stores, provided by one or more providers.

2.6.1 MD-SAL

OpenDaylight leverages the Model Driven Software Engineering (MDSE), which is defined by the Object Management Group (OMG). MDSE describes a framework based on consistent relationships between models, standardized mappings and patterns that enable model generation and, by extension, code/API generation from models [23].

Model-driven Service Abstraction Layer (MD-SAL), the Model-driven approach to service abstraction, presents an opportunity to unify both northbound and southbound APIs and the data structures used in various services and components of an SDN Controller [4]. Basically, MD-SAL defines a common layer, concepts, data model and messaging patterns, and also provides infrastructure/framework for application and inter-application communication. MD-SAL also provides common support for user-defined transport and payload formats (JSON) for NB communication [23]. MD-SAL can store data models defined by plugins, allowing provider and consumer plugins to exchange its data through the data stores.

MD-SAL exposes two different APIs, Binding APIs and DOM (Document Object Model) APIs [23]:

Binding APIs are Java-based APIs which uses interfaces and classes generated from YANG models. They provide a compile-time safety, as they use Java language to implement its content.

DOM APIs are payload format APIs that interpret YANG models and provide functionality from any valid model. DOM format makes the model more powerful and efficient but provides less compile-time safety.

MD-SAL data handling is separated into two brokers: a binding-independent and binding-aware brokers. Binding-independent brokers (DOM) is the core component of MD-SAL and intercepts YANG models at run-time. DOM broker uses YANG to describe data and instance identifiers to describe paths to specific elements in the data stores. DOM broker is responsible for manipulating requests in the data store, and it relies on the presence of YANG schemas, which are interpreted at run-time for functionality-specific purposes, such as RPC routing, data store organization, and validation of paths.

Binding-aware brokers rely on Java-based APIs, generated from YANG models and Java DTOs (Data Transfer Object), which are enforced by code generation. Binding-aware brokers are connected to DOM-brokers so that application or plugins can communicate with their respective binding-independent equivalents. Figure 2.12 shows brokers connection and the relationship with consumers and providers.

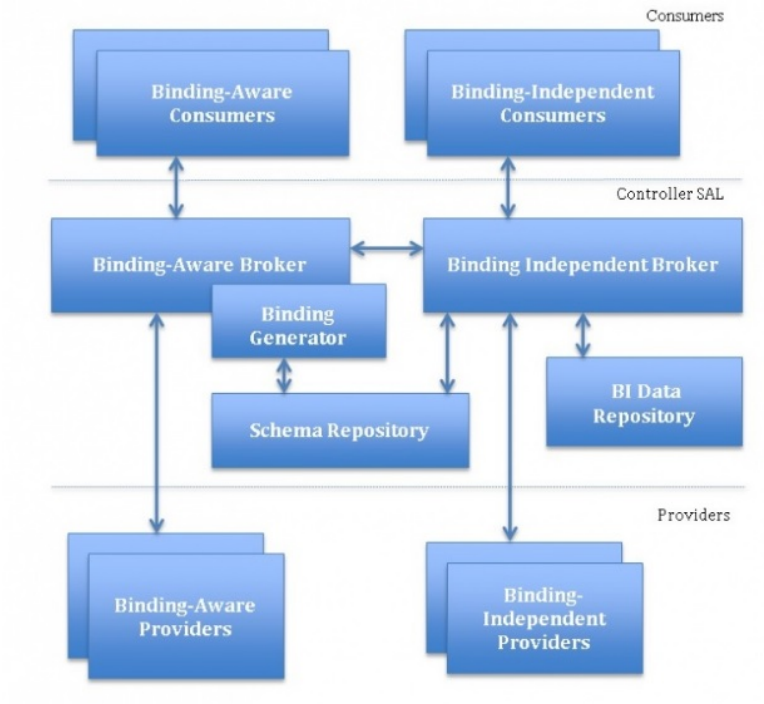


Figure 2.12: Architecture of MD-SAL [4].

OpenDaylight data storage is composed of data trees, with two divided logical data stores views: operational and configuration. However, the data view of both is independent and can be addressed using instance identifiers. For instance, operational data can represent the current state of a specific device while configuration represents actual device configuration.

2.6.2 YANG

ODL uses Yet Another Next Generation (YANG) as data modeling language which was basically developed to model RPCs, notifications, configuration, state data of network elements, as well as constraints to be enforced on the data [13]. It was created to be a data modeling language for the NETCONF network configuration protocol, and used as the modeling language for ODL plugins. YANG models the hierarchical organization of data as a tree in which each node has a name, and either a value or a set of child nodes. YANG provides the description of nodes, as well as the interaction between those [23]. Figure 2.13 illustrates the data tree organization of YANG.

YANG presents the advantages of a human readable representation, hierarchical data model being possible to reuse types and groups. It is extensible through augmentation mechanisms and

supports operation definitions, such as RPC and notifications. In ODL, YANG is used to model almost everything, from applications and plugins to the internal MD-SAL behavior.

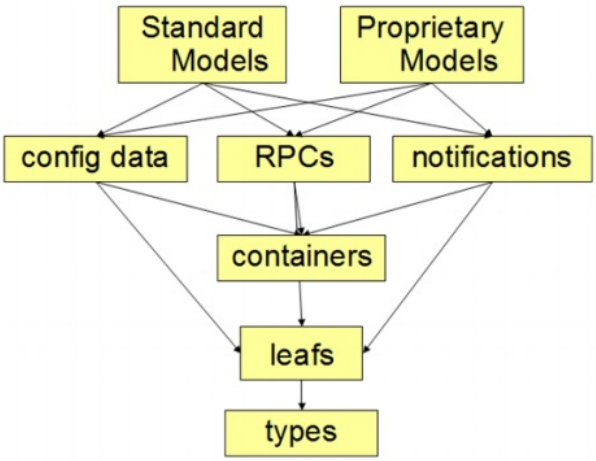


Figure 2.13: YANG data tree modeling [23].

YANG is managed by the YANGTools project, which contains modules that describe the code generation from YANG models, the mapping between YANG and DOM/Java formats, the modeling of data stores and its operations (transactions, RPCs, and notifications).

Figure 2.14 shows the ODL plugin structure built from a YANG model. Based on the YANG data model definition, YANGtools will generate Java-based classes for the defined model, as well as REST APIs to give data access to external applications. Therefore, the plugin implementation should handle data state changes and take action on the appropriate devices.

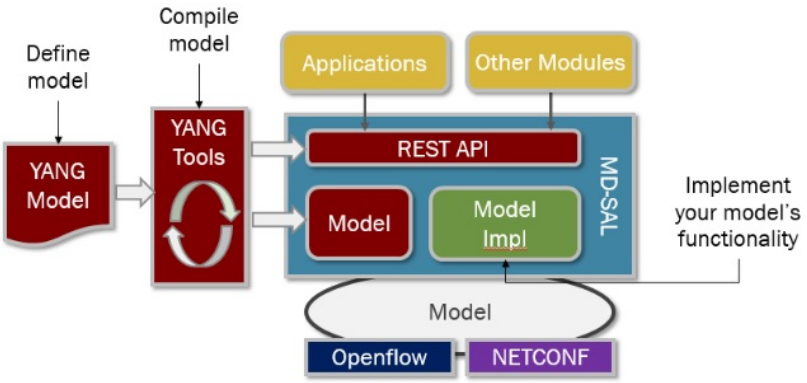


Figure 2.14: ODL plugin structure with YANG model definition in the MD-SAL environment [3].

2.6.3 SFC implementation

The OpenDayLight SFC project [8] implements the SFC architecture [26] as defined by IETF. This project provides the infrastructure such as the chaining logic and SFC APIs, to provision a service chain in the network. SFC project, as any other ODL feature, is modeled in YANG, which

mainly comprises of SFs, SFFs, classifiers (SCF), service chains (SFC), chain paths (SFP), etc. OpenDayLight's modular architecture enables the possibility of many SFC methods implementations. ODL SFC project implements NSH technique over VxLAN encapsulation. SFC also implements a technique based on L2 OpenFlow rules using VLANs or MPLS to encapsulate the packets. SFC YANG models can be easily augmented to include extra data related to other SFC implementations as well. SFC implementation uses SB plugins to install configuration rules on switches. It can install rules though OpenFlow or OVSDB (Open vSwitch Database) plugins, both open-source ODL projects.

The SFC model defines its components (SFFs, SFs, SFPs, etc), as well as RPC to actually deploy a service function chaining (Rendered Service Path - RSP). For instance, the Service Function component, modeled in YANG, presents the following parameters:

- *Name* is the unique name to identify a specific SF.
- *Type* refers to VNF type function, such as firewall, DPI, IPS, etc. Usually, an environment can have multiple SFs of a specific type. Therefore, providing the ability to perform load balance of SFs.
- *Data Plane Locator* specifies how service functions can be reached by an SFF. In this element, it is defined the SFF connected to it, the transport type (L2, VxLAN, etc) and its network addresses (MAC or IP).

To configure a Service Function Forwarder the following parameters are modeled:

- *Name* is the unique name to identify a specific SFF.
- *Node* refers to the ODL node that stores this switch information, in other words, the switch identification.
- *Data Plane Locator* specifies the SFF interfaces. Each switch port must be defined here with transport type, address and port number.
- *Service Function Dictionary* informs all SFs connected to this SFF. Each item on SF dictionary must inform the plane locator name from the SF and the SFF.

Figure 2.15 shows a RESTCONF example of two elements from the SFC configuration model. On the left side, a chain named *c1* is configured to reach a firewall, a DPI and an IDS. These elements represent the ordered list of service function types that form a service chain. On the right side, it is also shown an SF instance configuration of type IDS, called *sf1*. Other parameters to reach this SF are also configured such as MAC address and VLAN.

SFC project also models other SFC elements, such as Service Function Path, where it is specified path symmetric information, the classifier used, and SFC identification. Classifier model is built based on an SFF without SFs dictionary. Rendered Service Path (RSP) models the RPC to deploy a service chain. Considering multiples SFC implementations, a specific SFC technique is triggered depending on data configuration, such as chain transport and encapsulation types.

```
"service-function-chain": [
  {
    "name": "c1",
    "sfc-service-function": [
      {
        "name": "fw",
        "order": 0,
        "type": "service-function-type:fw"
      },
      {
        "name": "dpi",
        "order": 1,
        "type": "service-function-type:dpi"
      },
      {
        "name": "ids",
        "order": 2,
        "type": "service-function-type:ids"
      }
    ]
  }
]
```

```
"service-function": [
    {
      "ip-mgmt-address": "10.0.0.11",
      "name": "sf1",
      "sf-data-plane-locator": [
        {
          "mac": "00:00:00:00:00:11",
          "name": "sf1-plane-0",
          "service-function-forwarder": "SFF2",
          "transport": "service-locator:mac",
          "vlan-id": 304
        }
      ],
      "type": "service-function-type:fw"
    }
  ]
```


Figure 2.15: Example of RESTCONF for an SFC and a single SF in the SFC configuration model.

SFC OpenFlow Renderer

Most of SFC techniques implemented in ODL are achieved by data plane configuration rules using OpenFlow. The ODL SFC project provides an architecture for SFC OpenFlow renderers, as shown in Figure 2.16. NorthBound SFC API exposes SFC model for operational configuration, which is stored in the MD-SAL layer. The SFC OpenFlow renderer is invoked after an RSP call. Different OpenFlow renderers have its own business logic and are responsible for creating OpenFlow rules regarding the SFC technique employed. Finally, the OpenFlow plugin is called to install the rules on switches.

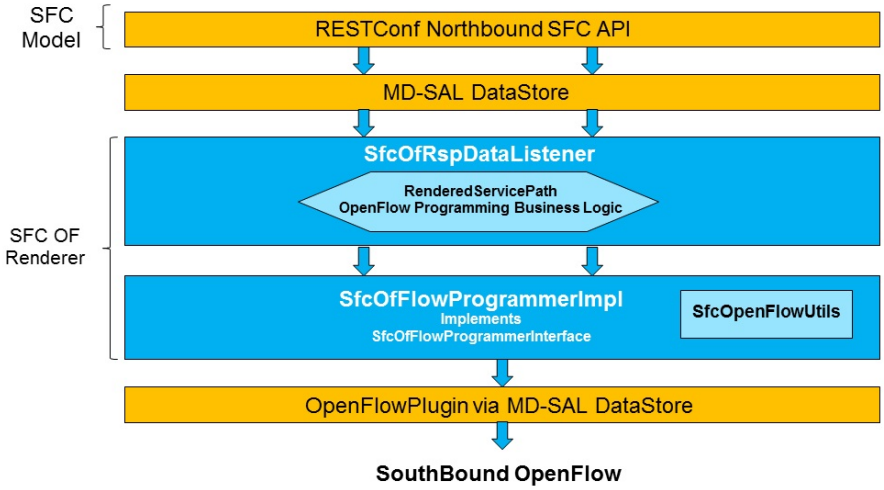


Figure 2.16: SFC OpenFlow Renderer Architecture [6].

Two examples of SFC OpenFlow renderers are NSH and VLAN renderer. NSH renderer creates OpenFlow rules that matches NSH fields and forwards packets to specific SFs at each chain

hop. Currently, NSH is implemented over VxLAN-GPE [50] tunnels. Therefore, the generated OpenFlow rules will set the remote tunnel IP for the next hop destination. The SFC VLAN renderer creates OpenFlow rules to match chain identifiers encoded in the DSCP field of IP header and match the source MAC address to identify the chain hops. Once matched, these OpenFlow rules will set the destination MAC address of the next hop and forward the packet through a specific switch port.

SFC OpenFlow Renderer uses flow pipeline tables to process packets. The described matches and action are installed in OpenFlow tables as shown in Figure 2.17. The tables are divided by functions, such as *Transport Ingress* responsible for matching incoming packet based on supported encapsulations, *Path Mapper* will match based on tunnels ID, such as VLAN ID and remove them, *Next Hop* that identify next hop destination and finally *Transport Egress* that encapsulates and outputs the packets.

Table ID	Function
0	Classifier
1	Transport Ingress
2	Path Mapper
4	Next Hop
10	Transport Egress

Figure 2.17: Pipeline tables for SFC OpenFlow renderer [6].

2.7 SFC troubleshooting

Service Function Chaining is used in scenarios such as network security [36], data center networks [34], and mobile networks [40]. SFC can also be deployed in hierarchical layers [37] allowing an SFC organization into multiple domains with different SFC encapsulations. In complex deployments, service functions can have its instances spread in several servers. These aspects make the debugging process complex, specially in huge network topologies. It might be hard for network operators to find root causes of a problem when the system is not working properly.

SFC configuration involves multiple steps, such as: defining a chain, translating the intent chain into network policies, installing switch flow rules and even configuring service functions. Any misconfiguration can lead the system to erroneous behavior, with packets being forwarded to wrong SFs or even dropped along its path. A malfunction or overloaded SF can degrade the performance of the whole chain. Troubleshooting can be a difficult task in the SFC domain, even when network traffic does reach the final destination. It might be challenging to assure that the traffic correctly passes through all configured service functions or detect overloaded hops.

Different SFC approaches present a variety of techniques to implement the SFC encapsulation which affects the complexity to properly evaluate such implementations. Once SFC is deployed, it is hard to evaluate or troubleshoot possible problems, regardless of the SFC technique used. For instance, if a reachability problem is detected in the chain path, it might be a massive work to find

where the packet is being lost. Network operators may need to dig into each device and inspect packets using tools such as *tcpdump*[1]. Some of the common SFC environment problems that should be requirements for a troubleshooting tool to detect are [12] :

- Wrong switch configuration;
- Conflicted rules installed on switches;
- Switch connectivity problem;
- SF not working or presenting connectivity problems;
- SF introducing high latency in the traffic;
- Assurance that all network functions are being reached;
- Assurance of correct packet forward ordering of network functions;
- Overloaded network link;
- General debugging on a deployed chain.

Network operators spend significant effort in ensuring that network meets their intent behavior [21], with the lack of tools to check network functions in the path. The increasing scale and dynamism that NFV introduces add, even more, challenges to detect possible misconfiguration. OAM for SFC [12] provides a reference framework for Operations, Administration and Maintenance (OAM) for Service Function Chaining (SFC). It indicates the aspects of SFC that should be monitored. The monitoring element should have capabilities to monitor aware and unaware service functions, service function performance, service functions forwarders and the classifier. OAM for SFC highlights monitoring functions such as connectivity, continuity, trace and performance measurements. Packet trace function enables the detection of above common problems. Once a packet trace is generated, a path can be monitored to detect connectivity and performance issues such as packet losses and overloaded SFs. Packet traces give informations such as prof of transit of network packets and then pinpointing the problematic location when traffic is delayed or does not reach its destination.

An SFC trace tool should detect all network elements that interact with network packets in a specific chain. Detecting both SFFs and SFs allow the generation of complete packet traces in order to check SFC correctness.

Moreover, it is important to the SFC trace generation tool to be agnostic of the SFC encapsulation mechanism. Currently, there are several proposals of SFC encapsulation techniques [14, 24, 58]. However, none of them has yet been adopted as a de facto standard for Service Function Chaining. Network Service Header (NSH) [24] is the most accurate proposal regarding the SFC architecture. NSH has optional metadata fields that can be used for trace and performance purposes. However, NSH is far from being considered as an SFC standard as it is a new protocol stack and demands switches and service functions to have explicit NSH support.

[1] http://www.tcpdump.org/

3. RELATED WORK

Some tools and frameworks that generate packet traces may be used by network operators to help identify recurrent problems in the SFC environment. Some of the tools described in the literature include: SFC Traceroute [45, 59], Tracebox [16], SDN Traceroute [11] and NetSight [28]. Related works considering network verification are also presented.

3.1 Services Function Chaining Traceroute

Services Function Chaining Traceroute [45] defines a protocol that allows a user to check liveness and get reports of the service-hops of a service path using NSH. The proposed protocol defines a trace packet that makes use of the metadata space from NSH header. Basically, it defines a trace packet that will traverse the SFs from a specific chain and store information from a defined chain hop. Figure 3.1 shows the trace packet format in the NSH header. Identified by *MD-type* field on NSH header, SFC Traceroute uses the OAM (Operations and Management Requirements) protocol [12] to define a trace header. The SFC Traceroute defines the following fields to store trace information:

- *Trace Msg Type* defines a trace type: 1 for Trace Request and 2 for Trace Report.
- *SIL* (Service Index Limit): The index of the service function hop which is intended to get information.
- *Dest Port* is the destination Port where trace report must be sent.
- *Dest IP* is the destination IP address where trace report must be sent.

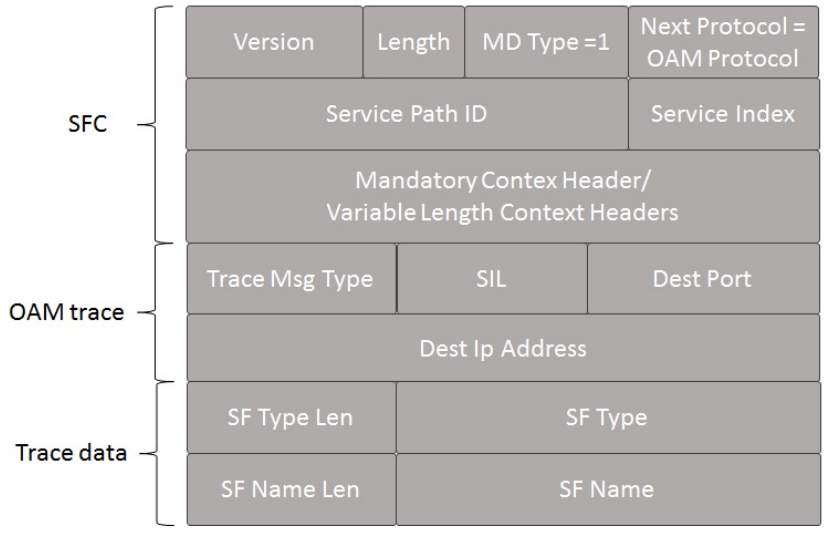


Figure 3.1: SFC Traceroute protocol [45].

When a Service Function receives an SFC *trace request packet*, it decrements the Service Index (SI), as for any other packet. If SI is equals to the *Services Index Limit* (SIL), SF adds its identifier information at the end of the existing headers as shown in Figure 3.1. Finally, the Service Function sends the packet back to SFF, regardless the SI value.

An SFF will route trace packets based on service path ID and service index just like any other NSH packet. This guarantees that a trace packet follows the same path as data packets. The SFF will drop the packet and generate a report when: *(i)* SI is equals to SIL, *(ii)* it cannot find the next hop, or *(iii)* SFF receives a trace packet with SI $= 0$.

A *trace report packet* carries the identification of the last SF that processed the packet. In all other aspects, the trace report is exactly the same as a trace request. SF Type and Name from Figure 3.1 are the fields to store Type and Name information of Service Functions.

To build a trace report packet, SFF will use the same encapsulation as the received packet and it will copy the entire header (NSH, trace request headers, and report section) from the received packet and send to destination IP and port defined in the OAM trace header. If an SFF cannot find the next service-hop for a trace packet, it will drop the packet and generate a report packet, even if SIL is different from SI. This guarantees that the trace ends at the end of the path irrespective if SI has reached SIL or not. Moreover, it allows users to perform a trace that will traverse the entire path without having to know beforehand the number of service-hops in the path by setting SIL to zero.

SFC Trace Issue Analysis and Solutions [59] is a draft to provide solutions for some undressed SFC Traceroute issues. SFC traceroute requires aware SFs, to interact with trace packets, otherwise, it will report as an error. SFC traceroute just triggers SFs and does not identify SFFs. These limitations are addressed by this work. SFC Trace Issue Analysis and Solutions proposes to change the adding of SF information in the trace request packet, from SF to the SFFs. Therefore, SFFs will be responsible for adding next hop SF or SFF information. To achieve this changes the OAM trace header from Figure 3.1 is modified including new fileds such as: Number Index (NI), to store the number of traversed chain hops and Last Service Index (LSI) to store the last processed service index. This approach is able to trigger both SFs and SFFs, skip unsupported SFs and take into account chain multipath by broadcasting trace packets.

SFC Traceroute is a complete solution to generate traces for an SFC enabled domain. However, it is only suitable for NSH encapsulation being incompatible to any other SFC techniques.

3.2 Revealing Middlebox Interference with Tracebox

Tracebox [16] proposes an extension to the widely used traceroute tool, that is capable of detecting various types of middlebox interference over almost any path. Tracebox sends IP packets containing TCP segments with different TTL values and analyses the returned ICMP messages. In addition, Tracebox can often pinpoint the network hop where the middlebox interference occurs.

To detect middleboxes, Tracebox uses the same incremental approach as traceroute, i.e., sending probes with increasing TTL values and waiting for ICMP time-exceeded replies. While traceroute uses this information to detect intermediate routers, Tracebox uses it to infer the modification applied on a probe by an intermediate middlebox. Tracebox detects changes on controller bits from TCP/IP headers to detect a possible interference of Middleboxes.

Although this is a useful tool to detect Middleboxes, it cannot detect transparent middleboxes that do not change L3/L4 headers fields. This tool does not consider an SFC environment, where traffic might be classified and reach different chains. In an SFC environment, this tool could just identify some SFs and cannot detect SFFs,

3.3 SDN traceroute: Tracing SDN Forwarding without Changing Network Behavior

SDN traceroute proposes a tool that can query the current path taken by any packet through an SDN-enabled network [11]. The path is traced by using the actual forwarding mechanisms at each SDN-enabled device without changing the forwarding rules being measured.

SDN traceroute employs low-overhead probe packets to measure network paths. It uses an algorithm to detect adjacent switches by tagging packets using VLAN priority field. Then, using the results of this algorithm, it installs a small number of high-priority rules in every switch in the network, which allows them to trap probe packets coming from neighbors switches. Every switch contains rules to detect adjacent switches and steer packets to the controller. SDN traceroute injects a probe packet into the network to start tracing the route.

Considering a switch perspective, the probe packets will be detected and sent to the controller when it came from any adjacent switch. The controller detects the switch hop and change the tag to the current switch, and then sends the packet back to switch. Therefore, the packet is normally processed until it reaches the next switch in the topology.

This is an interesting solution as it does not change the installed rules in the switches. However, it is not complete enough to trace packets in an SFC environment because it just detects switch hops and cannot detect middleboxes. In this solution, the probe packets must traverse the controller, what causes additional delay in the packet path. Therefore, it is not possible to measure latency throughout the chain path.

3.4 NetSight

NetSight [28] is an extensible platform that captures packet histories and enables applications to concisely and flexibly retrieve packet histories of interest. NetSight assembles packet histories using postcards-event records created whenever a packet traverses a switch. Each postcard contains a copy of the packet header, the switch ID, the output ports, and a version number for the

switch forwarding state. To generate postcards, NetSight prototype transparently interposes on the control channel between switches and controllers.

NetSight is implemented on servers and stores all network headers traversing the network switch hops. NetSight uses a regex-like language to concisely specify paths, switch state, and packet header fields for packet histories of interest. NetSight exposes an API for applications to specify, receive, and act upon packet histories of interest.

Atop of NetSight is built an interactive network debugger, called *ndb* [27]. The objective of *ndb* is to provide interactive debugging features for networks, analogous to those provided by *gdb* for software programs. The idea is to use *ndb* to help pinpoint the sequence of events leading to a network error, using familiar debugger actions such as breakpoint, backtrace, etc.

This solution takes into account SDN networks and intent to diagnostic bugs that affect the correctness of forwarding, including control logic errors, network race conditions, configuration errors, unexpected packet formats, and switch implementation errors. This work does not consider SFC environment and detection of service functions. It might be challenging to leverage this work for SFC backtrace generation, due its heavy implementation requiring dedicated servers to store all network headers.

3.5 Network Verification

Another related research field is the static analysis and verification of network environments that also can be used to perform verifications in the SFC architecture. There are initiatives [21, 33, 55] that perform static and symbolic analyses of the network data plane with the objective to detect connectivity and isolation errors from the network. These works build models out of network topology and exercise these created models with symbolic network packets to predict errors, network misconfiguration and reachability.

3.5.1 SFC-Checker

SFC-Checker [56] is an SFC troubleshooting and diagnosis tool with the objective of checking the forwarding behavior of an SFC. The goal is to examine whether flows are forwarded correctly according to the high level service chaining policies. SFC-Checker can check the sequence of NFs any flow should traverse, check the correct implementation of intent policies and check network function configurations. This work has the objective to a build static analysis framework to capture the problems before the deployment.

SFC-Checker leverages existing middlebox abstract models and generalize them to a forwarding model for network functions. Each NF is described using a flow table and a state machine. The temporal relationship between states are described using a Finite State Machine (FSM). SFC-

Checker also develops an algorithm for the static analysis of stateful networks. It proposes a Stateful Forwarding Graph (SFG) that encodes both the state transitions and forwarding behavior. Therefore, an algorithm is developed to automatically generates SFGs from Network Function tables and FSMs.

An SFC tool using network verification and static analyses is complementary to an SFC trace tool, which intends to check the actual behavior of a deployed SFC. Although SFC verification tools are useful to detect possible network failures, these initiatives try to predict errors before the network deployment or SFC deployment. On the other hand, an SFC trace tool intends to detect root causes of possible errors in an online manner.

3.6 Review

The presented related works reinforce that network packet trace is useful to detect network errors and reduce the time spent on the troubleshooting process. However, most of the tools do not take into account an SFC environment, while the SFC traceroute proposal (Section 3.1) is tightly coupled with the use of NSH header, which is just one technique that is far from being supported for hardware devices and VNFs. Moreover, none of these tools collects timestamp measurements, which is also useful to detect congested paths. An SFC tracer proposal should fill this gap in the SFC environment and generate traces agnostic to SFC technique. The collection of timestamp measurements enables the latency computations throughout the chain path. Table 3.1 shows a summarization of the related works analyzing the following aspects which are considered as requirements for a complete SFC tracer solution [12].

- the tool need to be adapted to fit in the SFC environment;
- the tool is able to collect timestamp measurements in each hop;
- the tool is able to detect NFs;
- the tool is able to detect SFFs;
- the tool can be deployed in an ordinary SFC deployment with no spread software agents nor extra equipments
- the tool can be used in a deployed chain in an online manner;

Table 3.1 also includes network verification works exemplified by SFC-Checker [56]. The Table shows if the specific aspect are compatible or supported by the tools, and a comment if some adaptation is needed.

None of the presented tools have the desirable features shown on the table regarding an SFC environment. The presented SFC tools have different scopes of the current proposal, such as

Table 3.1: Related work comparison.

Work	SFC environment awareness	collects timestamp	detects NFs	detects SFF	no extra agents nor equipments	on-the-fly measurements
SFC Traceroute [45]	just for NSH chains	no	yes	yes	yes	yes
Tracebox [16]	no	no	yes	no	yes	yes
SDN traceroute [11]	no	no	need to be added in the model	yes	yes	yes
NetSight [27]	no	yes	need to be added in the model	yes	no	yes
Network Verification / SFC-Checker [56]	yes	no	yes	yes	no	no

SDN networks or just the identification of middleboxes. SFC Traceroute is only suitable for NSH, SFC-checker is a tool for static analyses with the purpose to predict errors before SFC deployment. The remainder tools are no meant to be used in an SFC environment and may need hard adaptation to be used in this scope.

4. SFC PATH TRACER

This work aims to propose a solution for troubleshooting SFC-enabled domains through packet trace generation, called SFC Path Tracer. SFC Path Tracer uses tagged probe packets to generate path traces in an SFC environment. It is also able to count network packets per flow and measure latency along the chain path. SFC Path Tracer is agnostic to SFC encapsulation mechanism employed, enabling its utilization in different SFC implementations and providing a trace which includes all network components in the SFC path.

4.1 Architecture

Figure 4.1 shows the high-level architecture of SFC Path Tracer and where it is placed in a network deployment. In a controller/orchestration layer, *SFC Model* represents the SFC configuration including all its elements (SFF, SFs, SFP, etc). *SFC Driver* is an SFC implementation with the responsibility to read the configuration in the SFC Model and then install *SFC rules* in the *Network Elements*. SFC Path Tracer reads all *SFC forwarding rules* installed in the switches and based on that, it adds *trace rules* to mirror network packets. The trace generation will be triggered only by probe packets (*pp*). Therefore, probe packets that traverse a target chain will be mirrored to the trace tool. SFC Path Tracer is able to detect these probe packets (*Probe Packet Listener*). Moreover, it analyses probe packets header information and compares it against the SFC configuration to generate the trace (*Trace output*). Therefore, it is possible to identify chain hops elements and then generate a meaningful trace, with switches and network functions names as they were configured in the SFC Model.

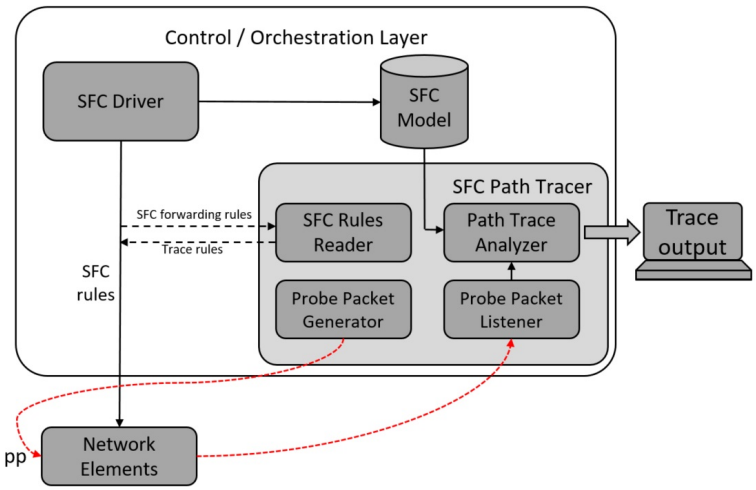


Figure 4.1: SFC Path Tracer architecture.

In order to be detectable, probe packets must be flagged. A probe packet will traverse a target chain as any other network packet. However, whenever a probe packet leaves a forwarder switch to its next hop, it is also mirrored to the trace tool. Probe packets can be identified in

many ways, using optional header fields or encoding meaningful information in the L2 header. The probe packet is used to identify a specific traffic and consequently get its trace. The use of probe packets allows a filter mechanism to restrict the steering of network packets to the SFC Path Tracer. Therefore, avoiding possible saturation of the channel that connects switches with the trace tool.

4.2 Use case

Service Function Chaining has several types of use cases. In all use cases, an SFC trace tool provides valuable data that eases the troubleshooting process. In order to exemplify the SFC Path Tracer use case, we chose a common service based on SFC use cases for network security [36]. Figure 4.2 shows a use case where a chain is configured to reach the *Firewall* and *IPS* (Intrusion Prevention System).

From Figure 4.2, the client sends probe packets and, through trace rules, installed by SFC Path Tracer, probe packets are mirrored to the controller in every chain hop. In the controller, SFC Path Tracer gets information from the received packets and identifies the switch ID and the next hop. It uses this information to compare against the configured SFC model and then generate a packet trace. The expected behavior from the example of Figure 4.2 is the following: traffic is classified by *SCF*, then it traverses the configured chain ending in a server.

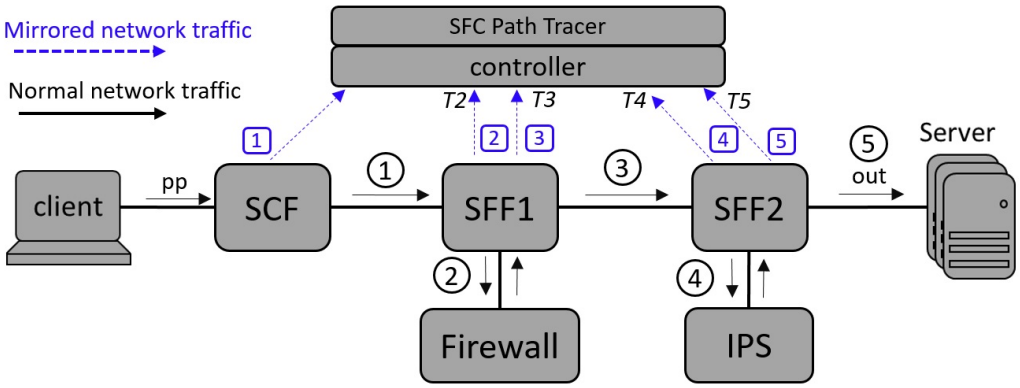


Figure 4.2: SFC Path Tracer example use case.

The continuous line, shown on Figure 4.2, means probe packets traversing the network elements, while the dotted line represents the mirrored probe packets being sent to the controller in order to generate the trace path. Whenever a probe packet leaves a switch, it is also mirrored to the controller. The packet flow sequence is identified by numbers in the figure. Each hop latency is computed by the difference between measured timestamps. For instance, $T3 - T2$ results in the Firewall latency while $T5 - T4$ results in the IPS latency.

The resulting path trace prints all SCF, SFs and SFFs reached by the probe packets. The trace output in the example of Figure 4.2 is $SCF \Rightarrow SFF1 \Rightarrow Firewall \Rightarrow SFF1 \Rightarrow SFF2 \Rightarrow IPS \Rightarrow SFF2$. In case something is wrong with the Firewall, SFC Path Tracer updates the trace

output just while the packet is traversing the chain and stops at the problematic SF. Therefore, it is possible to identify the SFC path region that is not working.

The delay for mirrored packets to travel from switches to the controller is ignored, since this delay will be added for both $T2$ and $T3$ from the previous example. The purpose of SFC Path Tracer is to pinpoint problems to help network operators to troubleshoot an SFC environment. SFC Path Tracer is not designed to be constantly executed and give precise latency information. SFC Path Tracer provides the exact path trace traversed by probe packets. Therefore, network operators could rapidly identify the root cause of a chain problem and save time in troubleshooting process.

4.3 Implementation remarks

SFC Path Tracer is implemented in the SDN controller layer. OpenDayLight [2] is the chosen platform. OpenDayLight (ODL) is an open source controller with several contributors from many companies and largely used in the SDN/NFV area. SFC Path Tracer is implemented as a plugin in the ODL controller under the ODL SFC project [8]. The SFC modeling schema from ODL allows the coexistence of multiple SFC forwarding implementations over different southbound protocols. SFC Path Tracer design was evaluated using SFC implementation that uses OpenFlow to configure switches as SFFs and SCFs in the SFC domain.

Since ODL and OpenFlow are used to evaluate SFC Path Tracer design, the OpenFlow channel is used to send mirrored packets to the controller where SFC Path Tracer is running. ODL's abstraction layer is used to watch switch rules installation and read the SFC model configuration. OpenDayLight basic concepts, used to implement SFC Path Tracer, are presented in Section 2.6.

4.4 SFC Path Tracer implementation

SFC Path Tracer is developed as a plugin in the ODL-SFC project and it does not have a specific model of its own. However, a YANG file is created to define trace RPCs to query trace information or clean old trace data. SFC Path Tracer interacts with other ODL modules through the MD-SAL layer, such as the OpenFlow plugin to install trace rules and SFC Model to get chain hop information for the trace generation. SFC Path Tracer was initially evaluated using VLAN OpenFlow renderer in ODL-SFC. This technique implements SFC for SF connected through L2 connectivity (Section 2.6.3). Based on SFC Path Tracer architecture, the tool implementation is divided into four parts: probe packet generation, trace rule installation, SFC rules reader and probe packet listener.

4.4.1 Probe packet generation

Trace generation is triggered by specific probe packets which are flagged in order to be identifiable. In this implementation, probe packets are flagged by an IP header field that is not commonly used for regular traffic and can also be used as a match field in OpenFlow rules. The used field is the 2-bit Explicit Congestion Notification (ECN), shown in Figure 4.3. This field is part of a modern redefinition of 8-bits ToS (Type of Service), which now is composed of 6-bits DSCP (Differentiated Services Code Point) and the 2-bits ECN. The 2-bits set from ECN will trigger OpenFlow rules to send packets to the controller.

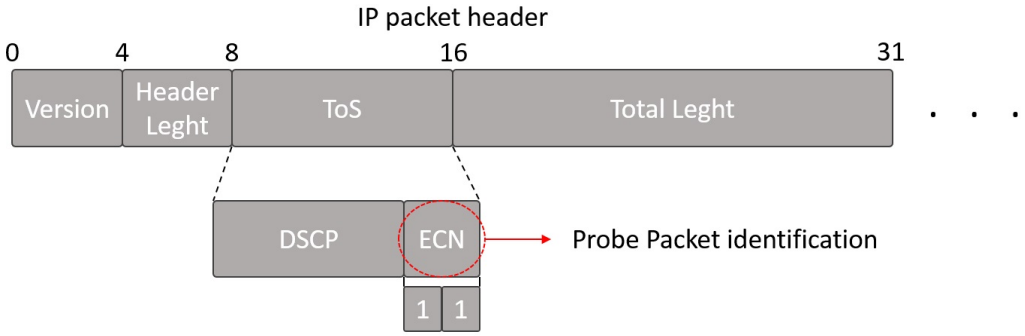


Figure 4.3: Probe packet identification.

Another justification to use a flagged packet is the necessity to filter packets or flows and send it to the SDN controller. Otherwise, all traffic would have to be steered to the controller, which could saturate the OpenFlow channel.

In order to generate an SFC trace for a specific chain, probe packets can be generated in two ways, by the controller or in the chain input. The controller can artificially inject probe packets in the chain input to generate the trace. A client placed behind the classifier, as chain input, can send probe packets. Probe packets can be sent as an unique packet such as pings or can also be part of a normal network traffic, with the classifier inserting the flag in a specific traffic flow.

In the case of inserting probe flags in a traffic flow, a sampling mechanism might be necessary for higher throughput, to avoid saturation of the tool. The sampling mechanism can be achieved in multiples ways. The SDN controller can dynamically create and remove OpenFlow rules to add the probe packet flag in the packets. Techniques to watch the OpenFlow counter [10] can also be used. Therefore, the OpenFlow counter of the sampling rule is read, and after a predefined number of hits the rule is erased. Sampling implementation in hardware switches such as *sFlow* [48] can be used to send packets to SFC Path Tracer. sFlow is a multi-vendor sampling technology embedded within switches and routers. It provides the ability to continuously monitor application level traffic flows at wire speed.

Probe packets might be sent over any transport layer that supports IP packets such as UDP, TCP, or even ICMP, depending on the type of traffic the network functions are able to handle. It is assumed that SFs being traced will not drop those probe packets. The use of ECN fields to

identify probe packets is an implementation choice of SFC Path Tracer in order to get the least impact. For services that make use of ECN bits the probe packet can be identified in an alternative way, for instance, encode information in the MAC address or tunnel optional fields (*e.g.* VLAN, VxLAN). Others probe packet identification techniques may also have its limitations. The probe packet identification could also be configurable depending on the type of traffic being traced.

4.4.2 Trace rules installation

SFC Path Tracer configures the generation of a trace for a specific SFC path by listening to the installation of SFC forwarding rules, *i.e* rules installed on *Transport Egress* table. SFC Path Tracer detects those rules and adds new trace rules with higher priority (p), as shown in Figure 4.4. SFC implementation on OpenDayLight defines a pipeline of switch tables that will process network packets. Each OpenFlow table has its purposes, such as transport ingress, path mapper, next hop decision and finally the transport egress. SFC Path Tracer detects those transport egress rules and adds the trace rules. The new trace rules are copies of the original egress rules, but with slightly modified matches and actions.

In the OpenFlow match, it is added the probe packet identification, in this case, the ECN field, in order to filter just probe packets. The OpenFlow action is changed to decrement IP TTL (time to live) field, write the output forward port into the OpenFlow metadata field, and add an action to forward the packet to the next table, defined as trace table. Since probe packets may reach the controller through different paths and from different switches, TTL is decremented to guarantee the trace ordering among hops. The output port is encapsulated into the OpenFlow metadata field in order to trace the next hop. The information encapsulated in the metadata fiel is used to query SFC configuration and discover the next service function destination.

Important to noticed that OpenFlow actions are applied in the specified order. Besides the original SFC actions, the TTL is decremented and the probe packet is forwarded to its original destination. However, the next actions write the port in the metadata field and forward packets to the trace table. Therefore, in this moment the probe packet is mirrored. On the trace table, the mirrored packet is matched again by ECN bits and sent to the ODL controller, where SFC Path Tracer will be listening. Figure 4.4 shows the ODL-SFC OpenFlow tables with the added trace rules highlighted in bold.

As SFC trace Monitor was initially evaluated using SFC VLAN OpenFlow renderer, the trace rules shown in Figure 4.4 considers this SFC technique. For other SFC techniques, some slight adaptation might be necessary.

Table	Match		Action
Next Hop	P	Do not care	Go to Transport Egress table
Transport Egress	**p = n+1**	**Next hop identification + ECN == 3**	**dec_ttl + Output : switch port + Metadata = switch port + go to trace table**
	p = n	Next hop identification	Output : switch port
Trace table	**p**	**ECN == 3**	**Output : controller**

Figure 4.4: OpenFlow rules installed by SFC Path Tracer.

4.4.3 SFC rules reader

Figure 4.5 shows the SFC implementation schema in ODL with the SFC Path Tracer module plugged in. Network operators configure the *SFC model* [60] in ODL by entering information regarding all SFC elements defined on the SFC architecture [26]. Once the *SFC model* is configured, the *RSP RPC* (Rendered Service Path) is called to actually deploy the configured chain. *SFC implementations* read the configured SFC model, detects which SFC technique should be used and generates the OpenFlow rules to configure SFFs and SCFs. Figure 4.5 highlights VLAN OpenFlow renderer being selected as SFC implementation.

As the OpenFlow plugin is used, the generated SFC rules are sent to the *OF data store* on the ODL MD-SAL layer. *OF data store* keeps the data related to the OpenFlow plugin. *OF-plugin* is responsible for listening any modification in *OF data store* and updating these changes in the switches.

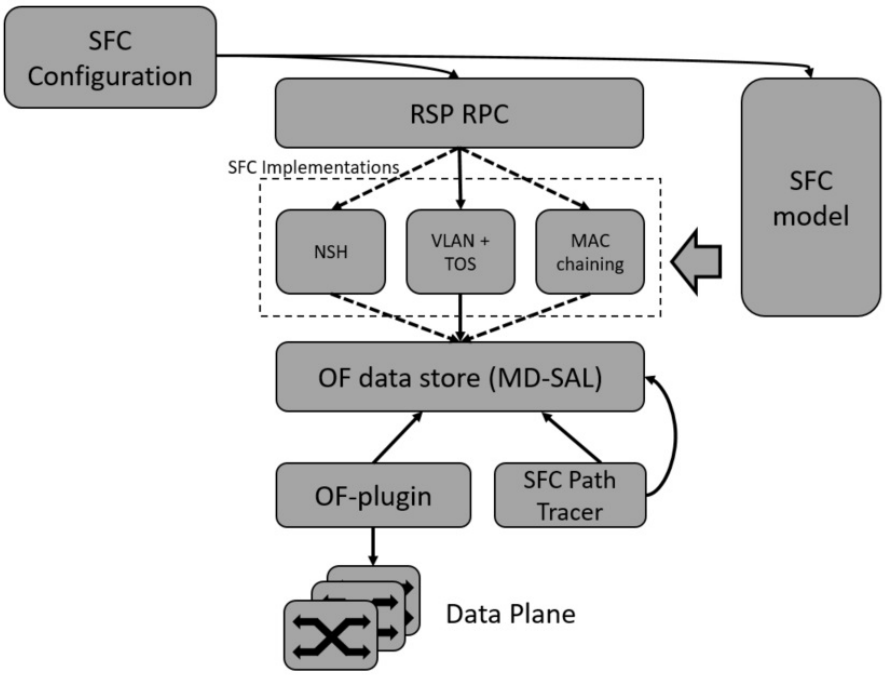


Figure 4.5: Implementation of SFC Path Tracer rules. Dotted arrow represents possible paths while continuous arrow represents the chosen path.

As shown in Figure 4.5, SFC Path Tracer also listens to OF data store changes looking for SFC transport egress rules. These rules are detected by the OpenFlow cookie of installed rules. OF cookie is an opaque identifier of OF rules defined on the rule installation. When detected, transport egress rules are used to create the new trace rules with the added match and action fields. Afterward, SFC Path Tracer writes the new rule back in the OF data store. *OF-plugin* detects the OF data store changes and updates switches with the extra trace rules.

4.4.4 Probe packet listener

SFC Path Tracer must listen to packets coming into the ODL controller. SFC Path Tracer registers a listener for incoming packets. This listener filters just probe packets flagged with ECN bits. Figure 4.6 shown the fluxogram of probe packet listener. Once probe packet is identified from packet-in, the packet is parsed to retrieve information such as the switch that sent the packet to the controller, OpenFlow metadata and IP/MAC addresses. SFC Path Tracer uses these retrieved information as inputs to read the SFC configuration and discover chain hops. From the switch that sent the packet to the controller, it is discovered which Service Function Forward handled the packet. From the output encapsulated in the OpenFlow metadata field, the next hop destination (next SF) is detected. SFC Path Tracer knows the SFs and SFFs by looking the SFC data model and finding the SFF name and the SF connected to the specific switch port. It is worth noting that more information can be used from the income probe packet in the controller. For instance, if the port is not present in the SFC configuration, the search might be done looking for the MAC address from a specific SF.

Once the elements are found in the configured SFC model, the ID field from IP header is read to uniquely identify a packet and create a new element in a *trace map*. According to RFC 6864, the IP identification field is a 16-bit value that is unique for every datagram for a given source address, destination address, and protocol. Therefore, it is possible to identify a single packet among a traffic flow enabling the use of concurrent probe packets.

The IP identification field is used to disambiguate packets in order to handle concurrent probe packets. The trace map associates IP ID to a traced path. Therefore, the hop information is stored in the trace map by each IP ID. In this map, every traced hop is kept linked with its input timestamp, in other words the timestamp when packet-in reached the controller.

Through a *Trace RPC*, the trace map is read in order to merge identical stored paths into the SFC *trace output*. In this output map, single traced paths are stored together with the counter of packets and the timestamp of each chain hop. The collected timestamps are used to compute the latency spent in each hop. Since IP ID field can identify 65536 packets, the process of merging identical stored paths are done periodically in order to clean the trace map and be able to identify the same IP ID from another packet.

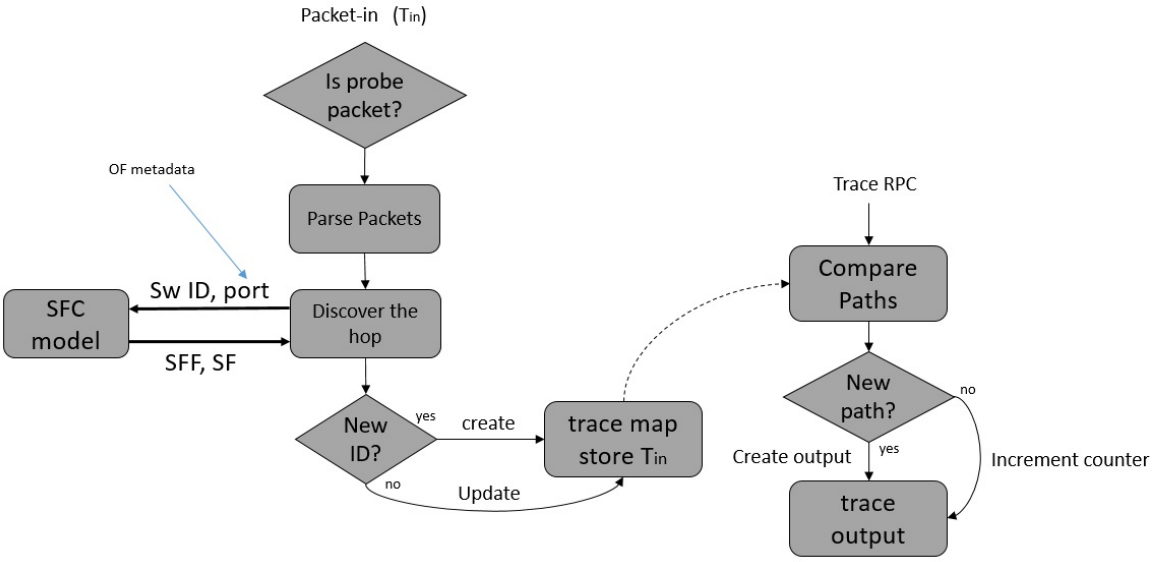


Figure 4.6: Implementation of probe packet listener.

4.4.5 SFC Path Tracer outputs

The output trace format is identified as $[n][p_1 - SFF - p_2] - [SF]$, where n is the number of packets, p_1 is the ingress port in the switch and p_2 is the egress port. If the egress port is connected to an SF, then the SF will be traced, otherwise, this hop is done and the process is repeated for the next hop. For each hop, SFC Path Tracer also prints the computed average latency among all received probe packets. Figure 4.7 shows an example of trace output for a chain segment. In this example, seven probe packets were detected and tree hops were traced, informing the switch ports, the next SF and the average latency for each hop.

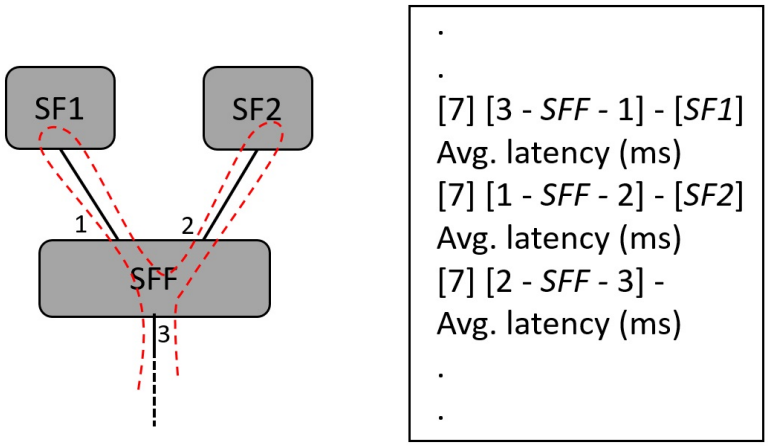


Figure 4.7: Example of SFC Path Tracer output.

Although the average latency is printed in the trace output, SFC Path Tracer keeps the hop latency information for every received probe packet along with its timestamp. This information is written in trace files in order to generate output graphs over time. For instance, in the example

of 4.7, a trace file is generated with seven entries for each hop containing the timestamps of each probe packet and the hop latency computed against the previous hop timestamp.

SFC Path Tracer provides two set of data, the latency and the number of probe packets arrived in the controller. SFC Path Tracer computes the incoming probe packet rate per second (pp/s) in order to prevent a possible saturation of OpenFlow channel. In order to plot graphs, SFC Path Tracer computes averages for every step in the graphs. Therefore, the output trace files are generated with 100 entries, which enable to plot graph with 100 steps.

4.5 Service Functions compatibility with SFC Path Tracer

SFC Path Tracer works with any SFC technique that installs rules in SDN switches, being compatible with a variety of network function types. The tool does not require any agent from SF side. SFC Path Tracer technique allows the utilization of any IP packet that carries a probe flag, *e.g.* ECN, which can be used in most transport protocols. It is just required that SFs do not drop those packets, otherwise the probe packet flagging mechanism should be done using an alternative approach such as encoding the probe in the MAC addresses. The probe packet information could also be configurable, given the network operator the possibility to choose where the probe packet flag will be encoded.

SFC Path Tracer may present compatibility issues with SFs which open new output connections such as TCP proxies. In this case, the probe packet flag encoded in the packet header will be lost and the next SFF will not recognize it as a probe packet. SFC Path Tracer also relies on IP ID field to unique identify a packet, which in this case would also be lost. However, this can be seen as an issue of the SFC technique itself. The SFC technique must have a mechanism to link input and output of an SF in order to identify that the packet belongs to a specific chain. The link between input and output could be done using outer headers such as encapsulation tunnels. SFC Path Tracer can be adapted to act in this link level, using special bits from the tunnel mechanism to encode the probe packet flag and identify the packet.

4.6 SFC Path Tracer in the SFC OAM framework

Operation, Administration and Maintenance (OAM) for SFC [12] discusses tool gaps to perform OAM function on an SFC environment. The gaps are related to verifying that there is connectivity between network elements in the chain and the continuity of those elements. Continuity is a model where OAM messages are sent periodically to validate or verify the reachability to a given SF or SFC. Performance and trace functions are also discussed as important functions to achieve the OAM for SFC. Some of these gaps can be filled by SFC Path Tracer. Table 4.1 shows this SFC tools gap analyses presented in the OAM for SFC [12]. The original analyses from SFC-OAM presents

this table with tool gaps for all SF and SFC functions. Table 4.1 pinpoints where SFC Path Tracer can be leveraged to achieve the required SFC-OAM functions.

Table 4.1: OAM Tool gap Analysis [12].

Layer	Connectivity	Continuity	Trace	Performance
Network Overlay	Ping	BFD [32]	Traceroute	IPPM [44]
SF	None	SFC Path Tracer	SFC Path Tracer	SFC Path Tracer
SFC	SFC Path Tracer	SFC Path Tracer	SFC Path Tracer	SFC Path Tracer

Existing tools used for network overlay does not work within the SFC environment. SFC Path Tracer can be used to analyze continuity of a chain, with the trace generation and compute the latency of each chain hop. With a trace generation, it is possible to infer the SFC connectivity, however, SFC Path Tracer was not designed to test the connectivity of individual service functions. Important to notice that SFC Path Tracer is designed to be a troubleshooting tool to assist network operator to find problems in the chain path. SFC Path Tracer is not suitable to be executed permanently, due to the additional traffic in the OpenFlow channel. However, SFC Path Tracer can be leveraged to monitor SFC paths with lightweight probe pings over time.

SFC Path Tracer provides a complete solution to generate packet traces in an SFC environment. It is agnostic to the SFC encapsulation mechanism employed, enabling its utilization in different SFC implementations, as opposed to SFC Traceroute [45]. It also provides a trace which includes all network components in the SFC path, in contrast to Tracebox [16] and SDN Traceroute [11] approaches.

5. EVALUATION

SFC Path Tracer was firstly evaluated using the SFC VLAN technique, implemented in OpenDayLight as an OpenFlow renderer. MAC Chaining and NSH techniques are also used to demonstrate the SFC Path Tracer compatibility. In order to generate traces, individual probe packets and regular HTTP load are used to traverse target chains. Auxiliary tools are used to generate the network traffic, such as *hping3* [1], *Tsung*[2] and *iperf*[3].

In order to evaluate SFC Path Tracer and easily create different SFC scenarios, a framework to emulate SFC topologies was created. These topologies may be formed by multiple hops with different SFFs and SFs enabling the configuration of multiple chains. With this framework, it was possible to create SFC topologies using SFC VLAN and MAC Chaining techniques. Both techniques rely on L2 connections between SFFs and SFs. In NSH case, which is implemented using VxLAN tunnels, SFC Path Tracer was evaluated in a different scenario, using separate VMs to run the SFF and each SF. The evaluation framework is created upon Mininet and OpenVswitch.

5.1 Mininet

Mininet is a system for rapidly prototyping large networks in a single laptop [35]. Mininet uses the lightweight virtualization mechanisms built into the Linux OS (Operating System), processes running in network namespaces, and virtual Ethernet pairs. This allows Mininet to scale up to hundreds of nodes. The main goal of Mininet is to support studies in the SDN domain, enabling self-contained SDN prototypes in a single PC. With Mininet, users can implement a new network feature or entirely new architecture, test it on large topologies with real traffic, and then deploy the exact same code and test scripts into a real production network.

Mininet provides a rapid prototyping workflow to create, interact and customize a software-defined network. With a simple command line interface (CLI) it is possible to create a network with SDN controllers, switches and hosts. The main components of Mininet are [35]:

- *Links* are virtual Ethernet pair or veth pair, they act like a wire connecting two virtual interfaces. Each interface appears as a fully functional Ethernet port to all system and application software. Links may be attached to virtual switches or a software OpenFlow switch.
- *Hosts* are network namespaces or containers [38] for network state. Containers provide processes with exclusive ownership of interfaces, ports, and routing tables. For example, two web servers in two network namespaces can coexist in one system, both listening to private eth0 interfaces on port 80. A host in Mininet is simply a shell process moved into its own container. Each host has its own virtual Ethernet interfaces, with its own IP and MAC addresses.

[1] http://www.hping.org/
[2] http://tsung.erlang-projects.org/
[3] https://iperf.fr/

- *Switches* are based on software and provide the same packet delivery semantics that would be provided by a hardware switch. Both user-space and kernel-space switches are available. An example of a virtual switch is Open vSwitch [43]. It is an open project widely accepted in the network virtualization area.

- *Controllers* can be anywhere on the real or simulated network, as long as the machine on which the switches are running has IP-level connectivity to the controller. The controller can run inside a virtual machine, natively on the host machine, or in the cloud.

Figure 5.1 shows an example of a Mininet network emulation running in the Linux operating system. An external SDN controller is connected via $eth0$, and user-space OpenFlow switch connects both $h2$ and $h3$, working as hosts in containers.

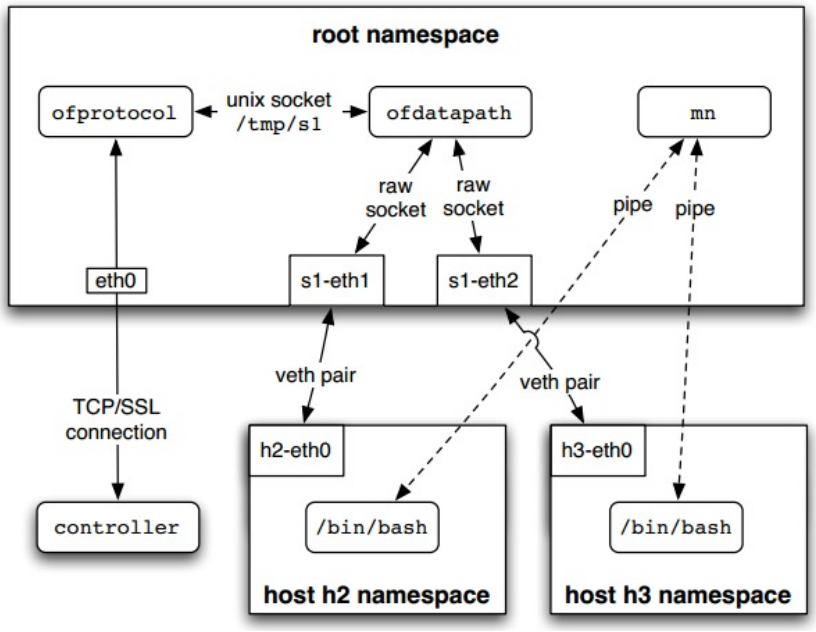


Figure 5.1: SDN environment example emulated by Mininet [35].

5.2 Open vSwitch

Open vSwitch (OVS) [43] is a multi-layer open source virtual switch for all major hypervisor platforms. Open vSwitch is an OpenFlow switch. It is deliberately not tied to a single purpose, tightly vertically integrated network control stack, but instead is re-programmable through OpenFlow [47]. It has user and kernel space implementations as shown in Figure 5.2. The first packet of specific flow results in a miss and the kernel module directs the packet to the userspace component, which caches the forwarding decision for subsequent packets into the kernel. The controller might be external, such as OpenDayLight. Open vSwitch is open to programmatic extension and control using OpenFlow and the OVSDB [46] management protocol.

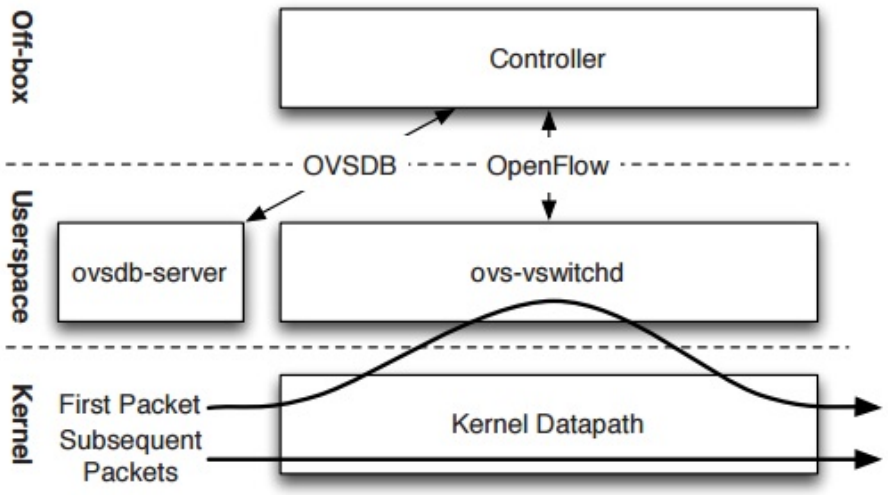


Figure 5.2: The components and interfaces of Open vSwitch [47].

5.3 SFC configuration framework

In order to create an evaluation environment, a framework was built to create topologies and configure chains on a single virtual machine. The SFC configuration framework consists in: (i) creating an emulated environment with Linux containers and (ii) configuring SFC RESTCONFs on ODL. The objective of this framework is to facilitate the process of SFC topology creation, allowing the rapid deployment of multiple chains on different topologies. SFC configuration framework uses the information of the built topology to generate SFC RESTCONFs to configure ODL with the desirable chain. The test environment contains the following components:

- *OpenDayLight (ODL)* is used as the SDN Controller to configure the virtual switches (SFFs and classifiers) to perform the chaining process.
- *Open VSwitch (OVS) [43]* is responsible for emulating the network switch components and connecting hosts and service functions on the topology.
- *Mininet [35]* provides the network topology and containers, which are used to emulate hosts and service functions. Mininet builds network topologies using OVS as the virtual switch and ODL as the controller.
- *Dummy service functions* are implemented to test service chains. Dummy SFs receive a network packet and just forward it to the next SFF, performing no modifications in the packet. The dummy service function was implemented using raw sockets, which is responsible for listening network packets and forwarding them back.

Figure 5.3 shows the test environment structure using the described elements. On top of this infrastructure, the *SFC configuration framework* was built. It is responsible to easily define a

network topology interfacing with Mininet to create *SFs* and *hosts*. Hosts worked as chain input and output, for instance, a client communicating with a server throughout a chain. Based on the configured topology, SFC configuration framework generates RESTCONF APIs to configure ODL and configuring the SFs, SFFs, SFPs, SCF and the chain. In ODL controller, the *SFC Driver* reads this SFC configuration, and based on that installs OpenFlow rules in *SFFs* and *SCFs*.

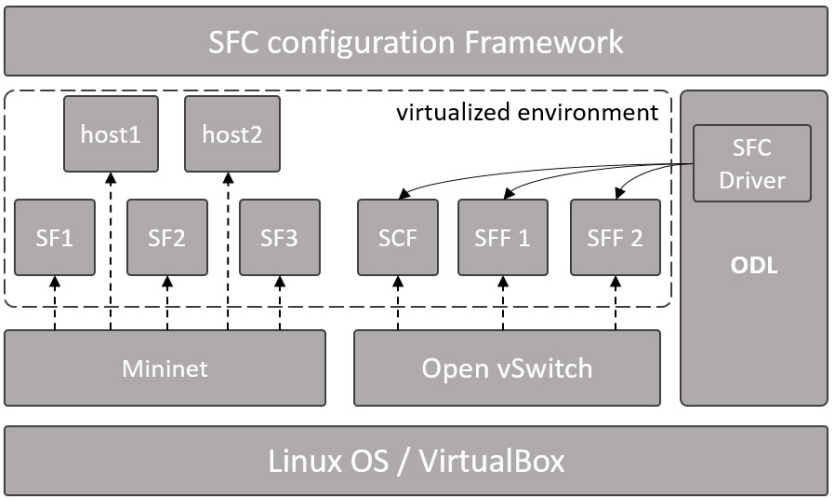


Figure 5.3: Architecture of SFC configuration framework.

The SFC configuration framework implements an SFC class that handles Mininet APIs available in *Python*. An API was developed to configure the chain topology, which is responsible to set up the Mininet topology and build RESTCONFs to configure ODL-SFC. Table 5.1 show all methods from SFC configuration framework.

Taking advantage of the SFC configuration framework, it is possible to easily define a variety of topologies such as shown in Figure 5.4. The figure exemplifies the use of SFC configuration framework illustrating the framework calls to fully configure an SFC environment. Firstly, (1) *SFC* class is instantiated with SFC encapsulation type and the controller IP, (2) the switches are added (*sff* and *scf*), (3) the hosts are added to be used as chain input and output (*h1* and *h2*), (4) *sf1* and *sf2* are added and connected to the *sff*. The SFs are associated to a type of function (*e.g.* IPS, Firewall) which later will be used to define the chain. Function *addLink* (5) creates a link between switches to form a topology. In order to terminate the chain and deliver the traffic to the original destination, (6) a gateway is created and connected to the classifier. Array *chain* (7) represents the chain itself, formed by a list of SFs. Then, (8) the chain is added using *addChain* function informing the chain name and the input classifier. Finally, (9) the chain environment is executed, where the Mininet topology is actually created and the built RESTCONFs are deployed on ODL.

Figure 5.5 shows the resulting topology created by the framework code shown in Figure 5.4. This topology was used to perform initial tests of SFC Path Tracer. The topology consists of a service classifier function (SCF) which will classify network traffic to traverse a chain. The chain is formed by *SFs* connected to the *SFF*, in this case, *SF1* $\Rightarrow$ *SF2*.

Table 5.1: Methods from SFC configuration framework.

SFC configuration methods	Description	Parameters
SFC()	SFC topology class is responsible to handle Mininet topology calls	(1) SFC encapsulation, (2) ODL IP
addSw()	Adds an SFF or SCF and builds related RESTCONF for ODL-SFC	-
addHost()	Adds a host to work as chain input/output	(1) switch where this host is connected
addSf()	Adds an SF and builds related RESTCONF for ODL-SFC	(1) switch where this SF is connected, (2) type of SF
addLink()	Adds a link between two switches and updates related RESTCONF for ODL-SFC	(1) switch 1, (2) switch 2
addGw()	Adds a gateway to terminate the SFC	(1) switch where this gateway is connected
addChain()	Adds an ordered list of SFs (SFC) and builds related RESTCONF for ODL-SFC	(1) chain name, (2) classifier, (3) list of SFs
deployTopo()	Starts the Mininet topology and deploys the configured chain	-

```
sfc = SFC(sfcEncap.VLAN, ctr.IP)          #(1)

scf = sfc.addSw()                         #(2)
sff = sfc.addSw()

h1 = sfc.addHost(scf)                     #(3)
h2 = sfc.addHost(scf)

sf1 = sfc.addSf(sff, 'fw')                #(4)
sf2 = sfc.addSf(sff, 'ips')


sfc.addLink(scf, sff)                     #(5)
sfc.addLink(scf, scf)

sfc.addGw(scf)                            #(6)

chain = ['fw', 'ips']                     #(7)

sfc.addChain('c1', scf, chain)            #(8)

sfc.deployTopo()                          #(9)
```

Figure 5.4: Code to create an SFC topology with SFC configuration framework.

5.4 Experiments description

SFC Path Tracer was evaluated using the SFC VLAN technique. The experiments were executed over the SFC configuration framework. The experiments used to evaluate SFC Path Tracer are:

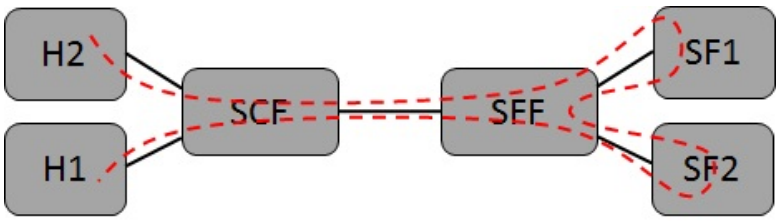


Figure 5.5: Topology used to perform initial tests.

- Experiment 1, in Section 5.5, evaluates *Probe packet delay* using forged packets with probe flags. Configuring a chain with multiple hops, normal and probe packets are sent to compute the extra delay of probe packets. Therefore, it is possible to detect the impact of mirroring packets to the controller.
- Experiment 2, in Section 5.6, presents a *troubleshooting evaluation* over a variety of topologies to guarantee trace output correctness. Also, the trace output is shown in a health chain and with a chain containing a failed SF.
- Experiment 3, in Section 5.7, evaluates the *chain performance* computing hop by hop latencies using HTTP traffic. An Artificial delay is induced in network links to assure the latency measurement correctness. The experiment is performed with a chain containing a real IPS as one of the SFs. Also, the latency distribution over time in the IPS is shown. The load in the IPS is changed in order to demonstrate that SFC Trace Monitor can detect the latency variation of an overloaded SF.
- Experiment 4, in Section 5.8, presents the *sampling of probe packets* in order use SFC Path Tracer with higher throughputs. Sampling is used with the objective to not mirror all packet of a traffic to the controller and then avoiding saturation of OpenFlow channel. The CPU load for sampled and non-sampled traffic is shown to observe the resource saving. The output of latency distribution is compared considering sampling and non-sampling packet mirroring.
- Experiment 5, in Section 5.9, computes the *probe packet rate limitation* in order to avoid probe packet dropping. SFC Path Tracer computes the incoming probe packet rate over time with the objective to alert when there is a high rate of probe packet being sent to the controller, what can cause wrong trace measurements.
- Experiment 6, in Section 5.10, evaluates SFC Path Tracer *with other SFC encapsulation* to demonstrate the tool support for multiple SFC techniques. SFC Path Tracer is evaluated with MAC Chaining and NSH.

5.5 Experiment 1 - Probe packet delay

In order to evaluate the delay introduced on packets that are being traced by SFC Path Tracer, single probe packets were sent through a chain and compared with normal packets that are

not being traced. Using the SFC configuration framework, a topology is created with the objective to range the number of hops in a chain in order to observe probe packet delays with multiple hops. The SFC encapsulation technique used in the experiments is the SFC VLAN OpenFlow renderer from OpenDayLight.

Figure 5.6 shows the largest chain used in the evaluation. This topology is formed by a classifier and three SFFs switches, connecting three SFs each. In order to adjust the number of hops in the chain, SFs were removed for each measurement until the smallest chain were formed just by *SFF1* and *SF1*. The numbers in Figure 5.6 represent the chain hops.

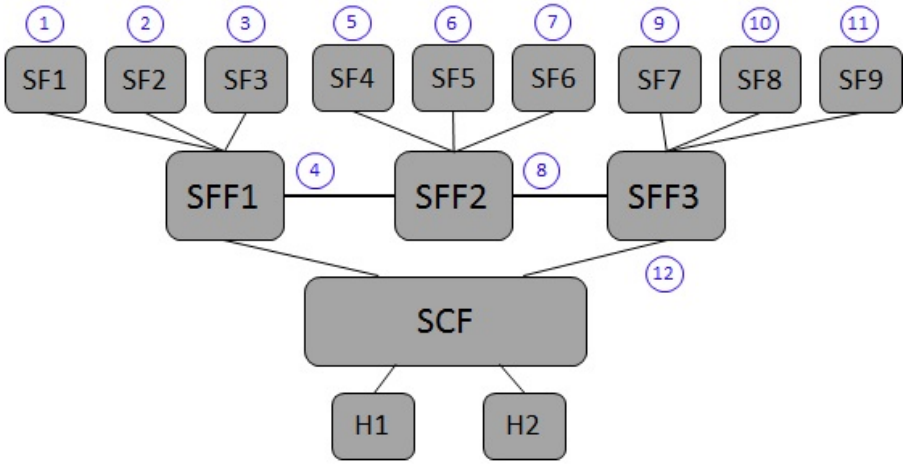


Figure 5.6: Topology used in the experiment.

Probe packets are generated using *hping3* tool. This tool can create custom packets to ping devices via UDP, TCP or ICMP. In the context of the experiments, *hping3* is used to create probe packets, *i.e.* IP headers with ECN bits set to one. Besides that, the probe packet carries an UDP payload that targets a closed port. According to RFC 792, if a source cannot deliver a datagram to a protocol module or process port, the destination may send an ICMP Port Unreachable back to the source. The experiments use this message to calculate the Round-Trip Time (RTT), and thus obtain the total latency of the chain.

In order to measure the delay, an UDP packet is sent through an unidirectional chain from $H1$ to $H2$. Figure 5.7 shows the RTT for normal packets (np) that were sent over the chain with ECN untouched and probe packets (pp) with ECN bits set. It was also evaluated the approach used by SDN Traceroute [11] (ppc). In this approach, probe packets are sent to the controller and forwarded back to the switches. Therefore, the controller is located in the direct path of probe packets, since they are not mirrored. The amount of time for packets to traverse the chain is computed collecting RTT measurements from an average 1000 pings executed in 10 cycles of 100 pings.

The additional delay of pp, compared to np, represents the cost of copying those packets in the switches in order to send it to the controller. Since this evaluation runs on virtual switches, this copy is notable. On hardware switches, the probe packet mirroring would have significant smaller cost. Although there is a small extra delay for pp, this delay is significant smaller compared to SDN

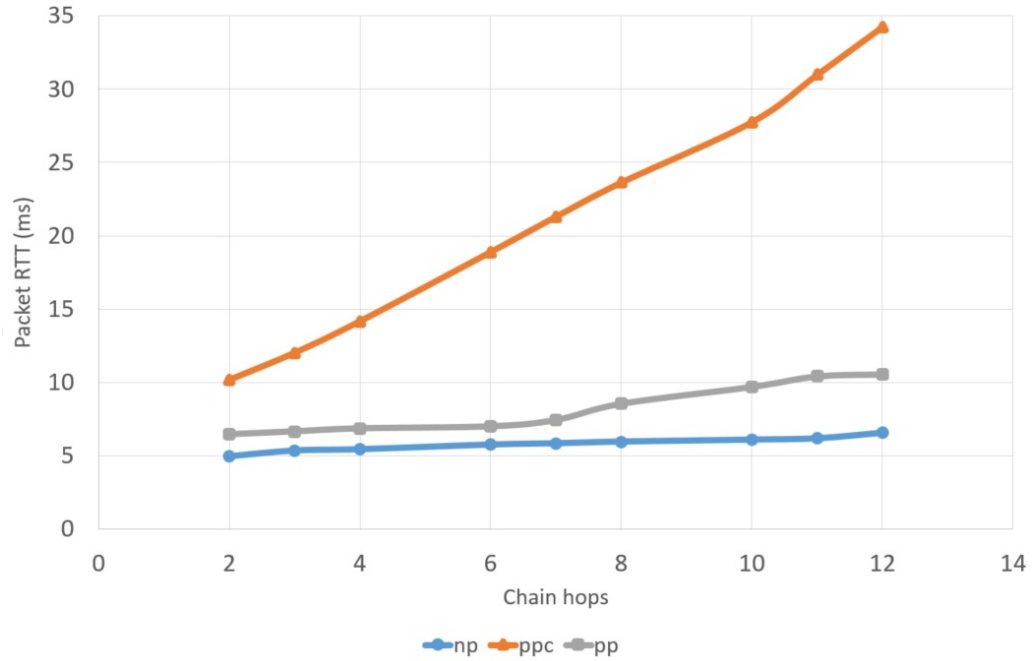
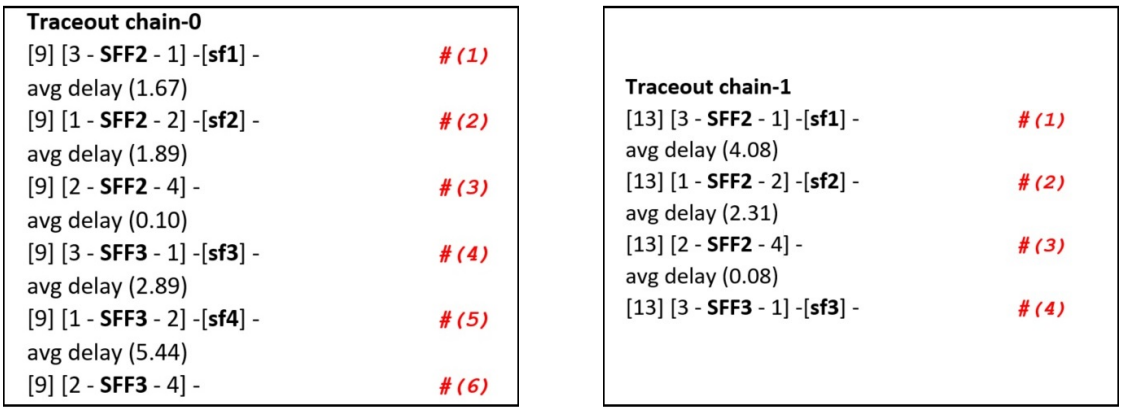


Figure 5.7: Packet delay with a varying number of chain hops: normal packets (np), probe packets (pp) and probe packet with controller in the path (ppc).

Traceroute approach (ppc). While the average of additional delay for ppc is around 2.3 ms per hop, for pp is approximately 0.3 ms per hop. The extra cost of ppc refers to the time probe packets to reach the controller and get back on switches. The fact that SFC Path Tracer does not introduce the controller in the direct path of probe packets enables the possibility to flag normal network packets, being able to generate traces of specific flows from real traffic.

5.6 Experiment 2 - Troubleshooting evaluation

SFC Path Tracer tool is useful to troubleshoot common SFC problems such as misconfigured policies and SFs connectivity issues. For instance, while setting up the test environment, SFC Path Tracer made explicit several errors such as bogus SFs, wrong OpenFlow rule sets installation, and dataplane misconfiguration (*e.g.*, disabled ports). With the use of SFC Path Tracer, such problems were easily identified observing the SFC trace output. Figure 5.8 shows the trace output of SFC Path Tracer from a chain with two SFF and four SFs. Figure 5.8a shows the trace output when the chain is properly working. In this case, the chain has six hops indicated on the right with numbers. On the left, the first number indicates that 9 probe packets were detected in each hop. Then, the SFF is traced with the ingress and egress port from SFF. At the end of each row, the next SF is traced, such as *sf1* (1). In case the next hop is other switch (3), nothing is traced. Bellow each traced hop, its average hop delay in milliseconds is printed. Figure 5.8b shows the trace output considering same chain but with *sf3* turned off. Since the packets are dropped in *sf3*, the trace is printed until reach the problematic SF. The trace output from Figure 5.8b pinpoints the SFC path location where the packet was dropped.



(a) Output from a working chain. (b) Output from a problematic chain.

Figure 5.8: Trace Output from SFC Path Tracer.

Figure 5.9 shows a graph representation of SFC Path Tracer output. In the figure, probe packets trace a chain composed of two SFFs and six SFs. In order to illustrate SFC path failures, $SF6$ was disconnected and the link between $SFF3$ and $SF5$ was configured to drop half of the traffic.

In order to generate the graph representation, the collected traces are merged considering a known SFC path. SFC Path Tracer generates one trace output for each unique traversed path and then count the number of packets regarding this trace output. Since a link is configured to drop half of the packets, two trace outputs are generated similarly as shown in Figure 5.8. Therefore, the trace outputs are merged and packet counter are combined to generate the graph representation. From the resulting graph output, it is possible to observe that all probe packets reached the first four SFs, 50% of the traffic is lost in between $SFF3$ and $SF5$, and no traffic is observed between $SFF3$ and $SF6$. Once it is detected the point where network packets are lost, it is possible to perform a deeper investigation to quickly find the SFC path failure.

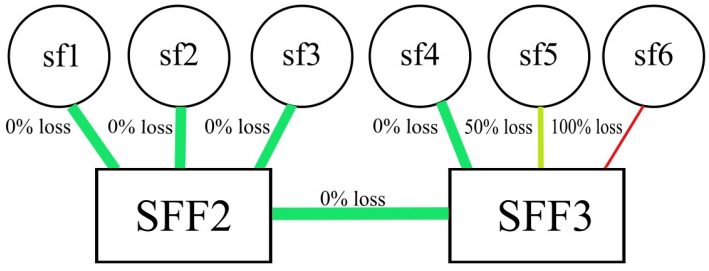


Figure 5.9: SFC Path Tracer graph output.

SFC Path Tracer accuracy regarding trace generation is verified testing the tool with fault injections. Given a specific topology, SFs were randomly turned off in order to simulate failures in the chain path. Therefore, the output trace generated by SFC Path Tracer was compared against the resulted topology, with failed SFs, and then verifying the correctness of trace output.

5.7 Experiment 3 - Chain performance

Besides being useful to troubleshoot common SFC problems such as misconfigured policies and SFs connectivity issues, SFC Trace Monitor is also able to characterize chain performance by measuring the delay introduced by the chain in each packet. Therefore, it is possible to visualize a delayed path or an overloaded service function. To enable the monitoring of a particular chain path, the whole traffic must be flagged in order to be mirrored to the controller. In this experiment, the classification rule that forwards packets to the chains is changed to add an extra action to set the ECN bits. Thus, the whole traffic forwarded to the chain is traced.

To evaluate latency measurements, a bi-directional chain is configured, *e.g.* packets are forwarded through a chain (upstream) and forwarded back through the same chain but in the opposite direction (downstream). In this experiment, using the SFC configuration framework, a chain is configured with two SFFs connecting two service functions each, as shown in Figure 5.10. The arrows in the figure represents the upstream direction of the chain. Besides dummy SFs, this experiment was executed with a real service function being part of the chain. We used *Snort*[4] in IPS (Intrusion Prevention System) mode, *i.e.*, inline through the path. In this mode, Snort analyzes the traffic based on configured rules. Additionally, a $10ms$ delay was artificially added in the link between *SFF1* and *SFF2* in order to emulate a path delay.

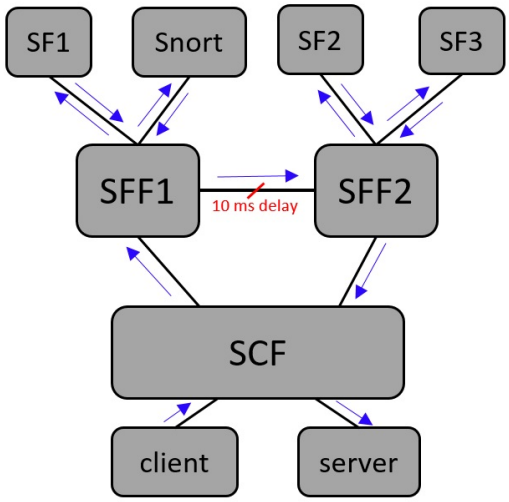


Figure 5.10: Topology used to evaluate chain performance using SFC Path Tracer.

In the chain input, a client sends HTTP requests to download a text file from a server. This traffic is tagged with the probe flag and classified to traverse the chain. Figure 5.11 shows the average latency and the standard deviation of each hop in both directions of the chain, upstream and downstream. The average is computed considering all packets from the HTTP load, which is around 4000 for upstream and 7000 for downstream.

[4] https://www.snort.org/

SFC Path Tracer correctly detected the delayed path between the SFFs. Besides, it is possible to observe the average delay of each service function. For dummy SFs, the average latency is around $4ms$ with higher deviation. However, for Snort the average latency is smaller than $1ms$. Dummy SFs are implemented in a single thread what may cause packet queuing, since they serialize the traffic. This packet queuing can increase the introduced latency by dummy SF in higher throughput. This is the reason it is observed a greater deviation for dummy SFs.

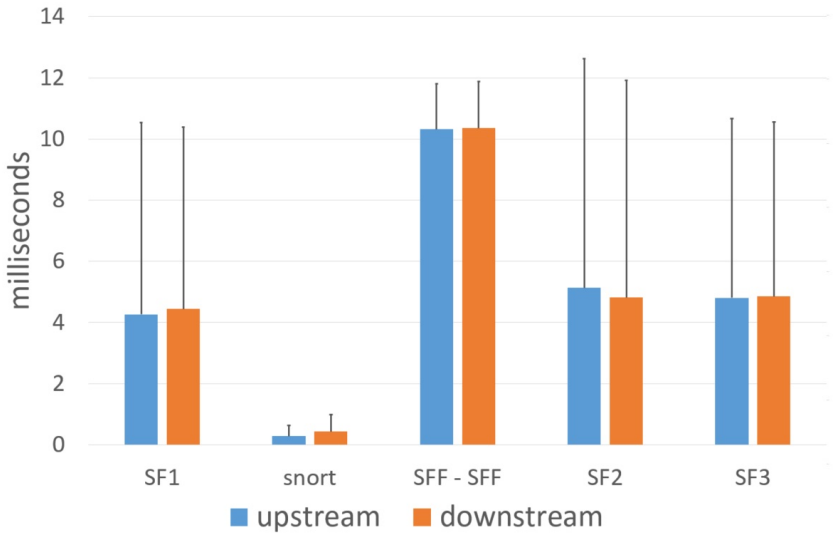


Figure 5.11: Latency measurement for each chain's hop.

SFC Path Tracer also enables the analyzing of latency considering a particular service function. For instance, it is possible to observe the latency distribution over time considering just a suspicious SF. In this experiment, we observe the latency distribution on Snort, as shown in Figure 5.12. The load of this experiment consists of a text file of $10MBytes$ being downloaded twice through the same chain of Figure 5.10 but with no delay between SFFs. Since this chain is composed by tree dummy SFs, the reached load rate is around $500kbps$.

The download is performed through a chain with an overloaded Snort. The overloading of Snort was emulated limiting the CPU resource for its process. Firstly, the experiment was executed with the Sort containing rules to analyze the TCP traffic (*with load*). Second, the same experiment was done with no rules installed in Snort, *i.e.*, the traffic just passes through Snort, with no traffic analysis (*no load*). Figure 5.12 shows that SFC Path Tracer was able to detect the load difference in Snort. With no load in Snort, the latency curve keeps flat bellow $1ms$, while with load, it is possible observe spikes up to $20ms$ of latency. Since the Snort is overload, the CPU resource is limited by each connection. Therefore, when Snort demands the CPU resource to analyze the packets we can observe the four latency spikes in the graph. Using SFC Path Tracer, an operator can troubleshoot a suspicious chain and watch a particular service function that may present intermittent problems.

SFC Path Tracer stores the timestamp of each probe packet arrived in the tool. These timestamps are used to compute the delta time between hops. Therefore, every traced hop is associated with its hop delay. Additionally, SFC Path Tracer also measures the probe packet rate (*pp/s*) over time in order to prevent saturations of OpenFlow channel. Figure 5.13 shows the rate

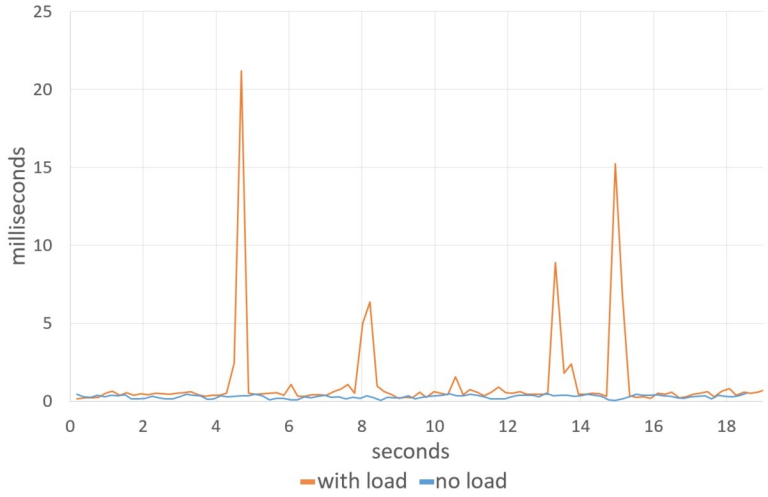


Figure 5.12: Latency distribution on Snort.

of probe packets regarding the experiment of Figure 5.12. This measurement provides information of any higher rate of packets reaching the controller, that could saturate the OpenFlow channel, which interconnects switches and the controller. The saturation of the OpenFlow channel could inject an extra delay in probe packets traveling to the controller and then providing wrong results of latency. The saturation can also cause the packet drop, what can produce wrong trace results. Figure 5.13 shows the probe packet rate in the controller for each hop of the chain. Since this date was collected simultaneously with the previous experiment, the curves show the *pp/s* considering Snort with load and with no load. For both measurements, the average of probe packets reaching the controller is around $400pp/s$ for each hop. As this chain contain 5 hops for upstream and 5 hops for downstream, the total average rate for incoming probe packets is around $4000pp/s$.

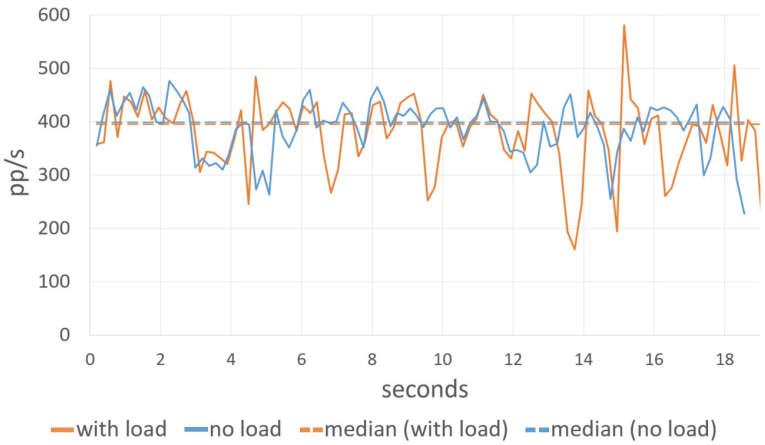


Figure 5.13: Rate of probe packets per second (pp/s) reaching the controller over time. *With load* represents the Snort analyzing packets, while *no load* means that traffic just passes through Snort with no analysis.

5.8 Experiment 4 - Sampling of probe packets

In order to support chain characterization of higher throughput traffic and also save compute resources, a target chain must have its traffic sampled to be traced. Instead of flagging the whole traffic to be mirrored to the controller, just small portions of traffic are periodically flagged. To evaluate the sampling of probe packets with SFC Path Tracer, OpenFlow rules were installed in the classifier to add the defined probe flag to packets. These sampling rules are installed with a hard timeout of one second, which is the minimum value allowed by OpenFlow. In OpenFlow protocol, when the hard timeout expires, the rule is automatically deleted. Therefore, the sampling rule is installed every 5 seconds to match a target traffic and thus achieving the probe packet sampling.

Sampling experiments were executed with *Tsung*, a tool used to test the scalability and performance of IP-based client/server applications. Tsung is configured to execute HTTP GET operations in a text file located on the server side. In this experiment, we set a limited throughput to properly compare SFC Path Tracer outputs between sampled and non-sampled traffic.

In order to achieve a load with higher throughput, the dummy SFs are not used in this experiment. Following the same topology scheme as the previous experiment, a chain is configured with just two service functions connected to a single SFF. Two Snorts are used as SFs (*e.g.*, $Snort1 \Rightarrow Snort2$). Using the same chain, we execute the test twice, (i) with all packets being mirrored to the SFC Path Tracer *i.e.*, with no sampling, and (ii) with traffic being sampled, mirroring packets every 1 second in a period of 5 seconds (*i.e.* 1 second adding the flag and 4 seconds with no flagging). Figure 5.14 shows the CPU usage on the machine where the whole experiment is executed, with the emulated network environment.

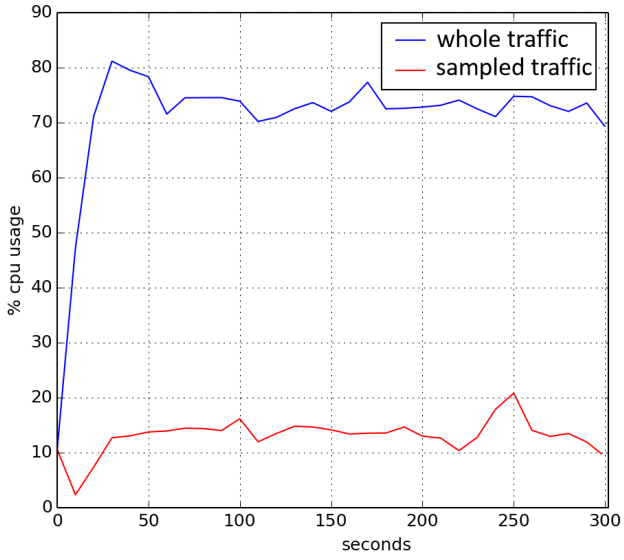
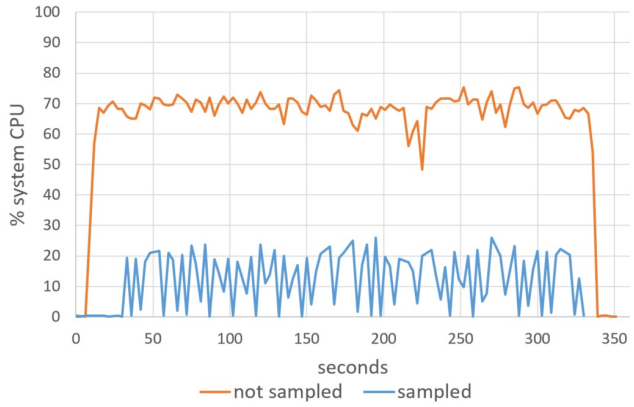


Figure 5.14: CPU usage comparison of whole traffic and sampled traffic being traced.

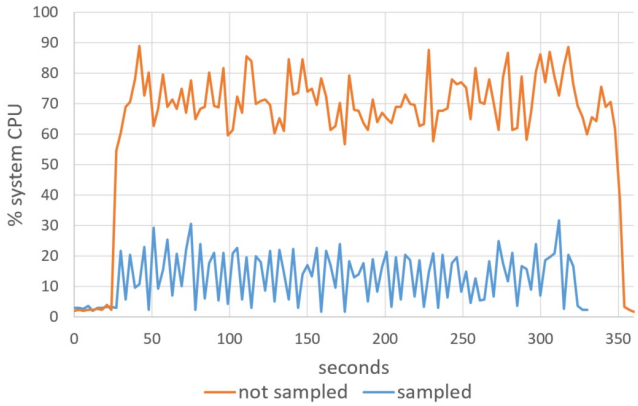
Since the whole experiment is executed on a single machine with the emulated network environment, when the whole traffic is traced it is observed a greater CPU consumption due to the

mirroring of all packets of a high throughput traffic and the parsing of this probe packets by SFC Path Tracer. Figure 5.14 shows the Tsung monitoring from CPU usage of the whole system with a chain load bandwidth limited in $4MBytes/s$. As expected, it is notable the CPU resource saving when sampling the monitored traffic comparing with the whole traffic being traced.

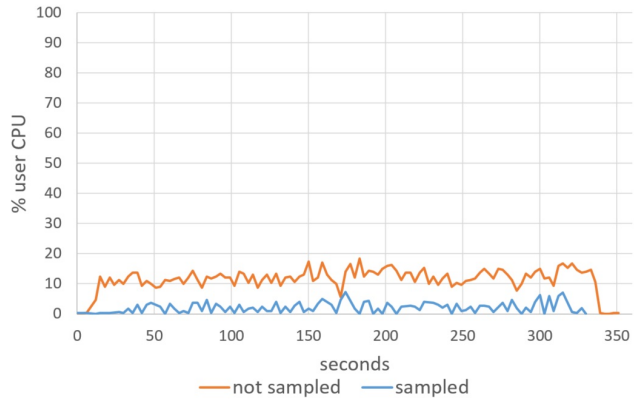
Figure 5.15 highlights CPU consumption analyzing the SFF (OVS process) and the SFC Path Tracer (ODL process). The percentage of CPU usage is regarding a system with four CPUs. Therefore, full CPU usage would be 400% of CPU consumption. Figures 5.15a and 5.15c show the CPU usage of OVS. It shows the system and user CPU consumption, where can be observed the difference between sampled and non-sampled traffic due to the extra cost of packet mirroring. Figures 5.15b and 5.15d show the CPU consumption of ODL controller. The increase of CPU consumption represents the probe packets processing by SFC Path Tracer. It is possible to notice that the system CPU of ODL have a significant increase comparing to OVS. In the ODL, SFC Path Tracer creates 4 times extra threads to process the probe packets, that is why it is possible to observe the CPU usage increase from around 15% to 75%.



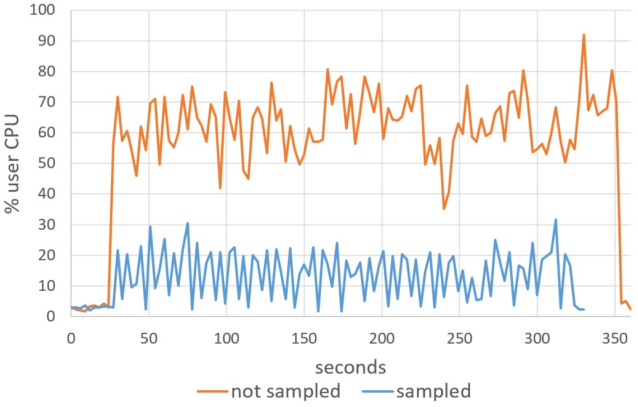
(a) OVS system CPU usage.



(b) ODL system CPU usage.



(c) OVS user CPU usage.



(d) ODL user CPU usage.

Figure 5.15: CPU resource consumption regarding switch and SFC Path Tracer.

It is possible to observe that the processing time from ODL takes longer than OVS due to probe packet queuing in the SFC Path Tracer. Therefore, the remaining time of ODL processing is regarding to the last probe packets parsing. On the other hand, the sampling process time is

shorter than non-sampling due to high CPU demand in the non-sampling case. Since the CPU resource is limited, when the probe packets are not sampled the throughput presents a higher oscillation causing the download time to take longer. Figure 5.16 shows the Tsung measurement of the load throughput between client and server through the chain that is being traced. Again, the measurement is performed for sampled and not sampled traffic mirroring. In both cases the load throughput keeps nearly the same. However, it is possible to observe a higher oscillation to achieve the defined throughput of not sampled mirroring due to high CPU demand.

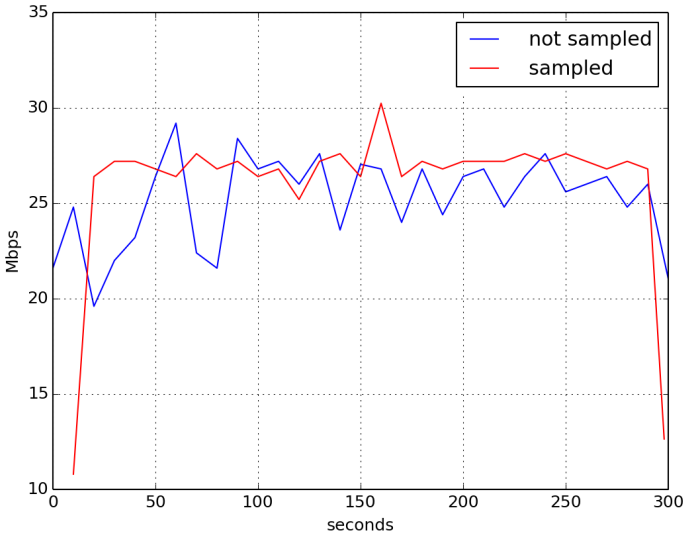


Figure 5.16: The load throughput of the chain while being monitored by SFC Path Tracer.

Figure 5.17 shows the latency distribution in one of the Snorts in the chain in order to compare the SFC Path Tracer output with sampled and non-sampled traffic. In this experiment, we artificially limited the CPU resource of the Snort process to emulate a heavy load during the traffic. This CPU limitation was added during the test to be able to see the heavy load transition. The results shows no big difference on the latency distribution considering just Snort. The sampling mechanism does not affect resulting latency output, it is possible to observe that both curves show the increasing latency between the interval of $90ms$ to $190ms$, when CPU resource was limited for Snort.

5.9 Experiment 5 - Probe packet rate limitation

The probe packet rate limitation can be observed from the pp/s curve provided by SFC Path Tracer. The data from this experiment was collected during the previous experiment from Section 5.8. Figure 5.18 shows the rate of probe packets (pp/s) being sent to the controller (packet-in) during the previous experiment with two Snorts. The graph is related to a single chain hop. The median curves of sampled traffic is bellow $500pp/s$, while the median curve of non-sampled traffic is around $1800pp/s$. Additionally, it is possible to observe that the non-sampled curve reaches around $2000pp/s$ and then goes down to almost $0pp/s$. This behavior indicates that we reached the limit

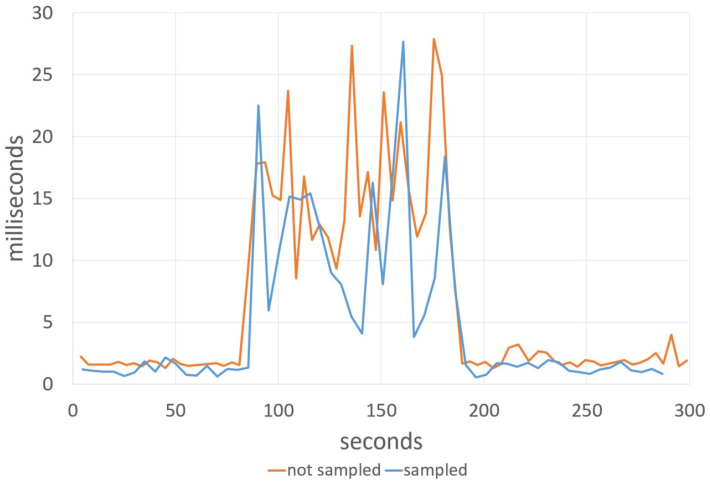


Figure 5.17: Normal and sampled monitored traffic.

rate for processing probe packets. Since the SFC Path Tracer is not able to process this amount of probe packets per second thought packet-in, some packets are lost and thus, it is possible to observe such behavior as shown in the graph of Figure 5.18.

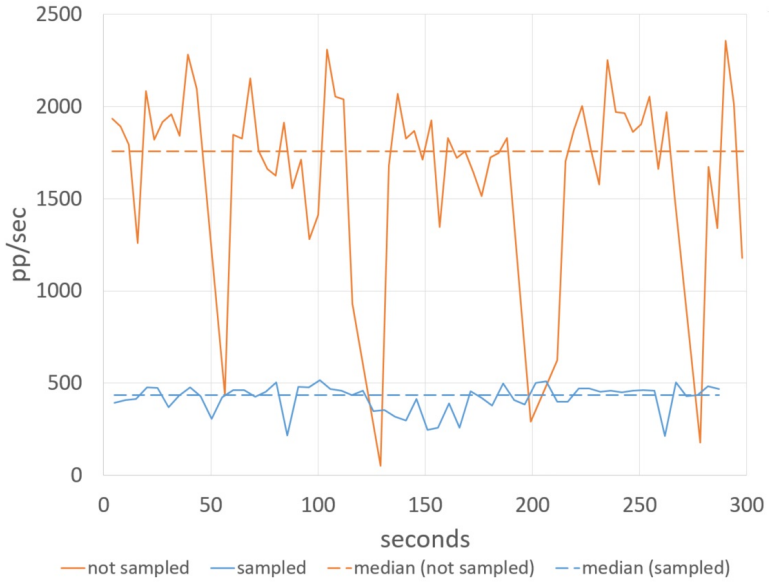


Figure 5.18: Rate of probe packets per second (pp/s) per chain hop reaching the controller over time.

SFC Path Tracer collects incoming probe packets considering a particular hop, as shown in Figure 5.18. Since the tested chain has 3 hops for upstream and also 3 hops for downstream, the total number of incoming probe packets need to be multiplied by 6. From the valley in the graph after the rate of incoming packets reach around $2000pp/s$, it is possible to infer that SFC Path Tracer is able to process around $12000pp/s$ $(2000 * 6hops)$.

In order to demonstrate the probe packet rate limitation, we configure the same chain with UDP traffic considering just upstream direction, *i.e.* 3 hops. To control the bandwidth load, *iperf* was executed on the client side (input of the chain), targeted to the server at the end of the chain. Figure 5.19 shows the measured incoming pp/s regarding a single chain hop. The tested bandwidth

is increased every 40 seconds. Again, it is possible to observe the probe packet dropping after reaching $4000pp/s$. Considering this chain has 3 hops, the same probe packet limitation is observed ($4000 * 3hops = 12000pp/s$). From observations of OF counter and number of packets arrived in the controller, it is possible to conclude that the ODL buffers that stores packet-in messages are overloaded and causing the probe packet losing.

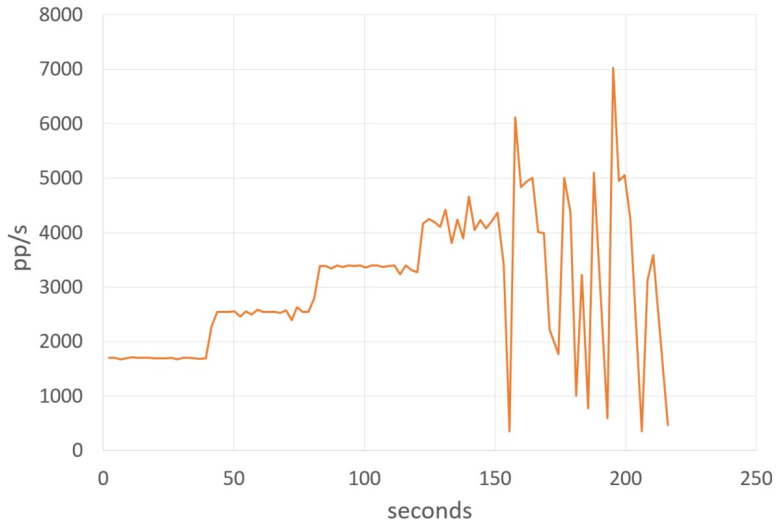


Figure 5.19: Rate limit of probe packets per second (pp/s) using tree hops.

It is important to notice that SFC Path Tracer is a proof of concept and it was not implemented to achieve the most effective performance. The limitation found is considering the current testbed, with a single machine. The limitation can be different in other deployments with ODL clustering for example. This implementation uses the OpenFlow channel to send probe packets to the controller, what have known limitations. In a real deployment that may demand a higher measurement precision with bigger sampling windows, a dedicated VxLAN tunnel can be used to forward probe packets to SFC Path Tracer.

5.10 Experiment 6 - SFC Path Tracer with other SFC encapsulation techniques

SFC Path Tracer was designed to be agnostic regarding the SFC encapsulation technique. Some SFC encapsulation may require small adaptation in the trace rules. However, the high-level design may be used for almost every known SFC encapsulation. In order to demonstrate the SFC Path Tracer compatibility, the tool was also tested with MAC Chaining and NSH.

5.10.1 MAC Chaining

Similarly to SFC VLAN, MAC Chaining also implements an SFC encapsulation for Service Functions with L2 connectivity. However, instead of encapsulating the chain ID in the TOS field

from IP header, MAC Chaining encapsulates it in the MAC addresses. MAC Chaining is implemented in OpenDayLight, as a different OpenFlow renderer and follows the defined ODL-SFC pipeline tables for OpenFlow rules. No adaptation was needed to support SFC Path Tracer.

Using the implemented SFC configuration framework, an SFC topology is created similarly to Figure 5.10, with four dummy SFs and two SFFs. SFC correctly traced probe packets, presenting the same output as SFC VLAN. In order to exemplify a comparison between the two encapsulation techniques, Figure 5.20 shows the latency distribution for both SFC VLAN and MAC Chaining techniques. The chain is exercised with UDP pings carrying the probe flags. Dummy SFs may introduce latency in the chain with significant variation among tests, as can be seen by standard deviation error bars. Therefore, the collected latency values in SFs hops are illustrative, with the intention to show the possibility of such comparisons using SFC Path Tracer. As both SFC techniques are similar, the latency values are in the same range of milliseconds per hop.

From the figure, it is possible no notice that MAC Chaining traced the classifier switch while SFC VLAN did not. Although it is possible to trace the classifier in both techniques, SFC VLAN does not include the classifier in the SFC model. In the SFC VLAN, the classifier is implemented through scripts that directly install classification rules in the OVS switches and thus, with no visibility to SFC Path Tracer to trace it.

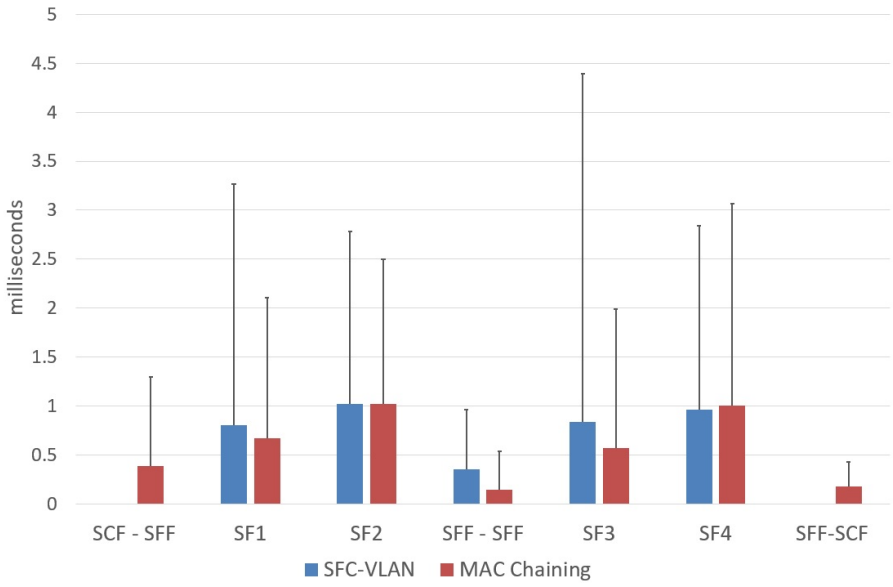


Figure 5.20: Latency comparison of SFC Path Tracer using SFC VLAN and MAC Chaining encapsulation techniques.

5.10.2 Network Service Header

Network Service Header (NSH) is implemented in OpenDayLight also as a different OpenFlow renderer. This implementation uses VxLAN-GPE tunnels to forward NSH packet among SFFs and SFs. SFC Path Tracer was slighted adapted to work with NSH chains. In the SFC VLAN and

MAC Chaining, the output port is encoded in OpenFlow metadata field to be used as input to query the SFC configuration and discover the next chain hop. For NSH-VxLAN this cannot be used since a single tunnel port may reach different destinations depending on the remote IP. Therefore, this remote tunnel IP could be encoded inside metadata field for the next hop identification. However, in this implementation, SFC Path Tracer does not use OpenFlow metatada for NSH packets. Instead, SFC Path Tracer parses NSH header from incoming probe packets, and based on *NSP* and *NSI* fields from NSH, it is possible to identify chain and hop identification and thus, query the SFC configuration for the trace output.

Unlike other experiments, this test does not use the SFC configuration framework due to the necessity to configure a topology with VxLAN tunnel connecting SFs and switches. Figure 5.21 shows the configured chain. As chain input and output, *H1* and *H2* are deployed as containers in the same machine of the classifier (*SCF*), they are connected via L2 connectivity. *SFF*, *SF1* and *SF2* are separated VMs interconnected via VxLAN-GPE tunnels. In the SFs, Python scripts are executed to parse and reply incoming NSH packets.

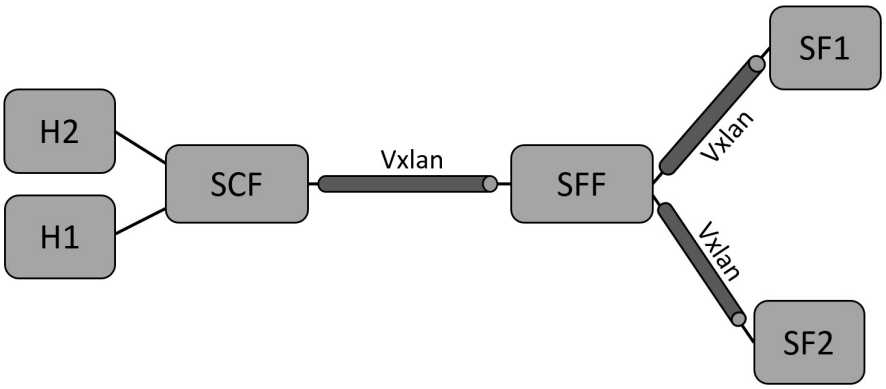


Figure 5.21: NSH topology.

In the same way as the previous experiment, UDP pings are sent through the chain and traced by SFC Path Tracer. SFC Path Tracer correctly generated a trace as: $SCF \Rightarrow SFF \Rightarrow SF1 \Rightarrow SFF \Rightarrow SF2 \Rightarrow SFF \Rightarrow SCF$. The registered latency for both SFs is $15ms$ each. The latency is greater comparing with the latency values of SFC VLAN and MAC Chaining (Figure 5.20) due to VxLAN overhead and mainly due to SFs implementation in Python to parse NSH packets. The dummy SFs used in the other experiments were implemented in C.

5.10.3 SFC encapsulation remarks

SFC Path Tracer was evaluated using SFC ODL implementation such as SFC VLAN, MAC Chaining, and NSH. However, SFC Path Tracer design can be applied to others SFC encapsulation methods. For instance, it is trivial to adapt it to Five-tuple methods, SFC techniques that use flow identifiable information to forward SFC traffic. In cases where ECN is impossible to be used

to flag probe packets, MAC address fields could encode probe packets flagging information. SFC Path Tracer can also be adapted to work with IPv6 headers, flagging probe packets information in extension headers. SFC technique that relies on traditional switch routing protocols such as 802.1q (VLAN) and MPLS may not be trivial to adapt the SFC Path Tracer as it may not use OpenFlow or any other SDN protocol.

Table 5.2 shows how probe packets flagging information could be encoded (ECN and MAC columns) considering different encapsulation techniques. Although SFC Path Tracer was just evaluated using ECN as probe flag, just chaining the trace rule installation and the filter mechanism for probe packet listener it is possible use MAC or other probe encapsulation method. OpenFlow column shows the SFC encapsulation mechanisms that may be implemented using OpenFlow protocol and thus being possible to support SFC Path Tracer, otherwise SFC Path Tracer cannot be adapted.

Table 5.2: Supported SFC encapsulation methods.

SFC methods	OpenFlow	ECN	MAC
NSH [24]	✓	✓	✓
NSH L2	✓	✗	✓
Segment Routing (VLAN) [17]	✗	✓	✓
Mac Chaining [14]	✓	✓	✗
Five-tuple	✓	✓	✓
MPLS-SPRING [58]	✗	✗	✓
Port Chain [9]	✓	✓	✗
SFC VLAN [7]	✓	✓	✗

6. CONCLUSION

This work presented the SFC Path Tracer, a troubleshooting tool for SFC environments. SFC Path Tracer proved to be useful for identifying problems in SFC paths configuration. The variety of problems make it difficult for the operator to troubleshoot a problematic chain path. Therefore, packet trace information reduces the debugging time by pinpointing the origin of a possible problem. SFC Path Tracer also provides latency information hop by hop enabling the identifications of delayed links and overloaded SFs.

SFC Path Tracer was implemented in OpenDayLight, an SDN controller largely used in the NFV/SDN area. Therefore, SFC Path Tracer was evaluated in a totally emulated environment without the requirement of any type of simulation. The implemented SFC configuration framework enables the creation of this emulated environment. With this framework, it is possible to easily create an SFC environment and automatically deploy the SFC configuration on different topologies containing SFFs and SFs.

The strategy of decrementing TTL field from IP packets ensures correct ordering of network elements in the trace. This enables the possibility to mirror probe packets to SFC Path Tracer, which does not introduce significant delays in probe packets. The usage of IP identification field to identify single probe packets among the traffic flow enables the use of concurrent probe packets, allowing the trace generation of regular traffic and not only forged pings carrying the probing flag. For regular network traffic, it is possible to use sampling techniques to add the probe flags in the packets and thus, measure chain performance and compute chain latency hop by hop.

SFC Path Tracer was designed to be a troubleshooting tool to assist network operator in the process of debugging chain path. SFC Path Tracer is not suitable to be executed constantly with regular network traffic due to the extra traffic generation in the channel that interconnects switches and the controller, where SFC Path Tracer is running. However, with lightweight probe pings, SFC Path Tracer does not introduce significant overhead in the chain path. Therefore, SFC Path Tracer can be constantly used to give precise trace information, containing all reached network elements by network packets in the chain.

SFC Path Tracer was published in the student demo track of IEEE NFV-SDN conference in 2016 at Palo Alto/USA. The SFC Path Tracer tool was also accepted as a short paper in the IEEE IM conference that will be in Portugal in May 2017.

SFC Path Tracer presented limitations to handle probe packets at high throughput. This tool was implemented as a proof of concept and it was not optimized regarding performance and concurrent thread management. To be able to process a higher rate of probe packets, instead of using the OpenFlow channel, a VxLAN tunnel could be used to send probe packets to the tool. Another aspect that could be improved is the use of alternative techniques of probing encapsulation. This could be configurable and then increase the SFC Path Tracer compatibility. SFC Path Tracer

could also be implemented using *P4* [1] protocol instead OpenFlow, what could give more flexibility for possible new features.

SFC Path Tracer was just evaluated in an emulated environment in order to easily create a variety of topologies to assure correct tracing in different scenarios. An extension of this work could evaluate this tool in a more static environment but in real deployments with virtual machines, using real network functions and even physical OpenFlow switches. This can be done deploying service chains using OpenStack integrated with OpenDayLight. This SFC integration with OpenStack is planned for the next OpenDayLight release.

SFC Path Tracer can be leveraged to add new features for future works. SFC Path Tracer just collects measurement data and does not perform any type of automatic verification. This measurement data can be used to extent SFC Path Tracer and verify SFC paths in the SFC configuration. Based on the traced output, the tool could examine the SFC configuration and automatically confirm the SFC path correctness. A future work could also use this trace output for automatic verification such as ping SFs, check switch connectivity with SFs or even rewrite SFC OpenFlow rules. Latency data also can be used to perform monitoring such as SLA verification and alert generation of a delayed path.

[1] http://p4.org/

BIBLIOGRAPHY

[1] "ContexNet: Subscriber Aware SDN Fabric enabling NFV". (Accessed on 12/05/2016), Retrieved from: https://www.hpe.com/h20195/v2/GetPDF.aspx/c04725726.pdf.

[2] "Lithium | opendaylight". (Accessed on 04/09/2016), Retrieved from: https://www.opendaylight.org/.

[3] "MD-SAL Wiki". (Accessed on 05/18/2016), Retrieved from: https://github.com/BRCDcomm/BVC/wiki/MD-SAL.

[4] "OpenDaylight Controller: MD-SAL Architecture". (Accessed on 01/28/2017), Retrieved from: https://wiki.opendaylight.org/view/OpenDaylight_Controller:MD-SAL:Architecture.

[5] "Service Function Chaining Home - Service Function Chaining - OPNFV Wiki". (Accessed on 11/22/2016), Retrieved from: https://wiki.opnfv.org/display/sfc/Service+Function+Chaining+Home.

[6] "Service Function Chaining — OpenDaylight Documentation Boron documentation". (Accessed on 12/21/2016), Retrieved from: http://docs.opendaylight.org/en/stable-boron/user-guide/service-function-chaining.html.

[7] "Service Function Chaining:Lithium User Facing Features - OpenDaylight Project". (Accessed on 05/18/2016), Retrieved from: https://wiki.opendaylight.org/view/Service_Function_Chaining:Lithium_User_Facing_Features#SFCOFL2.

[8] "Service Function Chaining:Main - OpenDaylight Project". (Accessed on 11/22/2016), Retrieved from: https://wiki.opendaylight.org/view/Service_Function_Chaining:Main.

[9] "Service Port Chain Workflow documentation". (Accessed on 09/03/2016), Retrieved from: http://docs.openstack.org/developer/networking-sfc/port_chain_system_and_flow.html.

[10] Afek, Y.; Bremler-Barr, A.; Landau Feibish, S.; Schiff, L. "Sampling and large flow detection in SDN". In: ACM SIGCOMM Computer Communication Review, 2015, pp. 345–346.

[11] Agarwal, K.; Rozner, E.; Dixon, C.; Carter, J. "SDN traceroute: Tracing SDN forwarding without changing network behavior". In: 3rd workshop on Hot topics in software defined networking, 2014, pp. 145–150.

[12] Aldrin, S.; Pignataro, C.; Krishnan, R. R.; Ghanwani, A.; Akiya, N. "Service Function Chaining Operation, Administration and Maintenance Framework", Internet-Draft draft-aldrin-sfc-oam-framework-02, Internet Engineering Task Force, 2015, work in Progress.

[13] Bjorklund, M. "YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)", RFC 6020, RFC Editor, 2010.

[14] Bottorff, P.; Fedyk, D.; Assarpour, H. "Ethernet MAC Chaining", Internet-Draft draft-fedyk-sfc-mac-chain-01, Internet Engineering Task Force, 2016, work in Progress.

[15] Carpenter, B.; Brim, S. "Middleboxes: Taxonomy and Issues", RFC 3234, RFC Editor, 2002.

[16] Detal, G.; Hesmans, B.; Bonaventure, O.; Vanaubel, Y.; Donnet, B. "Revealing middlebox interference with tracebox". In: Internet Measurement Conference, 2013, pp. 1–8.

[17] Dolson, D. "VLAN Service Function Chaining", *IETF (Internet Engineering Task Force) Internet-Draft, draft-dolson-sfc-vlan-00*, 2014.

[18] European Telecommunications Standards Institute (ETSI). "Network Functions Virtualization: Introductory White Paper". (Accessed on 06/10/2016), Retrieved from: https://portal.etsi.org/nfv/nfv_white_paper.pdf, 2012.

[19] European Telecommunications Standards Institute (ETSI). "Network Functions Virtualisation (NFV); Architectural Framework". (Accessed on 07/20/2016), Retrieved from: http://www.etsi.org/deliver/etsi_gs/nfv/001_099/002/01.01.01_60/gs_nfv002v010101p.pdf, 2013.

[20] Farrel, A. "Service Function Chaining". (Accessed on 07/21/2016), Retrieved from: http://datatracker.ietf.org/doc/charter-ietf-sfc/.

[21] Fayaz, S. K.; Yu, T.; Tobioka, Y.; Chaki, S.; Sekar, V. "BUZZ: testing context-dependent policies in stateful networks". In: 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16), 2016, pp. 275–289.

[22] Fundation, O. N. "Software-defined networking: The new norm for networks". (Accessed on 03/10/2016), Retrieved from: https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf, 2012.

[23] Garcia, B. R. "OpenDaylight SDN controller platform", Master's Thesis, Universitat Politècnica de Catalunya, 2015.

[24] Garg, P.; Quinn, P.; Manur, R.; Guichard, J.; Kumar, S.; Chauhan, A.; McConnell, B.; Smith, M.; Wright, C.; Elzur, U.; Halpern, J. M.; Henderickx, W.; Nadeau, T.; Majee, S.; Melman, D. T.; Glavin, K.; Agarwal, P. "Network Service Header", Internet-Draft draft-quinn-sfc-nsh-07, Internet Engineering Task Force, 2015, work in Progress.

[25] Greenhalgh, A.; Huici, F.; Hoerdt, M.; Papadimitriou, P.; Handley, M.; Mathy, L. "Flow processing and the rise of commodity network hardware", *ACM SIGCOMM Computer Communication Review*, vol. 39–2, 2009, pp. 20–26.

[26] Halpern, J.; Pignataro, C. "Service Function Chaining (SFC) Architecture", RFC 7665, RFC Editor, 2015.

[27] Handigol, N.; Heller, B.; Jeyakumar, V.; Maziéres, D.; McKeown, N. "Where is the debugger for my software-defined network?" In: 1st workshop on Hot topics in software defined networks, 2012, pp. 55–60.

[28] Handigol, N.; Heller, B.; Jeyakumar, V.; Mazières, D.; McKeown, N. "I know what your packet did last hop: Using packet histories to troubleshoot networks". In: 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14), 2014, pp. 71–85.

[29] Hemid, A. "Facilitation of the OpenDaylight Architecture". (Accessed on 03/18/2016), Retrieved from: http://mc-lab.inf.h-brs.de/projects/sdn/OpenDayLight.pdf, 2015.

[30] Jacquenet, C.; Boucadair, M. "An IPv6 Extension Header for Service Function Chaining", Internet-draft, IETF, 2015.

[31] Jammal, M.; Singh, T.; Shami, A.; Asal, R.; Li, Y. "Software defined networking: State of the art and research challenges", *Computer Networks*, vol. 72, 2014, pp. 74–98.

[32] Katz, D.; Ward, D. "Bidirectional Forwarding Detection (BFD)", RFC 5880, 2010.

[33] Kazemian, P.; Varghese, G.; McKeown, N. "Header space analysis: Static checking for networks". In: 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12), 2012, pp. 113–126.

[34] Kumar, S.; Tufail, M.; Majee, S.; Captari, C.; Homma, S. "Service Function Chaining Use Cases In Data Centers", Internet-Draft draft-ietf-sfc-dc-use-cases-04, Internet Engineering Task Force, 2016, work in Progress.

[35] Lantz, B.; Heller, B.; McKeown, N. "A network in a laptop: rapid prototyping for software-defined networks". In: ACM SIGCOMM Workshop on Hot Topics in Networks, 2010, pp. 1–6.

[36] Leung, K.; Wang, E.; Felix, J.; Iyer, J. "Service Function Chaining Use Cases for Network Security", Internet-Draft draft-wang-sfc-ns-use-cases-01, Internet Engineering Task Force, 2016, work in Progress.

[37] Lopez, D.; Homma, S.; Dolson, D.; Boucadair, M.; Liu, D.; Ao, T.; Vu, V. A. "Hierarchical Service Function Chaining (hSFC)", Internet-Draft draft-dolson-sfc-hierarchical-06, Internet Engineering Task Force, 2016, work in Progress.

[38] Ltd., C. "LXC Linux containers". (Accessed on 7/22/2016), Retrieved from: https://linuxcontainers.org/.

[39] McKeown, N.; Anderson, T.; Balakrishnan, H. "OpenFlow: Enabling Innovation in Campus Networks", *ACM SIGCOMM Computer Communication Review*, vol. 38–2, 2008, pp. 69–74.

[40] Napper, J.; Haeffner, W.; Stiemerling, M.; Lopez, D. R.; Uttaro, J. "Service Function Chaining Use Cases in Mobile Networks", Internet-Draft draft-ietf-sfc-use-case-mobility-06, Internet Engineering Task Force, 2016, work in Progress.

[41] ONF. "OpenFlow-enabled SDN and Network Functions Virtualization". (Accessed on 04/28/2016), Retrieved from: https://www.opennetworking.org/images/stories/downloads/sdn-resources/solution-briefs/sb-sdn-nvf-solution.pdf, 2014.

[42] OpenDayLight. "OpenDaylight Platform". (Accessed on 05/10/2016), Retrieved from: https://www.opendaylight.org/.

[43] OpenVswitch. "Open vSwitch". (Accessed on 7/19/2016), Retrieved from: http://openvswitch.org/.

[44] Paxson, V.; Almes, G.; Mahdavi, J.; Mathis, M. "Framework for IP Performance Metrics", RFC 2330, RFC Editor, 1998.

[45] Penno, R.; Quinn, P.; Pignataro, C.; Zhou, D. "Services Function Chaining Traceroute", Internet-Draft draft-penno-sfc-trace-03, Internet Engineering Task Force, 2016, work in Progress.

[46] Pfaff, B.; Davie, B. "The Open vSwitch Database Management Protocol", RFC 7047, 2013.

[47] Pfaff, B.; Pettit, J.; Koponen, T.; Jackson, E.; Zhou, A.; Rajahalme, J.; Gross, J.; Wang, A.; Stringer, J.; Shelar, P.; et al.. "The design and implementation of open vswitch". In: 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15), 2015, pp. 117–130.

[48] Phaal, P.; Panchen, S.; McKee, N. "InMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks", RFC 3176, RFC Editor, 2001.

[49] Quinn, P.; Guichard, J. "Service Function Chaining: creating a service plane via network service headers", *Computer*, vol. 11–47, 2014, pp. 38–44.

[50] Quinn, P.; Kreeger, L.; Lewis, D.; Yong, L.; Xu, X.; Elzur, U.; Smith, M.; Garg, P.; Manur, R.; Maino, F.; Agarwal, P.; Melman, D. T. "Generic Protocol Extension for VXLAN", Internet-draft, Internet Engineering Task Force, 2015.

[51] Quinn, P.; Nadeau, T. "Problem Statement for Service Function Chaining", RFC 7498, RFC Editor, 2015.

[52] Rosa, R.; Siqueira, M.; Rothenberg, C. E.; Barea, E.; Marcondes, C. "Network function virtualization: Perspectivas, realidades e desafios", *SBRC-Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, 2014.

[53] Sekar, V.; Ratnasamy, S.; Reiter, M. K.; Egi, N.; Shi, G. "The Middlebox Manifesto: Enabling Innovation in Middlebox Deployment". In: 10th ACM Workshop on Hot Topics in Networks, 2011, pp. 21.

[54] Sherry, J.; Ratnasamy, S.; At, J. S. "A survey of enterprise middlebox deployments". (Accessed on 03/12/2016), Retrieved from: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.379.3064, 2012.

[55] Stoenescu, R.; Popovici, M.; Negreanu, L.; Raiciu, C. "SymNet: scalable symbolic execution for modern networks". In: ACM SIGCOMM Computer Communication Review, 2016, pp. 314–327.

[56] Tschaen, B.; Zhang, Y.; Benson, T.; Benerjee, S.; Lee, J.; Kang, J.-M. "SFC-Checker: Checking the Correct Forwarding Behavior of Service Function Chaining". In: Network Function Virtualization and Software Defined Networks (NFV-SDN), IEEE Conference, 2016, pp. 134–140.

[57] Wells, A. T.; Krol, E.; Plzak, R. "FYI on Questions and Answers Answers to Commonly AskedNew Internet User Questions", RFC 1594, RFC Editor, 1999.

[58] Xu, X.; Li, Z.; et al. "Service Function Chaining Using MPLS-SPRING", Internet-draft, IETF, 2015.

[59] Yang, X.; Zhu, L.; Karagiannis, G. "SFC Trace Issue Analysis and Solutions", Internet-Draft draft-yang-sfc-trace-issue-analysis-01, Internet Engineering Task Force, 2016, work in Progress.

[60] Zhou, D.; Penno, R.; Quinn, P.; Li, J. "Yang Data Model for Service Function Chaining", Internet-Draft draft-penno-sfc-yang-14, Internet Engineering Task Force, 2016, work in Progress.

Pontifícia Universidade Católica do Rio Grande do Sul
Pró-Reitoria Acadêmica
Av. Ipiranga, 6681 - Prédio 1 - 3º. andar
Porto Alegre - RS - Brasil
Fone: (51) 3320-3500 - Fax: (51) 3339-1564
E-mail: proacad@pucrs.br
Site: www.pucrs.br/proacad