ONE PASS PACKET STEERING (OPPS) FOR MULTI-SUBSCRIBER SOFTWARE DEFINED NETWORKING ENVIRONMENTS

By

Julian Chukwu

July 2017

A Thesis

submitted to the Faculty of Graduate and Postdoctoral Studies

in partial fulfillment of the requirements for the degree of

Master of Applied Science in Electrical and Computer Engineering

(The M.A.Sc. Program is a joint program between University of Ottawa and Carleton University, administered by the Ottawa-Carleton Institute for Electrical and Computer Engineering)

Abstract

In this thesis, we address the problem of service function chaining in a network. Currently, problems of chaining services in a network (i.e. service function chaining) can be broadly categorised into middlebox placement in a network and packet steering through middleboxes. In this work, we present a packet steering approach - One Pass Packet Steering (OPPS) - for use in multi-subscriber environments, with the aim that subscribers having similar policy chain composition should experience the same network performance. We develop and show algorithms with a proof of concept implementation using emulations performed with Mininet. We identify challenges and examine how OPPS could benefit from the Software Defined Data Center architecture to overcome these challenges. Our results show that, given a fixed topology and different sets of policy chains containing the same middleboxes, the end-to-end delay and throughput performance of subscribers using similar policy chains remains approximately the same. Also, we show how OPPS can use a smaller number of middleboxes and yet, achieve the same hop count as that of a reference model described in a previous work as ideal, without violating the subscribers' policy chains.

Acknowledgements

My sincere appreciation to Professor Ashraf Matrawy (Supervisor) for his encouragements, dedication and for always being available to direct my research activity. My sincere appreciation goes also to Professor Dimitrios Makrakis (Co-supervisor) for his time, insightful comments, and for granting me the opportunity to start and complete my research program under his tutelage. A warm appreciation to the members of the NGN Research group at Carleton University for the good rapport and making my degree program an interesting one.

A special thanks to Adam Noel (Ph.D.) for his encouragements and his willingness in accepting the burden to review this thesis. A special thanks to Amarachi Ojimba for always being there to give me moral support.

A special thanks and appreciation to my parents for funding my Masters degree program, for their support, advices and encouragements all through my program. My gratitude to God for making my Master degree program a reality.

Dedication

To my Parents.

Contents

List of Figures

List of Tables

List of Abbreviations

API	Application Programming Interface
ARP	Address Resolution Protocol
COATS	Competitive Online Algorithm for Traffic Steering
CPU	Central Processing Unit
DLog	Deep Packet Inspector with Logging Capability
Dlog	Deep Packet Inspector with Logging Capability
DPI	Deep Packet Inspector
FPTAS	Fully Polynonmial Time Approximation Scheme
FW	Firewall
HTTP	Hypertext Transfer Protocol
IDS	Intrusion Detection System
ILP	Integer Linear Programming
IP	Internet Protocol
LB	Load balancer
LP	Linear Programming
MILP	Mixed Integer Linear Programming
MP	Maximum Parismony

MPLS	Multi-Protocol Layer Switching
MTU	Maximum Transmission Unit
NAT	Network Address Translator
NFV	Network Function Virtualization
NIC	Network Interface Card
NP-Hard	Non-deterministic Polynomial-time Hard
OEM	Original Equipment Manufacturer
OPPS	One Pass Packet Steering
RFC	Request for Comments
SDDC	Software Defined Data Center
SDN	Software Defined Networking
SLA	Service Level Agreement
TCP	Transmission Control Protocol
TL	Traffic Logger
ToS	Type of Service
VLAN	Virtual Local Area Network
VM	Virtual Machine
VNF	Virtual Network Function

Chapter 1

Introduction

As network requirements evolve, the need to add extra service functions (middleboxes) becomes a necessity. Individual middleboxes, each serving a single function out of a set of required functions, are deployed on networks in an ordered way to cater for the growing network and service requirements. The ordered deployment of these middleboxes constitutes the basic concept known as *Service Chaining*.

In formal terms, RFC 7498 [4] defines a Service Function Chain as an "ordered set of service functions with ordered constraints that must be applied to packets". This definition merges the policy chain (ordered constraints or intents applied to packet) with the sequence of service functions deployed on the topology to form the service chain. While this is correct, for simplicity, the sequence or order of service functions deployed on the topology alone is referred to as the *Service Chain*. The term *Policy chain* refers to a high level composition of an ordered set of the individual network functions that make up a Service Chain [5]. Studies [6, 7, 8, 9, 5] have shown that such a high level description of the network is a preferred way for network administrators to express policies. For example, assume that a firewall (FW), a NAT

and a load balancer (LB) are deployed on the network, and the network administrator specifies that HTTP packets should traverse these middleboxes in the order - from NAT to Firewall to Load balancer. Then a policy chain specification for the above policy can be written as $HTTP$: $NAT \rightarrow FW \rightarrow LB$. The idea is that policy chains can be formulated irrespective of the positioning of middleboxes in the underlying topology. This enables decoupling of the physical topology from the way packets are routed or steered through middleboxes. It further allows the network administrator or subscribers to specify policies without considering the nuances of the undelying middleboxes deployed on the topology and enables multiple policy chains owned by different subscribers to co-exist on the same network and share the same network links.

Hence, a policy chain may or may not contain all middleboxes or service functions that make up the service chain. Also, a policy chain may or may not map to the order of service functions on the topology and two or more policy chains may exist simultaneously on the same service chain. In the case that one or more policy chains exist on a network, the policy chain then constitutes a component of the Service Level Agreement (SLA) for a particular subscriber [10, 1, 8, 7, 9].

The introduction of Software Defined Networking (SDN) has generated significant research activities in addressing challenges associated with communication networks. One such challenge is how to choose an optimal way to steer a packet through a sequence of service functions as specified by a network administrator or a subscriber. In choosing a steering scheme to route packets according to a policy chain, it is required that the correctness of the policy is maintained [5, 1]. This has often been

achieved by developing mechanisms to modify the data plane and steer packets towards middleboxes strictly as specified by the policy chain [5, 9, 7]. Although steering packets strictly according to policy chains can ensure correctness, this does not ensure that different subscribers with different policies, sharing the same physical topology and service functions, have fairly the same latency distribution in terms of packet end-to-end delay.

1.1 Motivation

Rahul et al [11] stated that 43% of high severity incidents in networks are caused by middleboxes despite constituting just 11% of the population. Their research further states that out of the five most prominent impacts on network services caused by middleboxes, the issue of latency ranks second and problems arising from violation of service level agreements rank fifth. Other network service impacts by middleboxes include lost connectivity, service error and security issues, which rank first, third and fourth respectively.

StEERING[7] presented a scenario where there can be more than one subscriber sharing the same service function and the order with which each subscriber's packet traverses the service functions differ. This situation arises when each subscriber wants a different ordered set of middlebox treatment to be applied to his packet. Ocassionally, the network elements and middleboxes are deployed in fixed locations as in the case with traditional networks [5]. In a fixed network environment, a solution to solving the packet steering problem would be to formulate low level flow rules that will steer each subscriber's flow according to their respective intent as demonstrated in StEERING and SIMPLE [9].

This steering policy enables fulfilling the subscriber's policy chain. However, on close examination of this steering mechanism for the multi-subscriber environment, it is seen that the second ranking issue (latency) remains a challenge for this steering approach.

1.2 Problem Statement and Goal

An example of a multi-subscriber environment is a data center. A characteristic requirement of this environment is that it involves at least two parties: The *Service provider* and the *Subscriber*. A *subscriber* refers to a content provider that leases network infrastructure to provide contents for consumers. Examples of subscribers include Airbnb, Unilever and Netflix. They use Amazon's infrastructure to provide contents to consumers. By the term *service provider*, we refer to a company that provides infrastructure for subscribers to host their contents. Example of service provider in this context include Amazon and RackSpace.

The Service provider is usually constrained by physical resources while the subscriber has the freedom to choose the order by which packets should traverse the service functions before getting to the deployed server.

In this section we give a scenario to illustrate a potential problem in the multi-subscriber environment.

With reference to Fig. 1, we consider a data center hosting servers for two different subscribers; X and Y. The data center has a firewall (F) and a traffic logger (T) deployed on the topology as shown in Fig. 1. For Y, it is desired that packets to server Y travel through the traffic logger before reaching the firewall (red arrow in Fig. 1) and for X, it is desired that packets to X travel through the firewall, then

through the traffic logger (blue arrow in Fig. 1). In the service level agreement of X, we can specify the policy of X as *X:* $F \rightarrow T$ and in the service level agreement of Y, we can specify the policy of Y as *Y:* $T \rightarrow F$. A simple example of applicability is that X is interested in logging legitimate traffic, while Y is interested in logging both, legitimate and illegitimate traffic.

We acknowledge that the topology considered in the presented example is small compared to middlebox deployments we find in a large network. However, this example can be easily extended to cover cases of large stateless policy chain deployments. While steering packets to achieve the service level agreement (SLA) of the two subscribers can be done using only data plane configurations [9, 2, 7], we show that with tags, packets from both subscribers can traverse the network in one pass and still meet the SLA.

Visually described, our goal is to develop a technique that can be applied in transforming the packet steering approach shown in Fig. 1 to the packet steering approach displayed in Fig. 2 (note the red arrows in both figures) without violating the Service Level Agreement (SLA). For this, we develop a packet steering technique called *One Pass Packet Steering (OPPS).*

OPPS is a steering technique capable of satisfying the service level agreement of different subscribers in shared and constrained (in terms of the number of service functions) multi-subscriber SDN environments by steering packets through a sequence of stateless middleboxes in one direction.

In order to implement the OPPS technique, we introduce new header fields to track middlebox processing contexts, and a lightweight OPPS Module that extends

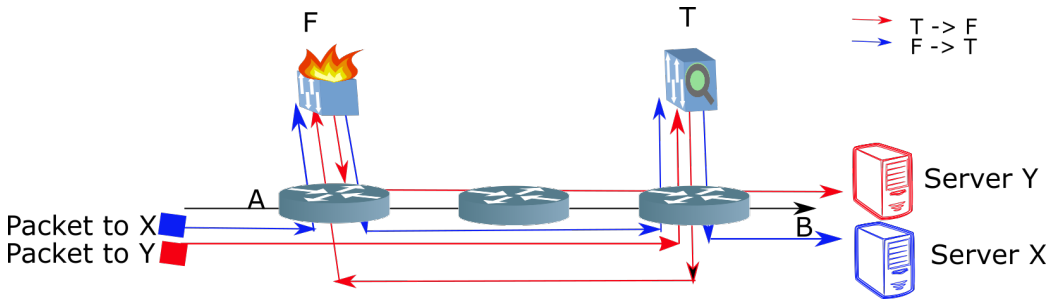

Figure 1: Multi-subscriber environment showing paths traversed by packets to server X and server Y.

middleboxes and performs operations using data stored on the newly added header fields.

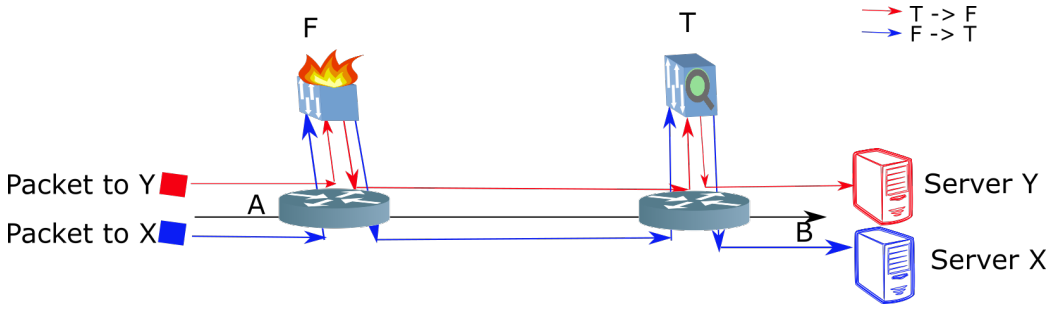

Figure 2: Paths traversed by packets using the One Pass Packet Steering approach.

1.3 Thesis Contribution

OPPS aims to ensure that subscribers sharing a service chain (i.e., the same sequence of middleboxes on a physical topology), or having different policy chains (i.e., different intents on how middleboxes are ordered on the service chain) but the same service functions (i.e., similar policy chain composition) in their policy chain, achieve similar latency distributions.

Thus, our main contributions are:

- The introduction[1] of One Pass Packet Steering (OPPS) for groups of policy chains containing similar stateless service functions. For this we develop a special service sub-layer dedicated to service function chaining and an architecture to support OPPS.
- We also introduce and explain new algorithms that act on the service sub-layer to determine when certain actions should be applied on packets.
- We present results for adopting OPPS in Software Defined Data Center.

1.4 Research Publications

During the course of this Masters degree program, this publication has been produced:

- J. Chukwu, A. Matrawy, and D. Makrakis, "One pass packet steering (OPPS) for stateless policy chains in multi-subscriber SDN", in proceedings of 2017 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS): SWFAN 17: International Workshop on Software-Driven Flexible and Agile Networking (INFOCOM17 WKSHPS SWFAN'17), Atlanta, USA, May 2017.

This paper has been submitted for review:

- J. Chukwu, A. Matrawy, and D. Makrakis, "Consolidating Policy Chains Using One Pass Packet Steering in Software Defined Data centers", submitted to IEEE Transactions on Cloud Computing for review.

[1] Initial proposal and preliminary results of OPPS were presented in [12].

1.5 Thesis Outline

The remainder of this thesis is organized as follows:

Chapter 2 discusses background knowledge needed to assimilate the contents of this thesis. The background knowledge includes explanation of terms and concepts used through the remainder of this thesis.

Chapter 3 discusses related past works. The discussion in chapter 3 comprises analysis, deductions and summaries of similar works done prior to this thesis.

Chapter 4 presents the One Pass Packet Steering method. We discuss in detail the algorithms and modules that make up OPPS.

Chapter 5 discusses the setup of our environment and gives initial evaluation and challenges in using OPPS.

Chapter 6 introduces use of OPPS in the SDDC environment and focuses on describing ways to overcome the challenges discussed in Chapter 4. It also examines the benefits and trade-offs coming with the use of OPPS.

Chapter 7 concludes this thesis and highlights future work.

Chapter 2

Background

2.1 Middleboxes

Middleboxes are becoming an integral part of small, medium and large scale enterprise networks [13] with posibility of scaling up to 2000+ middleboxes per network [11]. According to RFC 3234 [14], A middlebox can be defined as an *"intermediary box performing functions apart from normal, standard functions of an IP router on the data path between a source host and destination host."* A distinguishing factor that is unique of a network function categorized as middlebox is that middleboxes, unlike routers, which can also be categorized as intermediary boxes, do not perform standard network routing functions. A single middlebox can be composed of several functions, therefore a more granular approach to middlebox classification can be adopted [15]. Edeline et. al. [15] identified middleboxes based on policies and middleboxes were classified based on the purpose for which they were created. Classification was further done based on actions resulting from middleboxes and complications arising from these actions. Also, extensive middlebox classification has been carried out

in "Middlebox: Taxonomy and Issues" [14]. However, with all these classifications, the task of designing a simplified general model that describes and captures all kinds of middleboxes has been a challenge due to the absence of a blueprint for middlebox modelling. In "Modelling middleboxes" [16], four middleboxes were analysed and a general model for middleboxes consisting of *zones, input pre-conditions, state databases, processing rules, auxiliary traffic, and state fields* was given. Following the work reported in [16, 15, 14], it can be seen that middleboxes generally follow a high level pattern as in Fig. 3. A packet is usually given as the input, state variables are applied to the packet if applicable, a processing action is carried out, and a result or verdict is given. The results from packet processing can be classified in to forwarded modified packets, dropped packets, and forwarded unmodified packets [15]. To further illustrate this classification concept and this high level pattern that middleboxes follow, let us consider how three popular middleboxes (a network address translator, a load balancer and a firewall) work and how they fit in this general structure. While this is a brief description of their operation, these middleboxes have been extensively discussed in [14, 16] and also in RFC 2979 for the firewall and RFC 3022, RFC 3027 and RFC 2993 for NATs.

2.1.1 Network Address Translator (NAT)

The NAT translates the network bound (external) IP address and port to the inbound IP address and port. The reverse happens when a packet leaves the network. NATs also maintain state information on flows, which is used for mapping inbound packets to corresponding outbound packets and vice versa. By state information, we mean that some sort of signature is maintained per flow or session by the middlebox (NAT)

or a state database. We may also think of "State" as a key - value pair used in identifying packets after they are re-written; for example in the case of NATs. In our high level pattern, we can say that the NAT takes a packet input, applies some state variables, and produces an output which can be classified as a re-written packet.

2.1.2 Load Balancer

Similar to NAT, Load balancers take packets as input, generate states for new packets or use state variable/information from previous packets to identify subsequent packets in a particular flow session, then route the flows to a specific endpoint. Usually Load balancers also re-rewrite the destination address and ports placed in headers to acheive efficient routing. This concept also matches the model shown in Fig. 3.

2.1.3 Firewall

Firewalls can be stateful or stateless. Stateful firewalls use state information to re-assemble fragmented packets before processing whereas stateless firewalls process each fragment individually without reassembling. In general, the firewall takes in a packet and processes the packet by applying rules to determine whether the packet should be dropped or not. If the packet is not dropped, it is forwarded without modification.

Succinctly, with the categorisation of middleboxes in [9, 15] based on actions, we agree that the general actions can be broadly grouped under the following - *Modified/Forward, Drop, Unmodified/Forward*. Hence, we adopt this classification to form part of the basis for our work.

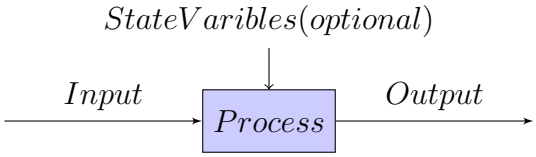

Figure 3: General Structure of a Middlebox.

2.2 Stateful and Stateless Policy Chains

This section draws inference from middlebox operations described in [14, 17, 15, 16] and gives a general high level description of the word *"State"* (a Stateful Policy Chain and a Stateless Policy Chain). "*To keep state*" or being "*stateful*" is the property of a system to store data that can be used at a later time for future reference.

2.2.1 Stateful Policy Chain

To define a stateful policy chain, we say a policy chain is stateful if certain information about a packet P at time t_{m} (were $t_{\mathrm{m}} < t_{\mathrm{n}}$) is stored and such information is considered as a factor in processing the packet at time t_{n}. In this context, this also means that a policy chain is stateful if its decision or action on a packet at a time t_{n} can be affected by the decision or action made regarding this packet at any earlier time $\mathrm{t_m}$. Example; Having a NAT before a firewall introduces a stateful policy chain because a firewall that could filter on the IP address of a packet could have the IP address changed by a NAT.

Hence, a policy chain is referred to as stateful if it contains a stateful service function whose action or decision on a packet affects the action or decision made by any other service function further down the policy chain. A stateful policy chain should not be confused with a stateful service function. A stateful service function is

a middlebox that stores or holds data, and uses this saved data for decision making at a later time, whereas, a stateful policy chain requires at least two middleboxes chained together and at least one of the middleboxes is stateful.

2.2.2 Stateless Policy Chain

In a stateless policy chain, the middleboxes that make up the policy chains are stateless. A middlebox is stateless if it does not store information about a packet P. Examples of stateless middleboxes include Firewall, Deep Packet Inspectors, Load balancers, Traffic Loggers, Network Analyzers and Statistical network functions that only poll information from packets for analysis. We consider load balancer as stateless if its action on the service chain does not alter the decision a packet makes down the chain. We refer to a stateless load balancer as a service function that balances the network load between two identical service functions or endpoints that performs the same function and can otherwise be represented as a single service function in the policy chain.

A policy chain is referred to as stateless if, for any service function on the policy chain, the result obtained on any such service function does not affect the decision made about the same packet by any other service function down the chain. Similarly, the decision made by such service function does not depend on results obtained from previous middleboxes.

Therefore, a service chain is stateless if the outcome of the decision made for a packet at time t_n is independent of any other decision made regarding this packet at any earlier time t_m along this service chain. Example; Having a Deep Packet Inspector (DPI) and a traffic logger (TL), a DPI will always drop a malicious packet that goes

through it while a traffic logger will always log a traffic that passes through it.

2.3 Middlebox Deployment Models

This section introduces the Choke point model and the Way point model and discusses the essential difference that makes the migration to the Way point model promising.

2.3.1 The Choke Point Model

The internet protocol architecture was designed without consideration for middlebox support. Due to the lack of plan for middlebox support in the initial designs of the internet architecture, middleboxes were deployed on network paths and where fused to the network in a static and rigid manner, just before the network nodes or endpoints which they support. This deployment strategy, known as the Choke Point model [1, 18], was applied in order to ensure that packets traversed middleboxes before reaching the service endpoint.

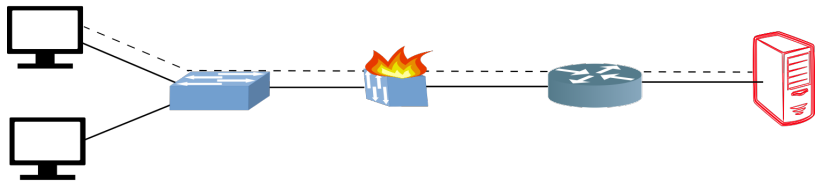

Figure 4: Choke Point deployment model (adapted from [1]).

Issues with The Choke Point Model

Deployment of middleboxes in traditional networks (networks composed of proprietary network devices, which are in most cases hardware devices - routers and switches)

followed the choke point model.

The choke point middlebox deployment model raises several serious concerns when applied to existing (traditional) networks. Three main challenges such deployment faces are:

(a) How can two or more subscribers, having different policy chains, share the same network link?

(b) What are the best positions to deploy middleboxes in networks shared by more than one subscriber, where all susbscribers have different policy chains?

(c) How can connectivity be maintained when a middlebox along a network path fails [1, 5]?

Moreover, deploying middleboxes along a network path (choke point deployment) goes against realizing desirable properties for middlebox deployment, which includes achieving correctness, flexibility and efficiency [5]. For correctness, it is desired that packets travel though a service chain in the correct order. This becomes an issue in a network with more than one service chain (e.g. a data center network). For efficiency, middleboxes should only traverse required middleboxes and for flexibility, the introduction or removal of middleboxes from the network should not disrupt the network [5]. To mitigate the concerns expressed above and also achieve the stated desirable middlebox deployment properties, the *Way point model* was introduced sucessfully [5, 1, 18, 5, 9, 7, 2, 10] to Software Defined Networking.

2.3.2 The Way Point Model

With the emergence of Software Defined Networking (SDN), the Way Point model was adopted in order to overcome the limitation of the on path deployment of middleboxes, which forces all packets going through the path to pass through the middlebox. The Way Point model specifically allows a middlebox to be deployed off the network path [1, 5]. The reason behind deploying the middlebox off the network part is so that Openflow switches can be leveraged to redirect or steer desired network traffic to desired middleboxes. With this approach, network traffic will not need to pass through middleboxes that are not needed.

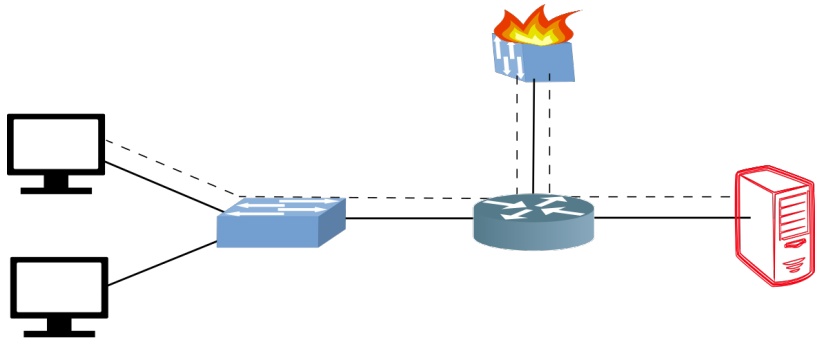

Figure 5: Way Point deployment model (adapted from [1]).

2.4 Middlebox Placement

Middlebox placement considers appropriate ways middleboxes can be deployed on the network topology to ensure that Service Level Agreements are not violated and also enable good utilization of the network infrastructure. For example, depending on the policy chain, the decision to place a NAT before a Firewall on a network may adversely

affect the network if policy chain intents are not considered. Therefore, as the number of policy chains mapped to a network and the number of middleboxes deployed on the network grows, placement can become challenging. In advanced cases, middlebox placement may consider resources available in the network. This is often the case in virtualized environments, where service functions can be deployed on demand. Hence, resources like CPU utilization, available memory and bandwidth characteristics can become decisive factors when deploying middleboxes in the network.

2.5 Packet Steering

Packet Steering is a concept used in packet routing to specifically identify routes a packet must take in order to fulfil the requirements specified in the SLA. Packet steering in middlebox deployment is different from the routing done by routers and switches, in the sense that packet steering in middlebox deployment often considers the unique characteristics about middleboxes deployed on the topology. Characteristics include the action of the middlebox on packets, the number of hops a middlebox is from a given point on the network, the processing and storage capabilities of the middlebox, the link utilization, and bandwidth associated with the path on which the network functions are deployed, and the overall throughput of the network. These characteristics are often used as metrics in deciding the path along which a packet should be steered.

Chapter 3

Related Work

Our work is inspired by the packet steering case for multi-subscriber environment provided in StEERING [7], where each middlebox is deployed from the source based on the number of subscribers using it, and a possible packet steering challenge presented in SIMPLE [9] where the subscriber's policy chain is routed back and forth on the service chain in order to fulfil the policy chain requirement.

As stated in StEERING [7], the goal is to give subscribers the freedom to select network services and also decide how they want packets to traverse these services. StEERING handles multi-subscriber environments however, the examples given in StEERING do not cover similar complex cases for multi-subscriber environments that we consider. StEERING places middleboxes closer to the source based on the number of subscribers using them and does not check to know if SLAs have been violated during packet steering like we do. Hence, StEERING would have to employ some redundancy to forward packets in One Pass. In chapter 6, we show our scheme steers packets in multi-subscriber environments using one pass and fewer resources than our reference model which has been described as ideal [2] and employs redundancy.

SIMPLE also develops a packet steering capable of steering packet back and forth on a chain, however, SIMPLE produces a higher number of hop count than OPPS and the our reference model whereas OPPS produces equal hop count with the ideal model.

A number of works have formulated and adopted different strategies for finding the best possible route that a packet can take through a sequence of middleboxes, from source to destination. In reviewing these packet steering strategies, we see that the solutions to packet steering challenges entails using techniques that encompass middlebox deployment techniques, resource management techniques and service chaining techniques. In this chapter, we discuss these techniques using as sources, related works produce for solving packet steering challenges.

3.1 Service Chaining Techniques

In order to chain a number of standalone service functions deployed on a network to satisfy a given policy chain, the following techniques have been adopted in previous works:

3.1.1 Tunneling

Tunneling is one of the service chaining techniques employed in shared network environments or networks where there are possibilities of service chain loops. Loops occur in cases where packets are required to travel back and forth on the same link before fulfilling the requirement of getting from source to destination. When a flow is tunneled, a tag becomes associated with the flow to enable proper identification of

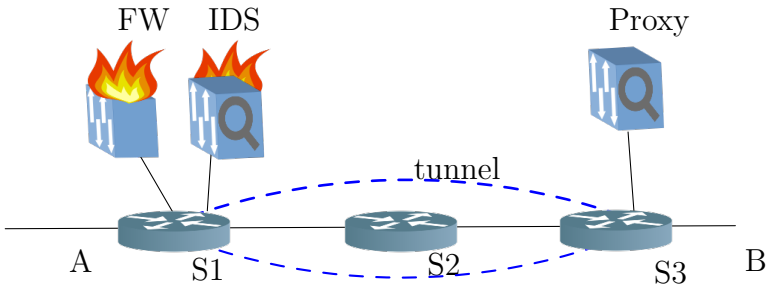

Figure 6: Tunneling technique for packet steering.

processing context. This processing context is the information that reveals information about the middleboxes that have processed the packet. These tagging can be applied using existing IP packet fields, MPLS or VLAN fields. For example, in [9], tunneling is used in the data plane for a loop scenario where the policy requirement for HTTP packets is for packets to go from FW $\rightarrow$ IDS $\rightarrow$ Proxy. This is represented with the diagram shown in Fig. 6 where the source is at A and the destination is at B.

3.1.2 Tagging:

One of the challenges inherent in some service chains is that they can make packets unidentifiable along the service chain. This happens mainly due to the re-writing action of some middleboxes.

The need to preserve a form of unique identification that remains such through out the life of a packet within the middlebox region is very important. To illustrate this need for packet identification, we cite examples from the work by Seyed et. al. [8].

Illustration 3.1.2.1. Fig. 7 shows a network with two hosts, *H1* and *H2*, physically connected to an IP capable network device. It is desired that *H1* has access to the internet cloud, while *H2* has access to only the local network. In the figure, we see that a NAT and a Firewall is deployed between the hosts and the internet.

The question here is: how do we prevent H2 from accessing the internet given that the NAT re-writes the internal IP address of H2 to the public IP before it hits the firewall?

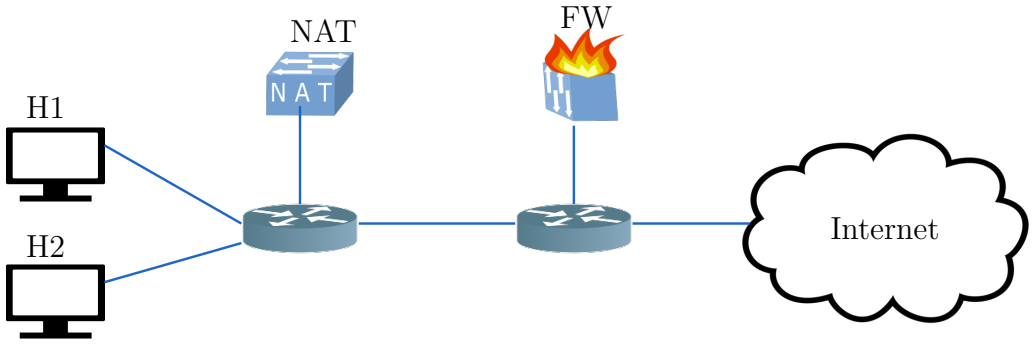

Figure 7: An example showing the need for Path Follow Policy

Illustration 3.1.2.2. Examining Fig. 8, it is desired that *H1* has access to the web page www.abc.xyz while *H2* does not. The challenge here is that if H1 accesses the page, a copy of the page could be cached at the Proxy, and if *H2* tries to visit the page, H2 could get the cached copy that is stored in the proxy.

Approaches to Packet Identification in Service Chains

1. **The 5 Tuples**

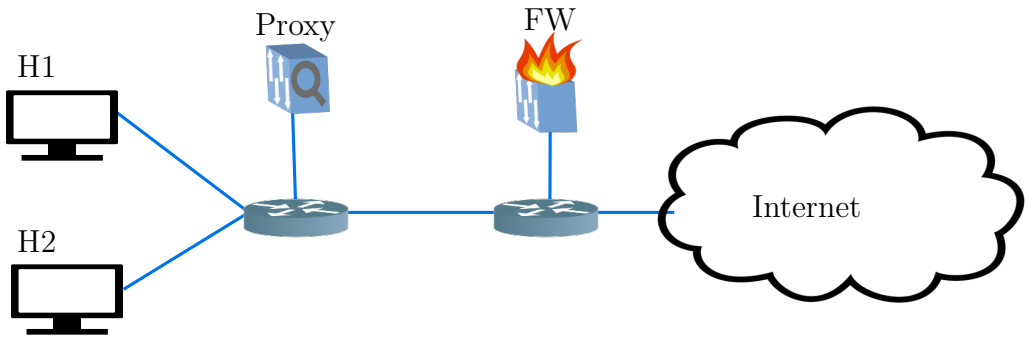

Figure 8: An example showing the need for Path Follow Policy

The 5 tuples consist of *Source IP, Destination IP, Source Port, Destination Port, Protocol*, and it can be used as a unique feature to identify a flow. In PLayer [5], depending on the operation of a particular network function, different IP header features may be used in addition to the 5 tuples, for instance, middleboxes that re-write some part of the header like the load balancer. Also, a different set of IP features may be used in cases were the 5 tuples may not be necessary to identify a flow; for example in a firewall, the identification of packets may depend on MAC addresses. This creates concern of having to particularly investigate all middleboxes to in order to become aware of which header combinations are suitable.

In SIMPLE [9], tags were used to support tunnels for efficient packet steering. The tag in SIMPLE encodes the processing context. By encoding the processing context, a middlebox can differentiate different states of a packet if the packet traverses that particular middlebox more than once. To further illustrate this concept, let us consider example 3.1.2.1.

Example 3.1.2.1. Assume a policy chain states that HTTP packets should be

processed in the order:

$$Source \rightarrow M_j \rightarrow M_k \rightarrow M_j \rightarrow Destination$$

How do we identify appropriate processing contexts in order to be able to differentiate the first instance of the packet at M_j from the second instance of the packet at the same middlebox?

In this policy chain, we observe that middlebox M_j will be visited twice. In order to differentiate between the instances, when the packet is supposed to be routed to middlebox M_k, versus when the packet is routed to the *Destination*, we need to encode where the packet has been, or which middleboxes have processed it in the past. Hence, if the packet arrives at middlebox M_j with the tag attribute *HTTP: Source*, we know that its destination is middlebox M_k, while if the packet arrives at M_j with tag attribute *HTTP:* M_k, we know that the packet should be forwarded to the destination as indicated on the policy chain.

One of the problems with the tagging approach [9] is that it advocates the use of existing IP related header fields, such as the MPLS label, ToS and VLAN fields, to encode middlebox information. In particular, ToS fields and VLAN fields were used in SIMPLE. However, middlebox operations and context information, as well as MPLS and VLAN operations are not mutually exclusive operations. Similarly, with the approach employed in [6, 8], one question that remained unanswered is the compatibility of these works with existing standards like the popular Differentiated Architecture standard [19] that uses the

same field. One thing that remains clear is that service function operations or protocols and routing standards or protocols are not mutually exclusive. Thus, service function operations and protocols should be able to run on DiffServ networks and likewise, any service function protocol or operation should be able to run over MPLS established paths and in VLAN environments, as well as be compatible with existing IPv4 and IPv6 protocols.

In [20], it has been noted that service chaining introduces a new network plane to the existing network architecture. Similarly, in our work we add a sub-layer between the IP layer and the transport layer adding an 8-byte long header field, to solve the packet steering problem for stateless policy chains. We envisage that our special field header can be used in networks running architectures and standards like the Differentiated Architecture standard [19], VLAN and MPLS.

In our approach, we push context derivation from the controller (as in [6, 8]) to the OPPS module and deduce the processing context using the lightweight OPPS module operations. One may argue that in a stateless policy chain, the need to derive middlebox processing context is trivial. However, we showed that this is non-trivial, with the example presented in Chapter 1, Section 1.2.

2. **Flow Correlation**

   Flow correlation is another technique used in packet identification when the 5 tuples are insufficient for identification [9]. This often occurs when a middlebox re-writes one or more fields in the 5 tuple, e.g. NATs re-write packet headers while translating addresses to either public or private. The intuition in flow correlation is that if a packet's header gets re-written, then its payload, which remains intact, will be used in identifying the packet. Hence, the payload of the

input packet is compared with the corresponding output packet payload and if they are the same, then the packet can be effectively identified and linked to the corresponding packet and its properties. One challenge with this approach that has been identified in [8] is that matching packets can be computationally expensive and also error prone; for example, when the entire packet is modified.

3. **Unique Identifier Tag**

   FlowTag [6] is another work that employs the use of the 6-bit ToS field in IPv4 for tagging. As discussed in the *FlowTag* paper, one limitation with the 6-bit ToS field is that the number of tags a 6-bit field can support is quite limited. In [6], this tagging feature does not apply to flow, rather it applies to a flow per middlebox context. Therefore, a flow may have two or more possible different contexts at a middlebox similar to the illustration in example 3.1.2.1. Seyed. et al. [8] built on their previous research in *FlowTags* [6] and used *FlowTag* to enforce policy in middlebox deployments. Revisiting illustration 3.1.2.1, Seyed et. al. identifies that the example in the illustration violates Origin binding (explained below). To fix the Origin binding problem, a unique tag is inserted in the ToS field of the IPv4 header and the flow label field for IPv6. The purpose of the tag is to identify subscribers *H1* and *H2* respectively so that when the IP address is re-written to a public IP address by the NAT, flow matching at the switch and packet filtering at the firewall can be done using unique tags. However, this may need a modification of the firewall to accommodate identification and packet scanning based on new tags.

   Seyed et. al. identified two desirable properties for packets traversing a service chain. They are Origin binding and Path follow policy.

Origin binding: In FlowTags, Origin binding is identified as a desirable property of packets. Origin binding implies that a packet should be self-identifiable throughout its life along the service chain.

Path follow policy: The second desirable property of packets traversing a service chain is the Path follow policy. Seyed et. al. described the Path follow policy as the ability of packets to traverse the service chain as specified by the policy chain, irrespective of processing by service functions along the service chain.

In *enforcing network-wide policies*[8], a Dynamic Policy Graph (DPG) is used to identify all possible paths and the context that a packet can have at any particular location on the service chain. The dynamic policy graph runs at the controller, hence the controller takes part in deducing middlebox context.

Given the challenge in illustration 3.1.2.1 and from the definition of the path follow policy, we observe that there is a violation of the policy. Seyed et. al. solves this problem by using the Origin binding policy and the DPG to identify enforce path policy. At each location along the service chain, the middlebox sends the middlebox context and tag information to the controller. The controller deduces the appropriate context or tag information using the DPG and responds back to the middlebox. This context information enables the middlebox to apply further processing or make deductions on the state of the packet and what the next destination of the packet down the service chain should be. A downside of the DPG is that it requires an expert to be able to enumerate all possible paths and corresponding outcomes before drawing up a dynamic policy graph. Moreover, listing all possible paths and outcomes from middlebox

processing on a large network could be error prone.

4. **OpenFlow Metadata**

   Zhang et al. [7] employs the use of the metadata field in OpenFlow (OF) to encode traffic direction and services. While our work is similar to StEERING [7] with respect to investigating packet steering, our work explores steering packets in one pass through the middleboxes with the aim to eliminate the need of traversing a link back and forth. Our work focuses on developing an efficient steering technique that steers a packet with respect to the underlying physical infrastructure. Like SIMPLE [9], we do not address placement in this work. Zhang et al. [7] and later Liu et al. [21] solved the end-to-end problem for multi-subscriber environments by placing most used middleboxes closer to the source. The logic for such placement is to minimize the packet latency between source and destination. While this approach minimizes the latency for many subscribers, it does not solve the latency problem for all subscribers. In our work, we aim to achieve a result that yields nearly comparable end-to-end delay for a group of subscribers having the same set of service functions in their policy chain, but each with different ordering.

3.2 Middlebox deployment Techniques

3.2.1 Middlebox Consolidation:

Another technique used in middlebox deployment is consolidation (shown in Fig. 9). Middlebox consolidation is the act of aggregating middlebox software functions into a single hardware entity. This occurs in often in virtualized network function

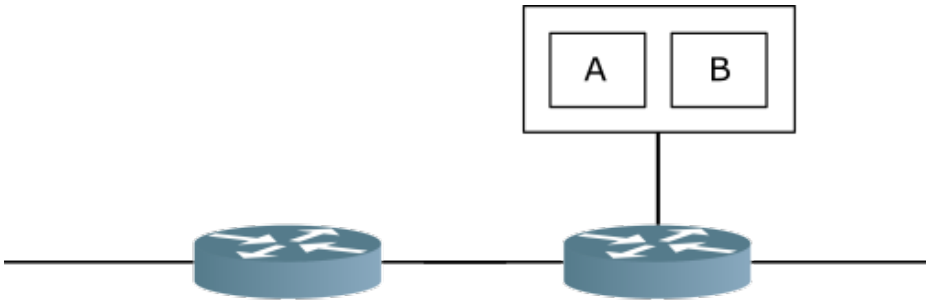

Figure 9: Consolidated middlebox environment (adapted from [2]).

environment (VNF). Middlebox consolidation is of two types - strict consolidation and non-strict consolidation. In strict consolidation [10], all middlebox software functions are aggregated into a single hardware while in non-strict consolidation [2], middlebox deployments are aggregated into groups based on defined parameters, usage or policy chains, and then each group is deployed on separate commodity hardware. The main aim of consolidation is to leverage the cost of commodity hardware, which is cheap compared to specialised hardware resources, and also to leverage the capability to modify and create new software functions. In consolidated middleboxes, packet steering is done within the different software deployed on each hardware. Although consolidation leverages cheap resources, it has been noted that a major limitation of middlebox consolidation is that it requires huge infrastructural change made to the network [8]. This is because most traditional networks rely heavily on proprietary middleboxes. Also, for proprietary middleboxes produced by different vendors, the process of integrating different vendor software into one commodity hardware raises issues relating to violation of the *Origin binding* and the *Path follow policy*. In our work, we adopt a non-consolidated model in testing OPPS, however, we note that OPPS can be deployed in VNF environments.

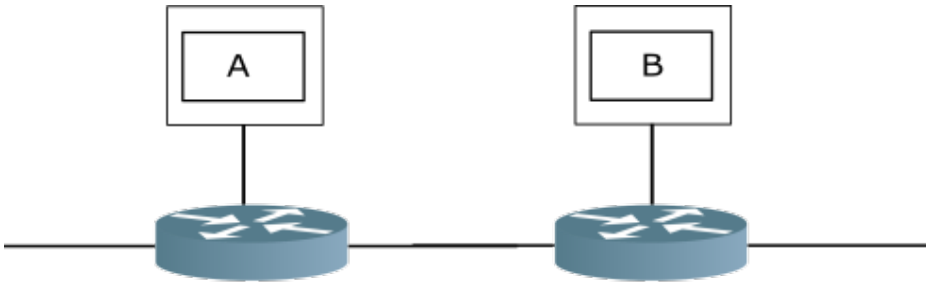

Figure 10: Non-consolidated middlebox environment (adapted from [2]).

3.2.2 Middlebox Non-Consolidation

In non-consolidated middlebox environments (shown in Fig. 10), middleboxes occur as standalone hardware elements as found in traditional networks [5, 7, 9]. Hence, hardware elements like the NAT and Firewall will exist on separate hardware devices, which are often proprietary.

3.2.3 Deployment at Functional Level

OpenBox [22] adopts the idea of uniting different middleboxes specified in a service chain by functional unit similarity. It achieves this by decomposing middleboxes into functional units similar to Click elements [23] and chains this set of functional units at the data plane to conform with the policy chain specified at the control plane. Due to low level decomposition of a given service chain, OpenBox preserves chaining at the data plane by attaching results to metadata. Similarly, in this work, we attach results from middleboxes to our custom header to be able to decide when it is sufficient to make a decision based on the accumulated results. Although OpenBox does not make provision for multi-subscriber environments while our scheme does, we envisage that original equipment manufacturers (OEMs) will continue to produce standalone network function equipment. Moreover, if OEMs were to release software

versions of network functions, then they would still come in standalone forms, so the hardware requirements to run two or more of these software network functions may vary, and users (enterprises) may purchase these software network functions at different times as the need arises. Hence, they may opt for extra commercial hardware having unique configurations. To cater for these uncertainties, our design looks at middleboxes deployed as standalone devices with extensible interface.

3.3 Resource Management Techniques

One of the benefits of Network Function Virtualization (NFV) and Software Defined Networking is the ease with which networks can leverage shared resources to provision service functions. Provisioning network resources comes with the challenge of optimal resource allocation and researchers have looked in to the problem in attempts to find optimal resource allocation models and management in SDN environment.

The problem of resource management in SDN can be broadly categorized into two problems: the Online resource management problem and the Offline resource management problem [24].

3.3.1 The Online Problem

The online resource problem entails using constantly changing network resources in making network decisions that ultimately affect either, how flows are steered in the network or how service functions and service chains are instantiated on the network. These constantly changing network parameters may include current CPU usage, link utilization and memory constraints.

3.3.2 The Offline Problem

In the offline problem, the offline resource management algorithm uses fixed or known network parameters to calculate and make decisions. Decisions such as, to identify the appropriate places in the network to place certain service functions or how to best to distribute flows and service chains in a network, are made in the offline resource management stage. One unique feature of the offline problem is that its algorithm is used once, which is usually on network start up, and actions (which may include placing middleboxes at appropriate locations) are taken based on the results of the offline algorithm before the network starts running. Parameters used in the offline problem include link capacity, bandwidth, CPU capacity, storage capacity, expected traffic volume, etc. Unlike the online problem, the offline parameters are expected to remain defined.

Some of the common methods used in modelling the resource management problem include ILP, MILP and Shortest paths. The LP technique has been identified as a computationally expensive approach and results [25] have shown that MILP consumes a lot of time. Therefore, it is not suitable for solving the online problem in large scale deployments. Table 1 shows a summary of other algorithms developed to solve different online and offline problems related to resource management.

Table 1: Summary of Works on Resource Management

Paper	Category	Aim	Algorithms
[25]	Online	To minimize usage of network resources by offloadding overutilized links, thereby saving operational costs.	MILP, heuristic (Consolidation) Algorithm.

Paper	Category	Aim	Algorithms
[26]	Online	Allocates link capacity and VM resources to maximize the number of demands (packets) a network can handle.	MP.
[21]	Offline	Places middlebox in network such that the end-to-end delay and bandwidth usage is minimized.	Heuristic (Greedy) algorithm and Simulated annealing algorithm.
[9]	Offline	To distribute or allocate service chains without violating constraints on switch capacity.	ILP.
[9]	Online	To minimize load on middleboxes in the network (load balancing).	LP.
[27]	Online	Minimize CPU, link utilization and minimize delays in the network through efficient middlebox placement and flow steering.	MILP.
[28]	Online	To minimize end-to-end latency with imposed constraints on capacity of service functions by traffic steering.	ILP.
[24]	Online	To maximize the traffic a network can handle.	COATS.

Paper	Category	Aim	Algorithms
[24]	Offline	To scale a class of traffic to fit service chains in a policy aware network without violating middlebox constraints.	ILP problem reduced to FPTAS.
[7]	Offline	To minimize total delay (modelled as hop-counts) for all users.	Greedy heuristic algorithm.
[2]	Offline and Online	To place middleboxes at locations that minimize utilization of machine resources, bandwidth and latency.	Inflation Rates.
[2]	Online	To steer a packet through a sequence of middleboxes without violating middlebox loading capacity.	Weighted Shortest Path.

Lessons learned, possible directions and challenges from cited works in Table 1 include:

- Similar to OPPS, the reviewed works on Table 1 aim to improve network parameters and resources. In OPPS, we attempt to reduce the delay and provide comparable throughput for all subscribers. However, our approach to improving network parameters is new.

- Online problems involving middlebox placement occur in virtualized network function environments and are usually deployed through the creation of service function instances. Although we do not treat OPPS in virtualized environments, we note that OPPS can be deployed in virtulized environment.

- Modelling the middlebox placement problem, resource management problem, and steering problem as an optimization problem is the most common technique used in middlebox management. The works [24, 9, 27] have identified optimization techniques such as LP's, ILP's and MILP's inefficient in terms of computation and time complexity. In this thesis, we take a different approach. We introduce a different packet steering technique called One Pass Packet Steering (OPPS) that reduces the number of middleboxes deployed while still maintaining a least number of hop count attainable on a service chain.

- The works [26, 21] have proven that the placement problem is NP-Hard and SIMPLE [9] identified the problem of flow distribution (load balancing) while considering constraints on middlebox and switch resources as NP-Hard. Also, the work [2] has identified packet steering over middlebox placement as the bane for more improved end-to-end performance, thus giving us stronger motivation to focus on packet steering in subsequent chapters.

- Works listed in Table 1 have implemented algorithms solving different online and offline problems however, what remains to be seen is a work that compares the different approaches mentioned above. A major challenge for this possible future work would be re-producing exact results on some of these works due to brief descriptions of algorithms, unstated assumptions and brief testbed descriptions and requirements. For this, we adopt a model defined in [2] as an ideal model for benchmarking OPPS. In our work, we refer to this ideal model as the redundant model.

Chapter 4

One Pass Packet Steering

4.1 OPPS System Overview

In chapter 1, we defined the stateless policy chain and gave an example. We proceed by giving an example of a stateful policy chain, outline the assumptions made, and then we discuss the system in general. Consider a policy chain requiring that packets go through a NAT, a Traffic logger and a Firewall in that order. The behaviour of the NAT will affect the identity of traffic logged by the traffic logger. It would pose a challenge to the firewall if it filtered packets based on the IP address of the original source. Therefore, with a NAT in the policy chain, the results from middleboxes down the chain are affected; hence, we say that the policy chain with a NAT is stateful. Previous solutions [6, 8] have identified and solved this problem by adding an extra unique identifier label in packet tags hence, we do not focus on this problem. *FlowTag* [6] can be extended with our work to provide this support.

We advocate that middleboxes should be modified such that the OPPS module takes control of carrying out certain middlebox actions. To simplify the objectives, we

assume that packet fragmentation does not occur within our middlebox island. We refer to a middlebox island as a cluster in a network that contains only middleboxes. We also assume that middleboxes recognize the newly-added fields. Fig. 11 shows the modules that make up OPPS. In the subsections below, we describe the core modules that make up the system.

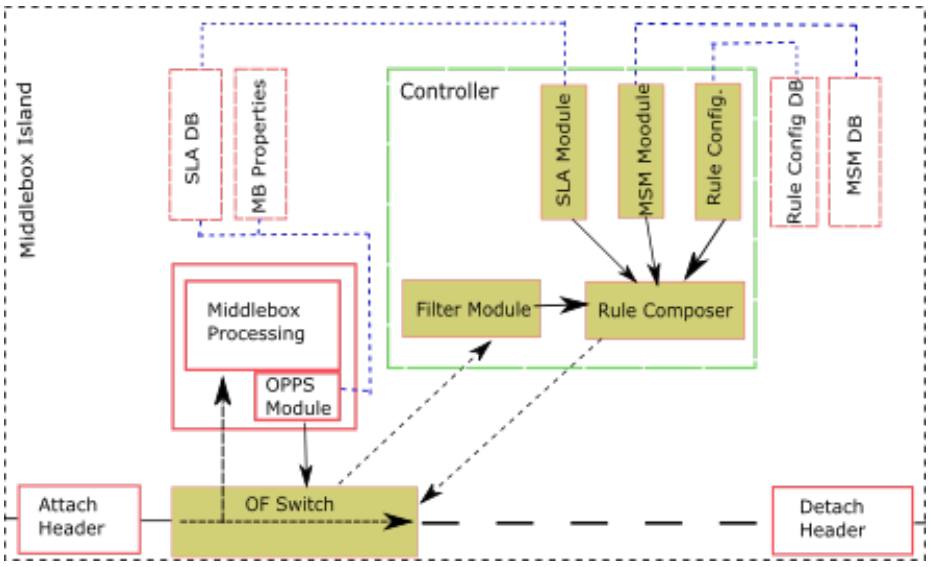

Figure 11: One Pass Packet Steering Architecture (Other OF Switches and middleboxes in the island are chained along the dashed line).

4.1.1 The Controller

SDN has the capability to automate and deploy changes like policy chain and topology changes in real time. Hence, we design OPPS for Software Defined Networking Environments. In this subsection, we describe the different modules that make up the SDN Controller.

- **The Filter Module:** The controller is composed of a learning switch capable of differentiating layer 2 requests from layer 3 requests. We are more interested

in layer 3 packets and layer 3 requests because they hold contents from outside the network, which need to be handled specially according to the SLA. For this reason, we also introduce extra supporting modules.

- **Middlebox to Switch Mappings (MSM) Module:** This module accesses and maintains a virtual representation of the topology in memory. This enables knowledge of the exact position of middleboxes within the linear topology, and the switches directly connected to these middleboxes.

- **Service Level Agreement (SLA) Module:** The SLA module holds each subscriber's identity (a unique identifier) and their corresponding policy chain. We note that unique identifiers should be a network labels or a combination of labels that can be used to match packets in an Openflow switch and also uniquely identify a subscriber's policy chain in the SLA database. These identifiers are needed for formulating subscriber specific flow rules and for routing packets through the Openflow switches. The choice of this unique identifiers can be IP addresses for a network whose internal network addresses has been resolved by a NAT, a combination of IP addresses and ports, embedded unique identifiers in Openflow meta data or any other combination of unique labels that can be used in the Openflow switch for matching subscriber specific packets. In our case, subscribers care about the traffic going to specific services which are identified by IP addresses. Hence, in our implementation, subscribers are identified by the destination address in the IP packet header.

- **Rule Configuration Database (Rule Config DB in Fig. 11)** The Rule Configuration Database contains configuration files that help the creation of

rule templates by the Rule Configuration Module. Fig. 25 shows sample entries in the configuration database.

- **Rule Configuration Module (RCM):** The RCM uses configuration parameters in the Rule Config database to create a default flow rule template for switches. This default template takes into account the direction of packet flow and fixed value parameters. In our work, examples of fixed value parameters include the flow rule action, input port numbers and output port numbers[2]. For the direction of packet flow, we have the *host to server direction.* and the *server to host direction.*

  The Host to Server Direction: The RCM creates a rule template that indicates for each flow per switch whether a flow rule is *"Active"* or *"Default"*. *"Active"* means that the subscriber's SLA policy contains a middlebox mapped to the switch, while *"Default"* means that the subscriber's SLA policy does not contain the middlebox mapped to the switch. When the flow per switch template is marked "*Active*", the output port configured by the Rule Composer Module (discussed later in the text) is the port connected to the middlebox attached to the switch, while when it is marked *"Default"*, the output port is the port connecting to the next switch along the direction of flow. These markings help achieve a dynamic, yet simple and extensible flow modification message customization.

  The Server to Host Direction: This configuration is similar to the *host*

[2]We use the same input port and output port number configuration for all switches e.g., for each switch on a service chain, the input port from the previous switch or host can be configured to port 1, the input port from a middlebox can be configured to port 3, the output port from the switch to a middlebox can be configured to port 2, and the output port from the switch to the next switch or host can be configured to port 4

to server direction configuration, except that the flows considered are those moving from the server to host. In our work, all flows in this direction were set with the *"Default"* label. Thus, we do not traverse any middlebox in the server to host direction.

For each of the direction and middlebox-switch indicator markings (*"Active"* and *"Default"*) specified above, the following matching parameters are specified: *in_port* and *dl_type*. Also the corresponding *actions* are specified. The *in_port* denotes the ingress port and the *dl_type* denotes the ethernet frame payload (that is IP or ARP). These parameters form the basic and complete default rule necessary to satisfy the policy of a subscriber with no service functions specified in the SLA.

- **Rule Composer Module:** The Rule Composer Module transforms high level input from the MSM module and SLA module to low level OpenFlow messages with the help of the RCM module.

4.2 The Middlebox Header

In designing the OPPS technique for a multi-subscriber environment, we devise a means to track the middlebox processing context by creating a new sub-layer. Fig. 12 shows the details of the new sub-layer added to the network header. The sub-layer lies between the Network (IP) header layer and the Transport (TCP) header layer. This sub-layer has been designed and its placement decided after consideration of the need to have easy portability to IPv6 in the form of extension headers [29], for ease of extensibility both in size and function, and without modification of existing network

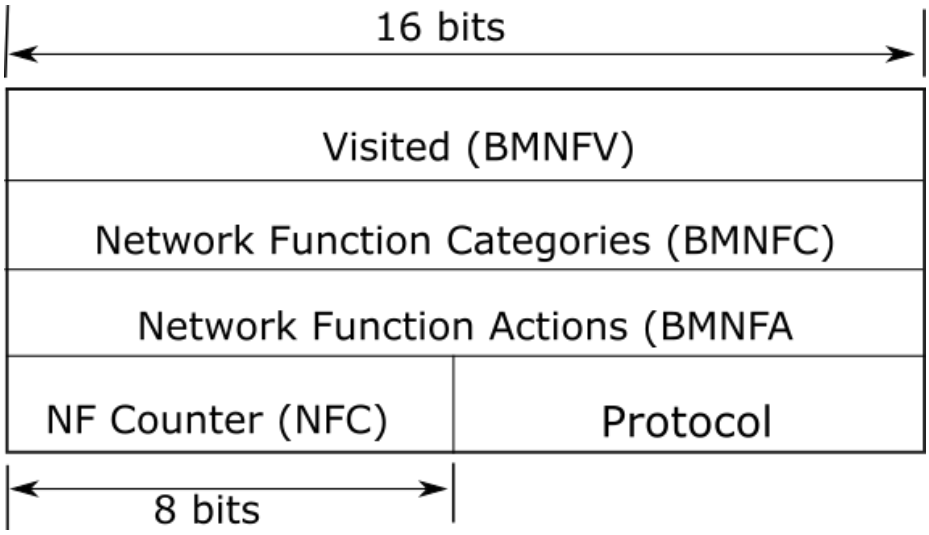

Figure 12: Middlebox Header Fields .

switches. The description of the middlebox header fields is as follows:

- **BMNFV field:** This field stands for *Bit Mask of Network Functions Visited* and is 16 bits in length. Each bit in the 16 bit sequence is used to indicate if a network function has been visited. Thus, when a network function is visited, its position in the topology is mapped to the 16-bit sequence accordingly and set to "1", otherwise it remains "0" to indicate that it has not been visited. The 16-bit length of the field currently puts a constraint on the maximum number of network functions that can be supported by the network to 16. This can be increased by increasing the size of the BMNFV field, however this number is sufficient for implementation in this work.

- **BMNFC field:** The 16 bit BMNFC field, which stands for *Bit Mask of Network Function Categories*, is used to specify a category for each service function on the service chain. There are two categories. The first is a category of service

functions that tend to drop or halt the forwarding of a packet along the service chain. This category is marked with the *"1"* bit. Examples of middleboxes in this category include the Firewall and Intrusion Prevention System. The second is the category of service functions that have no tendency to halt or stop the forwarding of a packet. Examples of network functions in this category include a Traffic logger, an encryption network function, and a packet classifier. This category is marked with the *"0"* bit.

- **BMNFA field:** The BMNFA field stands for *Bit Mask of Network Function Actions*. This field holds the result of each service function at the corresponding bit position in the BMNFA field. Similar to the BMNFC field, BMNFA is a 2-byte field. The 16-bit sequence is a bit map of the position of middleboxes in the service chain. The field contains either 1 or 0 at each bit position. Each bit position in the BMNFA field and the corresponding bit position in the BMNFC field enables us to choose from up to four possible different outcomes/actions. In our work, we have two actions (*drop* or *forward*). We use "1" for "*drop* packet", and "0" for "*forward* packet" when the corresponding bit position in the BMNFC is also *"1"*. For a "0" in the BMNFC field, the corresponding bit position in the BMNFA field is set to "1" [3].

- **NFC field:** The NFC is an 8-bit field used as a counter to count the number of network functions visited. NFC stands for *Network Function Counter*. The 16-bit size of the bit mask fields puts a constraint on the number of bits actually used by the NFC field[4].

[3] In our small scale implementation, it is possible to eliminate the BMNFC field, however we leave it for completeness.

[4] It is possible to derive the network function count from the BMNFV field by counting the number

- **Protocol field:** Functionality wise, the *proto* field is similar to the *Protocol* field of the IPv4 header [30] and the *Next Header* field [29] of the IPv6 header. Thus, it specifies the next level protocol in the packet.

4.3 The OPPS Module

4.3.1 Intuition

Before delving in to the algorithms and operation of the OPPS module component, we go through some elementary operations that will be beneficial to understanding the algorithms. We know that the XOR of two numbers X and Y equals zero if the numbers are equal, else it is greater than 0, if the numbers are different. For example, let X = 5, Y = 7 . In binary,

$$X = 101, Y = 111$$

$$X \oplus Y = 010$$

which is 2, i.e., greater than 0.

$$\text{If } X \equiv Y \equiv 5$$

then

$$X \oplus Y = 000$$

Now assume we are given an array of numbers where each number occurs exactly twice except one and we are told to find the number that occurs exactly once. Using the operation above, we can XOR each value by the one before, and when a value XORs itself, we can safely say the value has been eliminated in the array.

of bits set to 1, however, in our work, we use the NFC field for counting to aid faster processing.

Table 2: Intuition on operations in algorithm

Iterations	1st	2nd	3rd	4th	5th	6th	7th
Values	4	3	4	3	2	5	5
Binary	100	11	100	11	10	101	101
RES (R)	100	111	011	000	010	111	010
Analysis	$R > 0$	$R > 0$	$R > 0$	$R \equiv 0$	$R > 0$	$R > 0$	$R > 0$ $R \equiv 2$

For example, consider the array:

$$\{4, 3, 4, 3, 2, 5, 5\}$$

The first occurrence of each value is coloured blue and second occurrence of each value is coloured red. Let us perform a *"find"* operation to find the number that occurs just once. For our operation, we will store the result in variable *"RES"*, which is initialized to zero. Hence, for each array element X, we will perform the operation $RES = RES \oplus X$. Table 2 shows the result and analysis after each iteration of the above operation.

Looking at Table 2, we can assume that after the 4th iteration, it is sufficient to say that array elements from index 0 to index 3 appeared twice. Similarly, we can also assume that after the 4th iteration, we are back to the initialized value of RES, thus, we can say we are starting a new set of *"find"* operations on the 5th iteration.

Revisiting the original question, if we divide the array into two separate arrays, one consisting of values with blue colours (first occurrence) and the other consisting of red colours (second occurrence), it will give:

$$Array_A = \{4, 3, 2, 5\}$$

$$Array_B = \{4, 3, 5\}$$

If we state,

for each value *"Y"* in $Array_A$ and value *"Z"* in $Array_B$, find the value in the longer array, which does not appear in the array with the smaller length,

this will involve iterating both arrays *"N"* number of times and performing the operation

$$RES = RES \oplus Arrray_A[N] \oplus Array_B[N]$$

were *"N"* is the length of the longer array. Hence, this operation will give the same result as in Table 2. We can conclude from Table 2 that at any point $RES \equiv 0$, it is sufficient to say that all values from the beginning of the array till that point has occurred twice.

4.3.2 Module Description and Algorithms

The OPPS module sits at the outgoing interface between each middlebox and the switch as shown in Fig. 11. When a packet is forwarded to a middlebox, it is processed by the middlebox and then goes through the OPPS module. The OPPS module is the module that performs the bit manipulations on the middlebox header fields. Its functions include updating middlebox header fields, deciding when it is sufficient to implement an action, and to implement an action.

The *Rule Composer* module composes flow rules, which are proactively installed on the switches and used for packet forwarding.

When a packet first enters the middlebox island, it is forwarded to the first ingress network unit whose specific function is to introduce custom header fields between the IP header and the transport layer header. The packet is then forwarded to the next middlebox on the topology (that is contained in the subscriber's policy chain) irrespective of the middlebox's position in the policy chain. The middlebox processes the packet and decides to either drop or forward it. The decision made ("*action*" in Algorithm 1), the packet and the middlebox identifier[5] are passed to the OPPS module. In the OPPS module, these fields are received by the *PacketHandler* and are passed to the *UpdateHandler* as in line 5 of Algorithm 1. In the *UpdateHandler*, the middlebox identifier is used to retrieve the middlebox's category which is either "0" or "1" (line 3 of Algorithm 2). Operation "*IndexOfLeastSignificant1bit*" is used to find the index of the least significant 1 bit in the bit mask (note, there is only one bit having the value of "1" in the bit mask; the rest have "0" value) as in line 5 of Algorithm 2. The category and action are left shifted by the index (lines 6, 7) and set by appropriate bit operations (lines 8, 10) to align with the index of the least significant bit having value "1" in the bit mask, while the middlebox bit mask is XORed with the visited field in the packet header and updated in the header (line 9). Actions of middleboxes that are passive with respect to the life of a packet in the network (i.e. middleboxes whose bit positions in the BMNFC field are set to 0) are implicitly assumed to be *"forward"* and are encoded as 0 in the BMNFA field. Similarly, middleboxes whose actions include dropping a packet (i.e. middleboxes whose bit positions in the BMNFC field are set to 1) are encoded as 1 in the BMNFA

[5]The middlebox identifier is a locally unique name assigned to a middlebox on start up based on its position on the topology. This locally unique value can be formulated or retrieved by the middlebox or the OPPS module from a topology database. In our implementation, the identifier equals the bit mask of the middlebox on the topology ("*maskOFNF*" in Algorithm 1).

field for a *"drop"* action and 0 for a *"forward"* action. It is important that these field operations happen in deterministic time intervals that are always of the same value i.e., $O(1)$ or are of near constant time as they could have severe impact on the overall latency and jitter experienced by the system. In the algorithms, these operations are performed using mod(2) arithmetic.

Once the update is done, the OPPS module checks if such update is sufficient for a decision to be implemented using Algorithm 3. To be "sufficient for a decision to be implemented" means that, for a packet at a particular location on the topology, a minimum ordered set of consecutive service functions in the policy chain must have been visited. Hence, if it is sufficient to decide, then we make a decision whether to drop or forward the packet based on the BMNFV, BMNFC and BMNFA fields, as in Algorithm 4. The Action Implementer (Algorithm 5) finally implements the action, which is either drop or forward. This works within the scope of stateless policy chains because we only care about two outcomes - either the packet "*lives*" (transformed, modified, or remains untouched) or "*dies*" (gets dropped) within the middlebox island. One of our goals is to ensure that the SLA is followed and that a packet would not visit a middlebox it was not supposed to visit.

4.4 Sample Walkthrough

Different subscribers have different needs, and request that packets traverse some or all middleboxes in a different order. We will consider an example of how a malicious packet gets dropped in the network only after achieving the minimum requirement that satisfies its service level agreement. Consider an enterprise that hosts services and servers for subscribers with a chain of middleboxes deployed on the topology

Table 3: Middlebox actions and header values from Sample Walkthrough (Section).

Row label	Description	Field value after visiting middlebox			
		A	B	C	D
1	Middelbox category	c = 1	c = 1	c = 1	c = 0
2	maskOfNF (in Algorithm 1)	M_A = *1000*	M_B = *0100*	M_c = *0010*	M_D = *0001*
3	Packet fields ($BMNFA/NFC/BMNFC$) equivalent to (actions / counter / category) in Algorithm 2	0000 / 1 / 1000	0100 / 2 / 1100		0101 / 3 / 1100
4	Packet field visited, $V_{\text{before}} = M_x \oplus V_{\text{initial}}$ $x \in \{A, B, C, D\}$	1000	1101		0100
5	$V_{\text{after}} = V \oplus orderedListOfNFs[counter]$ $orderedlistOfNFs = [M_{\text{D}}, M_{\text{A}}, M_{\text{B}}]$ in Algorithm 3 $V_{\text{initial}} = V_{\text{after}}$ from previous column	1001	0101		0000
6	*if* it is sufficient to decide *then* make decision, implement decision *else* forward .	$V_{\text{after}} > 0$: decision: Forward	$V_{\text{after}} > 0$: decision: Forward		$V_{\text{after}} == 0$: decision:Drop B is Category 1 and says Drop

in the order A, B, C, D. Subscriber Z wants packets to traverse the middleboxes in the order $D \Rightarrow A \Rightarrow B$. We present Table 3 to illustrate this example with the corresponding results obtainable using the algorithms listed. The first column shows the row labels to denote the action sequence in the OPPS module, the second column shows the descriptions of actions and fields, and the remaining columns A, B, C, D show the values at each middlebox.

A walkthrough of a malicious packet detected to be dropped by middlebox B would mean that the packet passes through the *Attach Header* network function (as in Fig. 11 and Fig. 26), which marks the entry point of a packet in to the middlebox island. The *Attach Header* network function inserts an empty field label between the IP layer and the transport layer field label[6]. The packet is then forwarded to the first

[6]The idea behind this empty label is to have a field that the OPPS algorithms can store and retrieve values from across middleboxes

middlebox on the topology (in this case middlebox A) that is also listed in the policy chain[7]. At middlebox A, the packet is processed by the middlebox, the verdict of the packet is passed to the OPPS module and the action, counter and category fields are updated with values on row 3 of column A. The mask of middlebox A on the topology (M_A) is given as 1000, thus, the visited field is updated by a XOR of this mask with the visited field itself. All these update operations are done with Algorithm 2. We note that, the visited field has the same bit size as the action and category field, and it is initially set to zero. After updating the fields, we check whether we have satisfied a minimum consecutive portion of the policy chain. This operation is shown in row 5 of table 3 as a XOR operation between the visited field and the mask of the *counter*-indexed middlebox in the ordered list of masks that make up the policy chain. This is done by Algorithm 3 (Lines 7 and 8). We use the XOR operation so that when a service function in the policy chain has been visited on the topology, the bit position of the middlebox in the BMNFV field will be masked out (set to 0). In row 6, the returned value (V_{after}) of Algorithm 3 is passed to Algorithm 4 and a *forward* decision is made (table 3, row 6 - column A). Thus, after the packet goes through row 1 to 6 of the column for middlebox A, the packet is forwarded to middlebox B. Middlebox B detects the packet as malicious and the cycle (row 1 to 6 of column for middlebox B) is repeated but because the value of the BMNFV field (V_{after}) is greater than 0, therefore, it is not yet sufficient to decide (i.e., the criteria to execute Algorithm 4 has not been satisfied). Hence B forwards the packet to D. In D, the cycle is repeated and the module checks if the BMNFV field equals 0. It is observed that the XOR of the *visited* field with the first 3 middleboxes' bit mask

[7]We note that packet routing at the switch is carried out using subscriber specific flow rules created by the controller

in the policy chain and the first 3 middleboxes' bit mask in the service chain is now equal to zero. Hence the requirement that a minimum ordered set of middleboxes in the policy chain must be visited before a packet can be dropped, has been met and the packet is dropped.

Now let us consider another scenario for the same subscriber still maintaining the same policy chain. We recall that the policy chain specification for this subscriber is $D \Rightarrow A \Rightarrow B$. In this case, let middlebox A detect the malicious packet on the topology. Clearly, if we are to proceed with the Algorithms as listed, then the packet would visit middlebox B before reaching middlebox D when it has already been marked for a drop, thus, resulting in inefficient traversal of the middlebox. Also, the service level agreement is violated. Recall that we are more interested in dropping packets efficiently, because as long as a packet constitutes legitimate traffic, it will traverse the required middleboxes specified in the policy chain. Since the middleboxes do not affect the results of middleboxes down the service chain, we can go through the service chain, traversing middleboxes out of order. Therefore, as a principle, if we must keep moving down the service chain when middlebox A says drop, we should only traverse middleboxes that come before A on the policy chain. Hence, this violates that principle. To see a simple example of what could result from visiting middlebox B, assume middlebox B is of category 0 and perhaps it computes statistics using packet information from legitimate traffic only; therefore, with the OPPS, the computation results of middlebox B will be wrong. This puts a challenge on OPPS and in section 6.3, we explore a way to overcome this.

Chapter 5

Setup and General Evaluation

In this Chapter, we describe our emulation environment and discuss in detail how we implemented our Service Chaining test bed in Mininet to support OPPS. We further evaluate the performance of the OPPS Module and examine its impact on latency. Finally, we look at some challenges with OPPS.

5.1 Experimental Setup

Our emulation was done using Mininet running in a virtual box virtual machine, on a PC having an Intel core i5 2.6GHz processor and 8Gb random access memory.

5.1.1 Mininet Overview

Mininet is an emulation software used for building prototypes and analysing real network scenarios [31]. Mininet uses Linux containers to emulate switches and hosts, which can collectively be referred to as nodes. To interconnect nodes, Mininet creates virtual interfaces on nodes and uses Linux bridges to connect nodes and thus emulate

links. These links can be fine tuned through the Mininet API to set network link characteristics. Since Mininet is a network emulation tool, it uses real packets. Also, prototype networks built in Mininet can be tested using real network testing tools like Iperf and Ping commands.

5.1.2 Preparing Test Environment in Mininet

By default, Mininet does not support deployment of middleboxes. However, if one thinks of middleboxes as a piece of code or program running within a node that receives a packet on one interface, processes it, and pushes it out the other interface, one can implement an environment that supports the deployment of middleboxes. In our setup, middleboxes were emulated with custom Python scripts implementing different processing functions, running within Mininet hosts, and having the capability to process and forward packets sent to them. The ease of implementing and deploying extra custom functions, using real packets and real network testing tools to analyse network traffic makes Mininet a good choice to test the performance of OPPS.

By default, when a packet is sent to a network interface, it is checked upon its arrival at the interface for whether its destination MAC address matches that of the machine or node. If it is a match, then the packet is passed up the stack from the IP layer to the higher layers and eventually to the corresponding application layer for further checks and processing. If the packet is not a match, then the packet is discarded at the network interface. In our setup, whenever a packet is sent to a destination, we use flow rules to steer the packet through the appropriate middlebox. Note that flow rules do not necessarily have to use MAC addresses to steer packets. For instance, when a packet destined for a server arrives at the NIC (Network Interface

Card) of a node running a middlebox script, since the destination address of the packet does not match that of the middlebox, the middlebox is supposed to drop it. However, that is not what we want. We want the middlebox to be able to read the packet, process it and possibly forward it. To achieve this, one of the fundamental and low level ways to do this is to set the interface to "*promiscuous*" mode. When an interface is set to promiscuous node, it reads all packets arriving on its interface [32]. When the interface is set into promiscuous mode, the socket library API (available in major programming languages like C/C++, Java and Python) can then be passed parameters to enable the socket read raw ethernet frames from the NIC. Another way to achieve this is to use the libcap library available in C or alternative libcap wrappers available in other major languages. Since our development is done in Python, which typically aids fast prototyping and testing, we used a robust higher level open source Python library known as Scapy [33]. Internally, Scapy wraps the low level Python socket APIs to give a more robust high level API, and has the ability to set a port into promiscuous mode ready for sniffing. Thus, we used Scapy for capturing, manipulating and forwarding raw packets in each Mininet host.

We note that the entirety of our setup was implemented in a virtual environment on a private machine. The process of reading frames using any of the methods described above involves routing packets from kernel space to user space, which takes CPU cycles. Moreover, Scapy itself takes some time to process and parse packets, however, these concerns do not affect the outcome of our results since we evaluate the relative performance of two technologies (OPPS and the redundant model) implemented using the same library and the performance of the implemented algorithms (which are independent of the library and the kernel to user space routing of packets).

When forwarding packets out of a specific interface, Scapy tends to create a new socket for each packet sent out of the interface. This creates unnecessary and unpredictable delays. To overcome this, we create a single raw socket that can be used to push out packets from the OPPS module to the switch.

To connect a middlebox to a switch, we use two interfaces on the switch and on the middlebox. The reason is that since a packet sniffer (Scapy program) sits on one interface, sniffing all packets crossing the interface, relaying the "sniffs" to the middlebox program, pushing out the same packets to the connected switch through the same interface creates a loop in the sniffer. The end result is a continuous sending of the same packet. To overcome this, we separate the incoming interface on the middlebox with the outgoing interface on the middlebox. Thus, the sniffer sniffs packets on one interface, designated as the incoming interface, while the packets are sent out through the second interface, i.e., the sending interface.

Scapy also provides an interface for constructing custom headers and APIs for manipulating packets. Hence, its robust, self-explanatory documentation and self-contained APIs makes it a good choice in our implementation.

5.1.3 Preparing Nodes for End to End Connectivity with OPPS

The typical size of MTU (Maximum Transmission Unit) over ethernet is 1500 bytes. This means that the maximum allowable frame over all the switches on our network in Mininet has to be 1500 bytes at most. However, when generating packets at the hosts, there are tendencies to generate frames whose size is equal to the size of the MTU, especially when doing bandwidth testing with the Linux Iperf tool. Since hosts

can generate frames with size equal to the MTU, if proactive measures are not taken, unwanted effects could result at the first network function that attempts to attach the extra middlebox header.

Recall that in our setup described in the previous subsection, we introduced Scapy to help sniff raw frames and also inject raw frames back into the network. Thus, when we attach the custom header of 8-byte size and attempt to inject the raw frame back into the network, the frame would be clearly larger than the MTU size of the data link layer, resulting in unwanted effects, which typically include breaking the service function program running in the Mininet host.

Is fragmentation an option? What if we manually fragment this packet and attach the header to each fragment before resending. Clearly, this option would warrant unnecessary complexities like recalculation of fields in the original packet header. If we all allow the kernel to take care of fragmentation, then we will lose the option of having the header on the two fragments, so this approach is also not ideal.

To resolve this matter, we lower the MTU size on the sending host to 512 bytes while every other node's virtual NIC keeps an MTU size of 1500 bytes. This will allow the packet to freely pass through the network even when the middlebox header is attached. In production environments, a more appropriate alternative would be to set devices within the controlled environment (e.g., data center) to allow jumbo frames since access to the host or client may be futile.

5.1.4 Summary

We used the POX [34] controller to communicate with the network switches. The controller interacts with extra modules to formulate appropriate rules that are installed in the switches. The OPPS module, which contains these algorithms (written in Python), acts on the packet when it is in the outgoing interface of the middlebox, as shown in Fig. 11. In our work, each middlebox is extended with the OPPS module at the outgoing interface. Fig. 26 shows a sample work flow setup in Mininet.

5.2 On Emulator Evaluation

In addition to the capabilities of Mininet discussed in section 5.1.1, Mininet has the ability to support large topologies with over 1000 hosts [35]. It has been reported in [35] that Mininet uses a memory of 102 Mb for a Fat tree topology with 54 hosts and 45 switches, and 112 Mb for a Linear topology of 100 hosts and 100 switches, with approximately 1 minute setup and stop times for both topologies on a 2.4 GHz intel Core 2 Duo/6GB MacBook Pro. In our laptop, 2 Gb memory and 1 processor with an execution cap of 100% was allocated to the Mininet VM in Virtual Box. The Virtual Box process uses an average of 70 Mb memory and 13% of the CPU processing power of the laptop when Mininet VM is running. On observing child processes of the Mininet emulator while executing a "*pingall*" command on a linear topology of 100 hosts and 100 switches, we note that the controller uses an average of 5% CPU processing power of the Mininet VM, while the host and switch processes takes less than 0.8% of the allocated memory and 0.3% CPU power of the Mininet VM. However, we note that a process associated with the switch

("*ovs-vswitchd unix:/var/run/openvswitch/db.sock*") takes an average of 1.9% of the allocated memory and an average of 45% of the CPU processing processing power of the Mininet VM. We also note that the effect of other background processes could impact the performance of the emulator, hence, we try to minimize this impact by closing other user space processes (applications) when running our experiments.

One of the limitations of Mininet is the lack of a guarantee that a packet will be scheduled promptly and at the same rate by all switches [35]. Since Mininet "puts" a network in a single computer, and network components are represented as separate processes scheduled and multiplexed in time on the CPU to carry out network functions [35], there is the possibility of affecting large scale emulations that are intended to measure time sensitive parameters using simultaneously generated traffic from different hosts. For this, we generate traffic that tests the parameters of different scenarios (policy chains) separately. We note that neither, the topology and middleboxes, nor routing flow are modified during the test. Hence, the network was configured from the start to handle traffic for different scenarios, both in the OPPS and the redundant scheme.

To ensure a good working condition for our environment and the accuracy of our results, we manually checked for end-to-end connectivity, correctness of our custom scripts and the policy chains deployed, before carrying out evaluations. We performed this manual testing by testing each individual unit test on the correctness of the custom scripts as well as by injecting labelled (good and bad) traffic in our network and verifying that the outputs were the expected ones.

In summary, for the purpose of our experiment, we believe Mininet is conservative in terms of memory usage, has reasonable CPU performance, gives us the flexibility

to use real packets for emulation, has a large Open source support community, and offers a platform that allows us to create, customize, deploy and debug real code and network emulations.

5.3 Evaluation of OPPS Performance

We evaluate OPPS by measuring the average end-to-end delay for different numbers of service functions (middleboxes) and different arrangements. We extend the motivating example in Section 1.2, i.e. we assume that a service provider provides 8 middleboxes labelled A to H on a linear topology. For simplicity, the middleboxes on the topology, lettered from A to F, are ordered lexicographically.

To benchmark the performance of OPPS, we adopt the definition of optimality used in *Slick* [2] as our ideal scenario. Optimal (as used in *Slick*[2]) means that every middlebox exists at each switch such that each subscriber's policy chain can be satisfied in one pass without needing the custom header or the OPPS module operations. We refer to this ideal scenario as the *Redundant model* due to the redundancy employed in satisfying different SLAs.

5.3.1 OPPS Module Performance and Latency

To test the one direction end-to-end delay between two nodes in a network, the ideal scenario is to set and synchronize clocks on the two end nodes such that, just before a packet leaves a node, the timestamp is placed on the packet, and as soon as the packet is received by the second node, time stamp is taken at the second node. Clearly one major challenge with this approach is ensuring high accuracy in clock synchronization

on the two end nodes. An alternative approach used in estimating the end-to-end delay in the network is to use the Ping command, which uses the ICMP protocol for measurement. This approach is used because the need for maintaining high accuracy in clock synchronization between the two end nodes is eliminated. When a packet is sent, it is stamped with the current time. When it is received back by the sending node of the ping packet, the node measures the difference between the current time stamp and the time stamp when the packet was sent. Usually, this difference is split evenly and the result of this split is taken as the end-to-end delay since it is estimated that the time the packet takes to travel from point A to point B is approximately equal to the delay occurring when travelling from B back to A given fair network conditions in both directions. However, this is not always the case. For example, in our scenarios, consider a packet going from point A to point B (e.g. source to server), encountering middleboxes that take extra time to process the packet, whereas when returning back to point A the packet does not go through middleboxes. Instead, it goes straight from point B to A travelling only, through the switches. Therefore, the end-to-end delay for each direction will be different.

Thus, does measuring round trip time justify the need for end-to-end delay ? We aim to compare both the OPPS and the redundant model to find the overhead (i.e. processing cost) required to use OPPS. The end-to-end delay is a component of the round trip time and since the reason for measuring round trip time is to compare two different models, whereby, for each of the models, the round trip time can be modelled as:

$$T^r_{rtt} = T^r_{A\to B} + T^r_{B\to A} + P^r_{B\ \text{proc}}$$

$$T^o_{rtt} = T^o_{A\to B} + T^o_{B\to A} + P^o_{B\ \text{proc}}$$

where $T^x_{A\to B}$ = time to move from point A to point B using x (where x can be OPPS "o" or the redundant model "r")

$T_{B\to A}$ = time to move from point B to point A

$P_{B\ \text{proc}}$ = time taken by node at B to respond back to node at point A

T^r_{rtt} = total round trip time for the redundant model

T^o_{rtt} = total round trip time using OPPS. We assume, $T^r_{B\to A} \equiv T^o_{B\to A}$ and $P^r_{B\ \text{proc}} \equiv P^o_{B\ \text{proc}}$

The end-to-end delay for both schemes can be modelled as:

$$T_{end\text{-}to\text{-}end} = T^x_{A\to B}$$

where $T_{\text{end}-\text{to}-\text{end}}$ = actual end to end delay measurement

$T^o_{A\to B} - T^r_{A\to B}$ = processing cost

$$\text{therefore } T^r_{rtt} - T^o_{rtt} = \text{processing cost}$$

Hence under the validity of the made assumptions, the round trip time can be used as the basis of comparison instead of actual end-to-end delay.

As mentioned, to acquire the round trip time measurements, we use the ping command. For each length of policy chains in the range 2 to 8, we pick a subset of service functions and vary their arrangements to emulate different subscriber policy chains (as shown at the perpendicular axis (y-axis) of the graph displayed in Fig. 13). For each arrangement we measure the round trip time of 100 ping packets and plot the average RTT and the 95% confidence interval. In Fig. 13, we refer to a policy

chain group as a group of policy chains formed by permutations[8] of the same service functions. For example, we have 3 middleboxes, *A, B, C,* among the middleboxes in the topology. A sample policy chain $A \rightarrow B \rightarrow C$ is expressed as ABC (see Y-axis of Fig. 13). Other permutations - but not the only - of these service functions include CAB and BCA. Fig. 14 shows the impact OPPS has on the average RTT versus the number of service functions on the path, as well as how it compares with the redundant model. We expect that under normal network conditions, RTT values for policies within each policy chain group will be similar since the RTT value for each policy in the group comprises the sum of the processing cost of the middleboxes (summation is a commutative operation) and the end-to-end link delay. We assume the end-to-end link delay property is constant since all policies use the same link. We also expect that as the length of the policy chains across the policy chain groups increases for both schemes, the RTT values will also increase due to extra processing cost from policy chains with extra middleboxes.

In our results, it is observed that when a group of service functions that make up a policy chain are re-arranged to form different policy chains, they generate latencies that are comparable as shown in Fig. 13(notice the overlapping confident intervals). It is also observed that in both schemes, the average RTT increases as the number of middleboxes in the policy chain increases, in a seemingly linear fashion as shown in Fig. 14. The linearity observed in the redundant model is simply as a result of extra middlebox processing and the extra link path as the policy chain length increases. Whereas, in the OPPS scheme, the linearity is the result of the extra cost

[8]We measure the round trip time of all 6 permutations of a policy chain group containing 3 middleboxes, and observe they all have overlapping confidence intervals as expected, so we show 3 in our results. Due to the length of time in running emulations for all permutations greater than 3, for subsequent groups, we select, run emulations and show results for 3 permutations.

in the redundant model plus the cost of extra OPPS processing modules. To further explain the linearity observed in OPPS scheme, we provide a runtime analysis of our algorithm. With Algorithm 4, the processing time for the OPPS module at a middlebox is *O(k)* where k is the number of middleboxes in the policy chain. Thus, for k middlebox, the total OPPS module processing time across a policy chain will be $O(k^2)$. By optimizing Algorithm 4 as commented in line 14, the total run time for the OPPS module across the policy chain is reduced from $O(k^2)$ to $O(k)$.

In our experiments, we observe that the difference between the graph for the OPPS and the redundant scheme (see X-axis in Fig. 14) is fairly constant. This difference accounts for the extra processing time incurred by OPPS while processing packets. We note that the major difference between the redundant model and the OPPS model is the OPPS algorithm (which constitutes the OPPS module). Hence, the average time difference corresponds to the time it takes to run the the OPPS algorithm on a middlebox. This time difference does not imply a fixed value for the algorithm runtime, however we provide it as evidence of the behaviour of OPPS in our evaluation. We expect that this time difference can be improved by writing the OPPS algorithm in a compiled language.

5.3.2 Flexibility and Efficiency

Flexibility relates to the ease by which we can modify our topology, add or remove middleboxes, and still have a functional network with all SLA policies met. In our work, we do not implement a dynamic handler to observe changes as this an implementation task. One way to implement this is to have module that listens to file updates or changes and then trigger other modules to read and process the new changes.

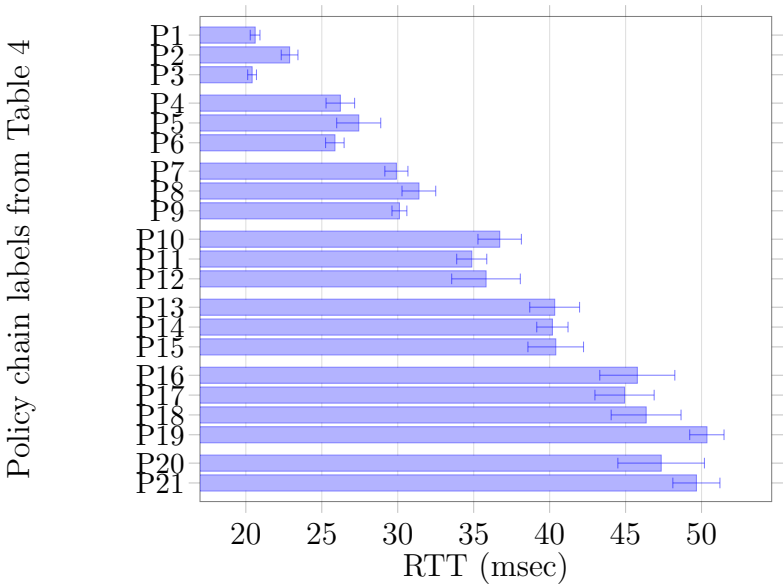

Figure 13: Latency measurements for groups of policy chains using OPPS.

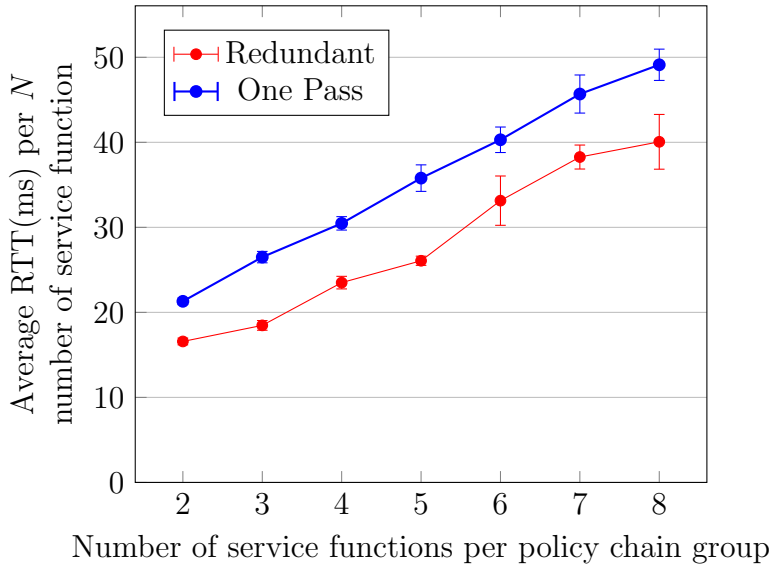

Figure 14: OPPS module performance compared to the redundant model.

Table 4: Table showing labels and corresponding middleboxes arrangement for the different policy chains in Fig. 13.

Number	Policy Chain
P 1	C D
P 2	A H
P 3	H A
P 4	C A B
P 5	A B C
P 6	B C A
P 7	F H C B
P 8	H B F C
P 9	B C H F
P 10	A F H B C
P 11	F B C A H
P 12	B C H F A
P 13	B C A E G H
P 14	H G B C E A
P 15	E B A C H G
P 16	H F C G A E B
P 17	E C A B F H G
P 18	A G F C B H E
P 19	H D F C A B E G
P 20	H A C B F H G E
P 21	B C A H F G E H

However, when a change occurs in the topology or SLA, only three components need to be updated.

- Category Database (MB Properties in Fig. 11) - This contains a list of all middleboxes on the service chain and their corresponding category. From this

list, each middlebox retrieves and sets its category in the BMNFC header as in lines 3, 7 and 8 of Algorithm 2.

- Topology Database (MSM DB in Fig. 11) - This contains an ordered, level-indented key-value pair representation of the topology, similar to the YAML serialization language [36]. The MAC addresses of switches are the keys and the masks of middleboxes are the values. A sample of the topology script is given in Fig. 15.

```
Script Description :                    |
Switch Mac - Middlebox                  |
             Bit Mask on                |      Middleboxes
             Service Chain              |
-------------------------------------------------------------
00000001 - 100                          |      FW
      00000003 - 010                    |      DLog
            00000005 - 001              |      TL
00000002 - 100                          |      FW
      00000004 - 010                    |      TL
            00000006 - 001              |      DLog
```

Figure 15: Sample topology script for Fig. 18 in Chapter 6 showing configuration for source (core) to destinations (S1, S2 and S5).

- SLA configuration for affected clients - This means updating the SLA database (from Fig. 11) of subscribers that request a new service or remove an old service.

To solve the efficiency problem, which involves traversing unnecessary middleboxes, The controller reads information from the configuration databases and uses the custom modules (see fig 11) to formulate subscriber specific flow rules for packet forwarding. Hence, only switches with middleboxes contained in a subscriber's policy chain will have flows to middleboxes matching the subscriber.

5.3.3 Challenges

Here, we present possible challenges when applying OPPS in real networks.

Representation and Implementation: In our work we use 16 bits in the middlebox header to represent middleboxes in the topology. However, it is possible that some networks may have more than 16 middleboxes in their policy chains, or a single service chain in the middlebox island consists of more than 16 middleboxes. For such cases, the 16-bit header is insufficient. To address this problem, one could explore dividing the islands into contiguous sets of islands or consider increasing the size of the header fields. We also note that OPPS deployment will require changes to the implementation of middleboxes.

Violation: For a single service chain hosting groups of policy chains with three or more service functions, OPPS can cause violation of the SLA. This places a constraint on how policy chains can be grouped on service chains. An example is given in Section 6.3.1.

Placement: The problem of determining the optimal policy chain consolidation or the number of policy chain that can be grouped on to one service chain given a limited number of resources(middleboxes) involves middlebox placement optimization. This optimization problem is left as a future work.

Stateful Policy Chains: There are common middleboxes in service provider networks (for example NAT), which have the capability to affect the results of other middleboxes down the service chain. This issue is outside the scope of our current work. In our work, we only consider stateless policy chains. Further investigation can however be done using the FlowTags [6] and [17] technique.

In the next chapter, we will discuss how to overcome these challenges and adapt OPPS to Software Defined Data Centers (SDDC).

Chapter 6

Consolidating Policy Chains in SDDCs

6.1 Background

Typical data center architectures are built based on the fat tree topology [37, 3]. A simple fat tree topology is shown in Fig. 16. One advantage of data center topologies based on the fat-tree topology (also referred to as the Clos network) is the ability to leverage inexpensive commodity hardware for switching elements and the ability to find alternate paths to connect two end hosts on the topology [3].

6.2 Related Work

With extensive research ongoing in service chaining techniques [2, 9, 7, 26, 21, 27, 38, 8, 24, 28], and in data center technologies [39, 37, 3], service chaining targeted for software defined data center has received some attention [40, 41, 25]. This make

sense since service chaining techniques focus on optimising network or service function parameters across two end nodes or across several links while data center techniques tend to focus on effective management and running of data centers. However, it is well understood that solving such service chaining problems end-to-end or across links also solves fundamental service chaining challenges in data centers. Still an unanswered question in the domain of service chaining and software defined data center is; are there packet steering techniques that can leverage the advantages of the software defined data center architecture to achieve improved packet steering?

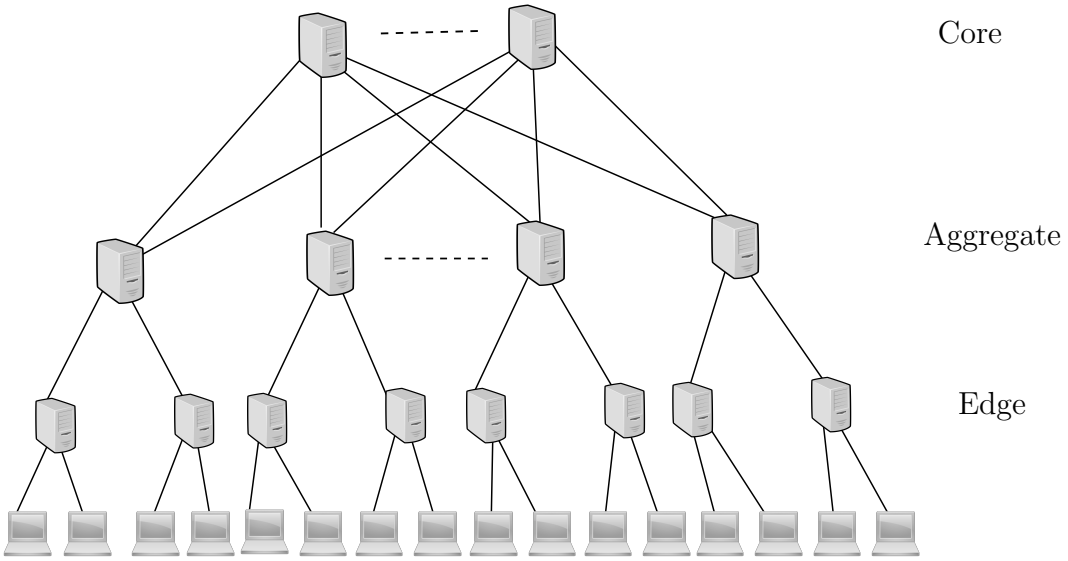

Figure 16: A simple fat tree topology (adopted from [3])

6.3 OPPS for Software Defined Data Centers

Section 5.3.3 looked at the challenges of OPPS using a simple linear topology. The motivation for choosing a linear topology is that the path between two points in a

complex or simple network can be reduced to a linear path between those two points. Although we identified challenges with the simple linear network, the question of how to mitigate these challenges is yet to be addressed. In this section, we discuss how to mitigate the challenges in a software defined data center. As mentioned in Section 6.1, a feature of data center architectures is the support for alternative routing paths. Two more features that we can utilize to address the challenges are the knowledge of the structure of the data center [37], as well as the locations of middleboxes in the topology. Specifically, we investigate how this knowledge of the controlled environment of an SDDC can be utilized to mitigate the challenges described in Section 5.3.3.

6.3.1 On SLA Violation

Let us consider a scenario where OPPS fails and examine how OPPS can leverage the nature of SDDC environments. We have a service chain comprising of 3 middleboxes (Fig. 17): a firewall (FW), a deep packet inspector with logging capability ($DLog$) and a traffic logger (TL). Assume that we have five policy chains, labelled as scenarios in Table 5, to be deployed on this service chain. By examining the policy chains, we see that with OPPS, scenario 1 is not compatible with the service chain, however scenarios 2 through 5 can be deployed using OPPS.

The reason is that, on examining the policy chain in scenario one, the TL is expected to log all flows to the destination before reaching FW. The FW is expected to drop the unwanted flows and the $DLog$ is expected to inspect, log good flows and drop malicious flows. Hence, the $DLog$ is only expected to see flows that made it through the FW. However, with OPPS deployed on the service chain topology in

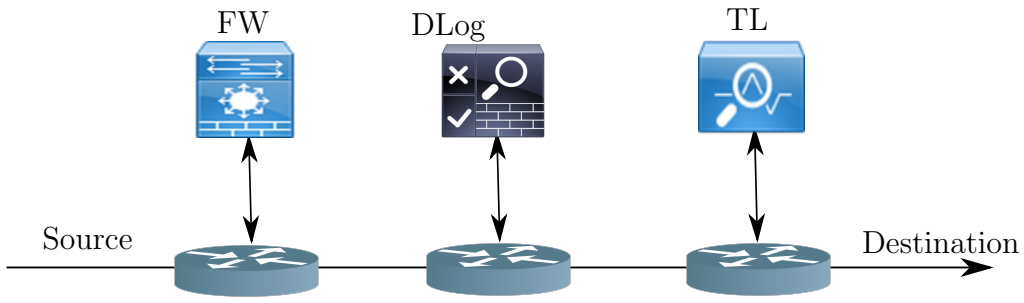

Figure 17: A typical single service chain that violates full consolidation of the policy chains listed in table 5.

Scenarios	Policy Chain	Redundant	OPPS Hops	Destination (in Fig. 18)
1	TL → FW → DLog	3	3	S5
2	DLog → FW → TL	3	3	S4
3	TL → FW	2	2	S3
4	FW → TL	2	2	S2
5	DLog → FW	2	2	S1
	Total Number of Middleboxes	12	5	

Table 5: Table showing summary of minimum number of hops and switches used in achieving the service level agreement for different policy chains.

Fig. 17, packets would have to traverse the firewall first. The *FW* marks unwanted packets for drop but does not drop the packets because, on the policy chain, it needs to visit the *TL* before any drop can be made. Hence the packet is forwarded to the next service function on the service chain, which also exists on the policy chain. This service function from Fig. 17 is the *DLog* option. The *DLog* inspects, logs and marks packets for dropping but, a drop is also not made because in the policy chain, the *TL* needs to be visited. Finally, the packet is forwarded to the *TL* which logs the packets and then the OPPS module drops unwanted traffic. Clearly, this violates the intent of the policy chain in scenario one, and the net result is that the *DLog* logs both, wanted and unwanted traffic as well as the *TL*.

To overcome this limitation of OPPS in a data center, one can leverage the advantages of data centers described at the beginning of this section. Thus, we can place middleboxes in such a way as to minimize the number of groups that can be formed from consolidating policy chains and still maintain the SLAs. Then, group compatible policy chains together and deploy them on a single service chain. In the example considered, this would imply consolidating policy chains two to five on one route and deploying policy one on another route. The net result would look as shown in Fig. 18, which shows our data center topology with partial consolidation of policy chains in Table 5. From Table 5, all policy chains in scenarios two to five will be routed on the path coloured green, while the policy chain in scenario 1 will be routed along the path coloured orange.

6.3.2 On Middlebox Representation and Implementation

In Chapter 5, section 5.3.3, we identified a solution for the middlebox representation problem. As for implementation of OPPS, we accept that middleboxes would need to be modified. Middleboxes need to be extended on the outgoing interface with the OPPS module (as in Fig. 11) to support OPPS. This extension would also require to modify how some middleboxes carry out actions on packets as described in Section 3.

6.4 Setup for SDDC Evaluation

We use Mininet to emulate a Software Defined Data Center topology. To evaluate our adoption of OPPS in Software Defined Data Centers, we create our own testbed in Mininet to support service functions. Our SDDC network contains two

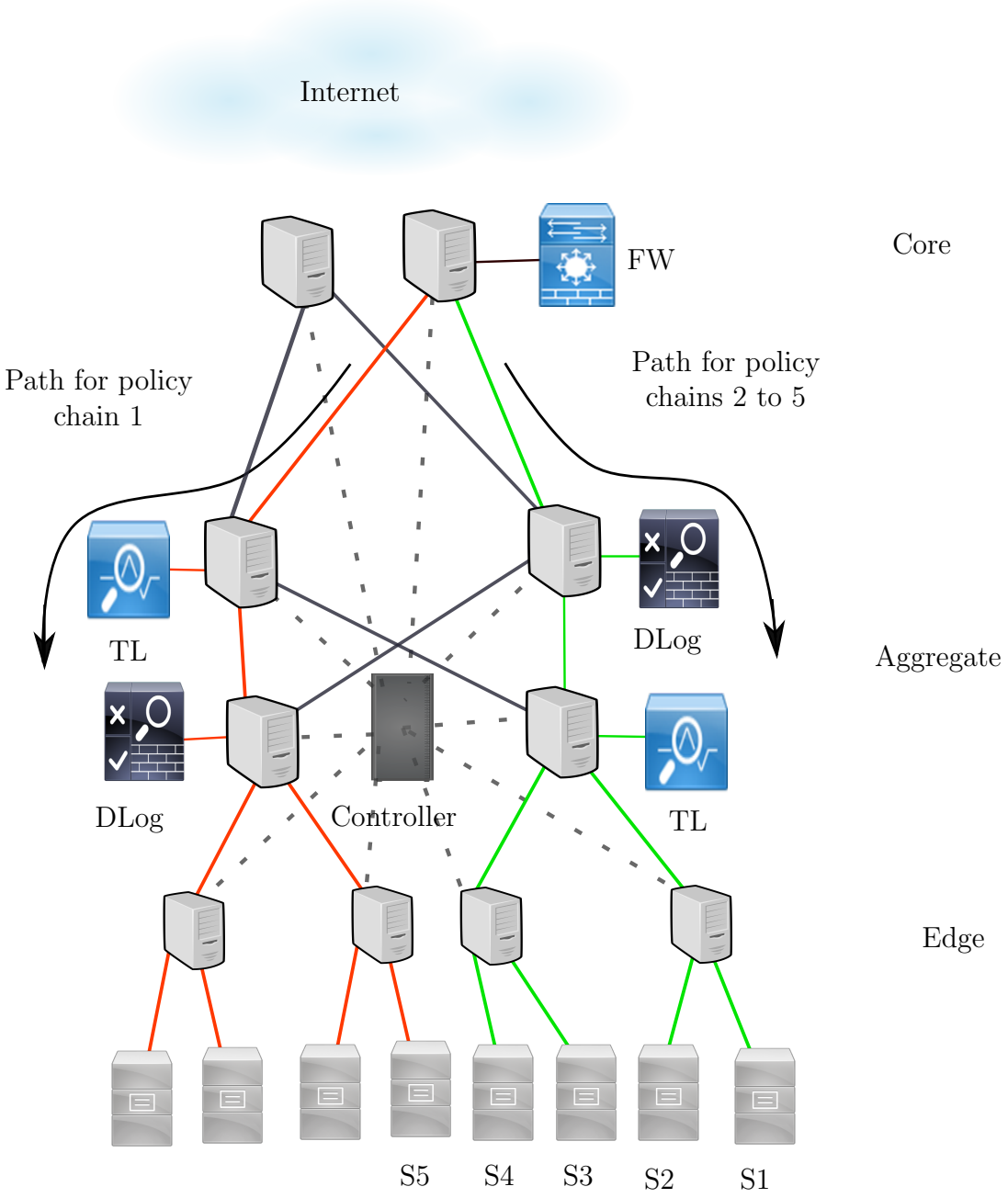

Figure 18: Overcoming limitation of OPPS in data centers.

core switches, four aggregate switches, and eight edge switches, as shown in Fig. 18. The core switches and edge switches contain scripts that add and remove the custom

header before and after middlebox operations have been performed on the packet, respectively. All links were configured as bi-directional links with 1 Gbps bandwidth capacity, 0 seconds link's (signal) propagation delay, 0 packet loss and 1000 packet queue size buffer. In implementation, we route raw packets from the kernel space to user space, to enable processing by our custom middlebox scripts. We emulate our DLog as a CPU intensive operation with an allocated fraction of the CPU at 0.6. the *FW* and the *TL* were allocated 0.3 and 0.1 respectively. We note that due to our OPPS module implementation, which involves routing packets from kernel to user space and by using the Scapy library, our measured parameters such as throughput appear lower than the set link characteristics. This is expected and it does not undermine the validity of our experiment since we use the same environment when comparing the two different techniques (OPPS and redundant scenarios). We emulate the redundant environment in a manner similar to the OPPS environment except that no OPPS modules were used for the redundant scheme. We expect that all the algorithms performed in user space will be included as kernel modules in real environments.

In real environments, users are often connected to core switches through edge switches and traffic statistics generated by aggregate users differ from statistics generated by a single individual. However, for our emulation and evaluations, nodes (hosts) acting as clients from the Internet were connected to the core switches, and nodes (hosts) acting as servers were connected to the edge switches. Each server has a specific policy chain that packets must traverse as specified in Table 5. Middleboxes built with custom Python scripts were attached as nodes to switches as shown in Fig. 18.

Flow rules were proactively installed in the switches on startup (the *Connection_Up* event in POX [34]) while the ARP traffic was resolved reactively (the *Packet_In* event in POX). The reason for installing the flow rules in a proactive way is that, before packets traverse the network, policy chains must already be known and confirmed. Also installation of the flow rules in a proactive way will help save flow rule formulation time.

It is assumed that policy chains do not change while the network is functional. The problem of handling a change in policy chain while the network is running is not within the scope of the current work.

Broadcast storms are characteristics of a typical data center network. In software defined data centers, a possible way broadcast storms can occur is during ARP resolution at the switches. In section 6.4.1, we describe in detail how we resolve this challenge.

6.4.1 Handling Layer 2 Traffic in Data Center Environmens

Forwarding Packets from Intelligent Devices

By convention, when a packet is being sent out of an intelligent device (i.e. a device capable of running its routing protocol, which determines the next hop to forward a packet; e.g. layer 3 switches and routers) reaches an interface for the first time, the node checks the network domain of the IP address in the routing table and determines if the packet should be forwarded to the gateway or another internal interface. If it needs to be forwarded to the gateway, then the packet is passed down to layer 2 (ethernet NIC), which encapsulates the packet in a single frame. One of the frame's header fields is the destination's MAC address. If the destination MAC address does

not exist on the device's internal ARP table, the device sends an ARP broadcast with *FF:FF:FF:FF:FF:FF* in the frame's destination address field, which is received by all devices connected to the layer 2 subnet.

The Layer 2 Problem

In an SDN data center network, assuming that the device wishing to acquire the MAC address of the gateway is a server directly connected to a switch (an Openflow capable switch for SDN environment), the server would have to broadcast an ARP packet to the switch. The switch will try to match the ARP packet to a flow rule, and since it is the first packet, a flow miss will occur and the packet will be sent to the controller.

With the default *l2_learning* that comes with the POX controller, which is not configured by default to reply to ARP replies, the in-port of the ARP packet and the MAC address will be learnt and stored in the *port-to-mac address* table in the controller. However, since the controller does not have the *mac-address to port* mapping of the destination IP address, the controller responds with an openflow message that directs the switch to flood the packet out to all switch ports except the in-port. The intent of the openflow message is to get the *mac-address to port* mapping of the destination IP-address at the corresponding switch. When a flow matching the MAC address to port mapping of the switch has been found and installed, the switch can then send out the ARP request through the appropriate interface. This cycle of flooding, resolving the port to address mapping, and forwarding of the ARP request packet, happens with every switch on the network that receives any flooded request. The end result is that there is too much broadcast traffic and too many openflow

buffer id errors in the network.

The Layer 2 Solution

To solve this problem, *Portland* [37] proposed a scalable and fault tolerant layer 2 routing and forwarding scheme for data center networks. The idea behind the scheme is to enable easy insertion and removal of switches in the network. The intuition applied in the scheme is to bridge the switches directly connected to end nodes with a fabric manager. The fabric manager acts as a special layer 2 routing and forwarding protocol. Thus, switches in the aggregate layer do not participate in forwarding layer 2 traffic and this helps achieve the desired plug and play effect for switches. We apply this idea by implementing a *mini-Portland* to take care of layer 2 traffic in the data center topology implemented in OPPS. In *Portland* [37], the edge switches were also implemented to re-write MAC addresses, thereby achieving a loosely-coupled network that is scalable. Implementing a full *Portland* is beyond the scope of this work however, we do acknowledge the idea borrowed to take care of the layer 2 traffic is applicable to our scheme.

6.5 Evaluation for OPPS Adoption in SDDC

6.5.1 Latency, Jitter and Throughput

To evaluate the redundant model and the OPPS model, we use the ping command to measure round trip time (delay) and jitter (the packet inter-arrival time), and the iperf command to measure throughput. To measure round trip time, we send 300 ICMP echo messages to the server at 1 second intervals. For the jitter, we calculate

the packet inter-arrival time from echo responses. To measure the throughput, we use the iperf TCP command with a default window size of 85.3 Kb to generate traffic for a period of 1200 seconds. We note that the iperf TCP command measure achievable TCP throughput in a link hence, it automatically adjusts the packet sent per second based on the bottleneck bandwidth and the round trip time of packet, therefore the traffic volume generated for each scenario varied for each 20-second interval. For all scenarios, we repeat our measurement process five times and plot all corresponding graphs using the average mean of our results and display the 95% confidence interval. Fig. 19 shows the average round trip time of packets traversing the topology shown in Fig. 18 from the source (internet) to the destination (any of the five servers) with the policy chain requirements listed in Table 5. Similar to Fig. 13, we see that policy chains with an equivalent composition of middleboxes have similar round trip times for each scheme considered individually as explained in Section 5.3.1. From Fig. 20, we conclude that for all 5 scenarios, the jitter composition of OPPS is comparable to that of the redundant model. It can be seen that the 95% confidence interval of the curves overlap for most of the points. In Fig. 21, the redundant model throughput is higher than that of OPPS. This is expected since the redundant model is considered as our ideal scenario (it has all middleboxes installed at every switch). Similar to the round trip time, scenarios with similar and equivalent middleboxes in the policy chain get the same throughput treatment with each scheme considered individually.

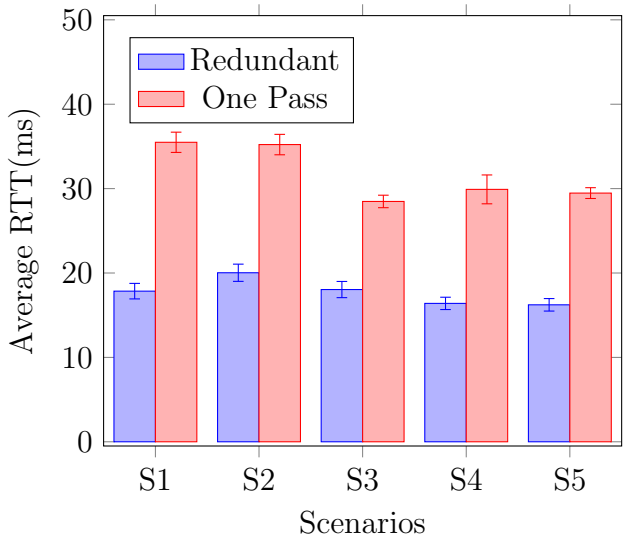

Figure 19: Average RTT for the OPPS and the Redundant schemes, with a 16-bit header when packet destination is Server Si $(1 \leq i \leq 5)$.

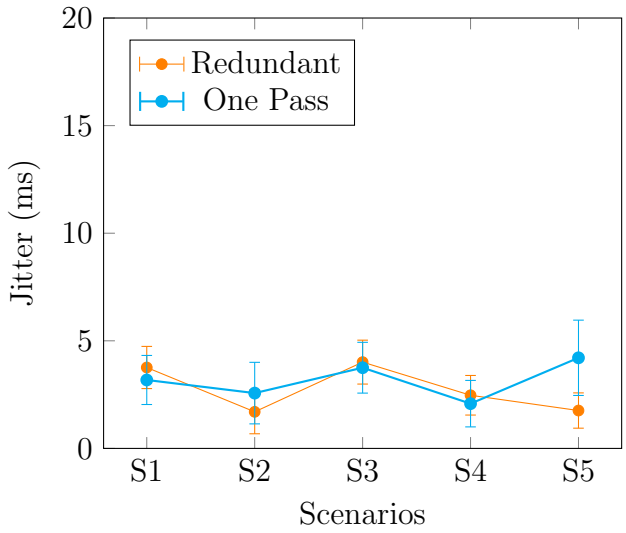

Figure 20: Packet jitter experienced by RTT corresponding to the OPPS and Redundant models with a 16-bit header scheme when the packet destination is Server Si $(1 \leq i \leq 5)$.

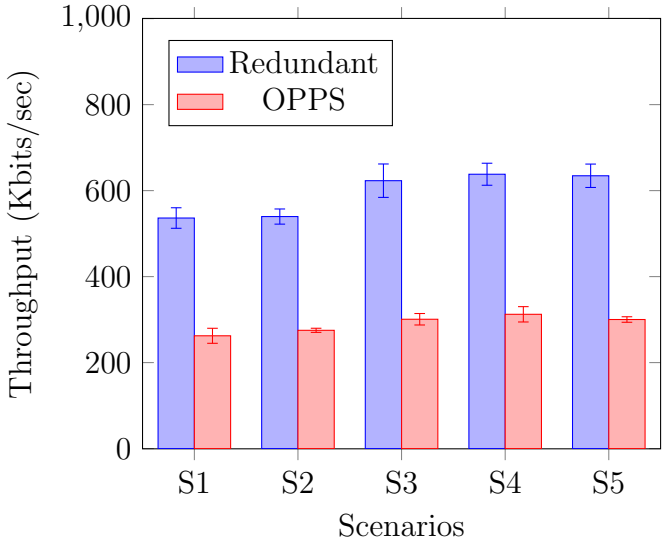

Figure 21: Throughput comparison between OPPS and the Redundant model.

6.5.2 Impact of OPPS

OPPS achieves its goal, which is to treat all subscribers using a similar and equal number of service functions equally, irrespective of the order in which the service functions may appear on the topology. However, to achieve this aim, OPPS introduces trade-offs, which include:

- addition of a field label (8 bytes long for our experiment)
- addition of an OPPS module to each middlebox.

Intuitively, adding a field label will affect throughput, as seen in Fig. 21, where the throughput of OPPS is not as high as the throughput of the redundant model. However, the added field sub-layer is still open for improvement and shortening. For example, if all subscribers in a data center had a maximum of four middleboxes in each of their policy chain, then the policy chains can be grouped such that a service chain would only comprise at most four middleboxes. In this case, it would make

sense to reduce the size of the middlebox field label. This will in turn improve the throughput. Fig. 22 shows a comparison between the round trip time of a scaled down version of the header and the original version as shown in Fig. 12. In the scaled down version, the BMNFC, BMNFA and the BMNFV are shortened from 16 bits each to 8 bits. Although no significant difference is observed in the round trip time due to the overlapping confidence intervals, the throughput (as shown in Fig. 23) shows some improvement when these fields are shortened to 8 bit for scenarios S1, S2, S3, S4, S5. Another improvement in size can also come from computing some fields such as the counter (NFC field) from the visited field (BMNFV) instead of appending them in the header label. However this would be a trade-off between throughput and delay, since computing the counter would incur its own delay.

The OPPS processing sub-layer added to each middlebox (Fig. 11) accounts for the majority of the difference in the delay between the OPPS model and the redundant model. The OPPS module and scripts have been implemented in an interpreted language (Python) in our experiment and hence, offers more opportunity for runtime improvement when implemented in a compiled language. Although the OPPS model takes more computing time due to processing, Fig. 20 and Fig. 24 tell us that the impact of the OPPS module on the packet jitter is comparable to that of the redundant model, hence we can conclude that OPPS has low jitter impact on the network.

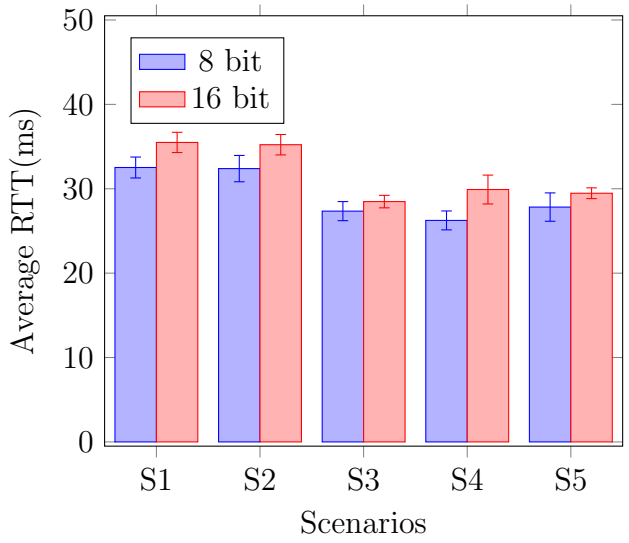

Figure 22: Average round trip time in OPPS using 8-bit header versus 16-bit header.

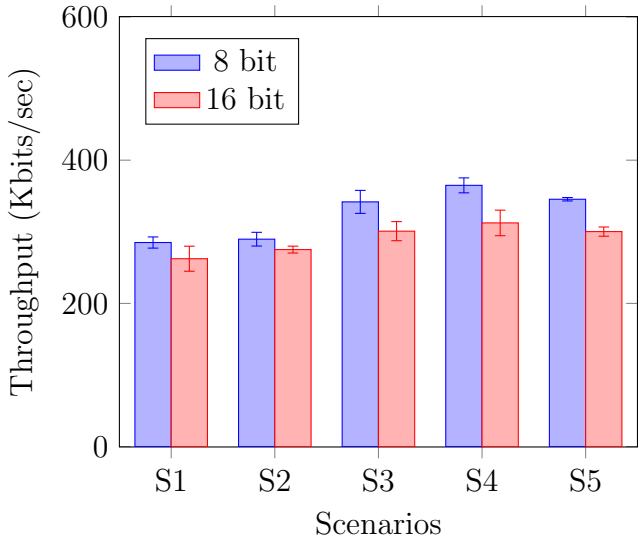

Figure 23: Throughput comparison in OPPS using 8-bit header and 16-bit version.

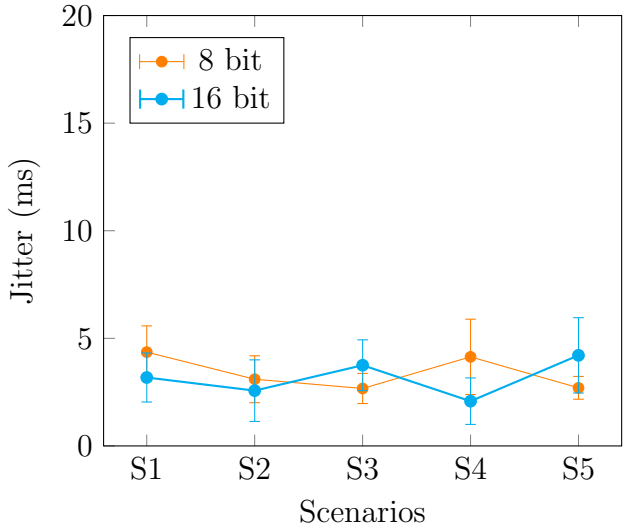

Figure 24: Jitter in OPPS using 8-bit header versus 16 bit header.

6.5.3 Benefits of OPPS

From the previous section, we see how OPPS can be further improved by taking into consideration the specifics on the network and application deployment. In this section, we will discuss whether OPPS is suitable for Software Defined Data Centers.

Table 5 provides a comparison between the number of middleboxes needed to keep the service level agreement while maintaining a minimum number of hops when using the Redundant model, and when using the OPPS. From the last row of table 5, we see that OPPS significantly reduces the total number of middleboxes deployed (by 58%) while still maintaining the same number of hops as in the Redundant scenarios. This is because OPPS requires only 5 middlebox deployments (as shown in Fig 18) to fulfil the SLAs of S1 through S5. Similarly, in terms of round trip time and throughput, shown in Figs. 19 and 21 respectively, OPPS and the redundant model have similar patterns. We note that the Redundant model performs better than OPPS in throughput and round trip time as explained in Section 5.3.1. From Fig.

20, the jitter incurred in OPPS is also comparable to the jitter experienced by the redundant model. This is one of the aims of OPPS i.e., to have little or no jitter such that its impact on the network does not degrade network performance.

6.5.4 Discussion

As shown from Fig. 18, we see that OPPS can leverage the data center architecture to bring about improved end to end delay and good savings on network resources. As discussed in section 6.5.2, OPPS can be fine-tuned depending on usage, however the general intent is that with OPPS, different subscribers can share the same resources and still get equal amount of access to resources, while OPPS can utilize a lesser number of middleboxes and achieve the same hop counts as a reference model, which has been described in previous work as ideal, without violating the subscribers policy chain. As the technique of OPPS is still a new concept, investigations on how to support varying middleboxes remains future research. We believe a major factor that has helped the integration of extra protocols like VLAN and MPLS to the TCP/IP stack is the conformation of layer 2 and layer 3 network devices to a standard irrespective of vendor implementation. It is this drive for conformation that has allowed interoperability of layer 2 and layer 3 devices across vendors. Such standards or models do not currently exist for service functions, thus this will inhibit the adoption of inventions relating to service functions. Moreover, the rapid shift to software implementations and virtualization of network functions - evident in Arjun et. al [39] with the use of "*Fbflow*" and load balancers - lack of a set out model or APIs for service functions hinders the progress to create smarter service function chaining concepts. This is because, with a shift to NFV or software service functions

without a set standard, innovations related to service function chaining may have to be heavily dependent on the operation of unique middleboxes. However, we envisage that service chaining concepts like OPPS and Flowtags [6] could be readily adopted by organizations operating large data centers that build, deploy and rely on their own internal infrastructure stack.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

This thesis introduces a new method for provisioning policy chains leveraging Software Defined Networking. By building a service-function-aware framework and introducing algorithms for middleboxes within the TCP/IP stack, we are able to enforce one pass packet steering for a set of policy chains consolidated all in one chain.

In this work, we identify two service function categories, a category of service functions that could drop packets (referred to with a category bit of "1" or simply category "1") along the service chain and a category of service functions that have no tendency to stop the forwarding of a packet (referred to with a category bit of "0" or category "0"). For this, we develop a packet steering technique that works when the following cases holds.

A The policy chain conforms to the order of middleboxes on the service chain.

B The service functions on the policy chain are of category "0"s only, and middleboxes on the service chain are arranged in any sequence.

C The service functions on the policy chain are of category "1"s and middleboxes on the service chain follow the order in which the service functions appear on the policy chain [9].

D Given that case C is maintained and the service functions on the policy chain are a combination of category "0" and category "1" middleboxes. On the policy chain, if there are category "0" middleboxes before a category "1" middlebox, then the category "1" middlebox may come before these category "0" (the reverse cannot happen) middleboxes on the service chain. When this occurs, middleboxes that should come after the category "1" middlebox on the service chain must be middleboxes of category "0" that appeared before the category "1" middlebox on the policy chain. If middleboxes still exist on the policy chain after the category "1" middlebox, then case A, B, C or D may be re-applied. Thus, when a packet must continue traversing the service chain after visiting a category "1" middlebox, it should visit middleboxes that come before the category "1" middlebox on the policy chain before visiting other middleboxes.

For policy chains that fit in to the service chains of cases A, B and C, we note that packets can be routed in one pass without OPPS and still maintain the SLA. For Case D which can occur in multi-subscriber environments where subscribers may have different middlebox arrangements in their policy chain, OPPS is required for

[9]We note that in virtualized environment, when a middlebox of category bit "1" and is consolidated with a middlebox of category bit "0", the outcome is a category "1" middlebox. For example, a Dlog as used in the example in section 6.3.1.

packets to be routed in one pass. Hence, OPPS works for linear stateless policy chains deployed on service chains that conforms to any of the cases above.

We also examined the challenges of OPPS and introduced practical ways to overcome these challenges, more specifically in SDDC networks. We carried out evaluations of our new framework using proof of concept emulation built on Mininet and showed with a fixed topology and different sets of policy chains containing the same middleboxes, how the end-to-end delay and throughput performance of subscribers using similar policy chains remain approximately the same. Also, we showed in our results, that our OPPS module with *O(k)* processing time across a policy chain adds low jitter to the network, can utilize lesser number of middleboxes and it achieves the same hop counts as a reference model, which has been described in previous work as ideal, without violating the subscribers policy chain.

7.2 Future Work

The act of consolidating policy chains into a single service chain and enabling a one pass technique is a new idea and could benefit from additional refinement. Ideas for improvement include:

- Enabling stateful policy chains using One Pass Technique.
- Exploring OPPS in VNF environment.
- Developing algorithms or strategies for determining optimal policy chain consolidation.
- Enhancement of the field labels to support unique tag identifiers and possible header compression techniques.

- Service chaining operations and standard routing operations are not mutually exclusive, hence, there is need for a discussion on adopting a standard for service chain headers that takes into consideration a general model, middleboxes must follow.

Appendices

Appendix A

OPPS Algorithms

Algorithm 1: Handles all packets from the middlebox

```
1  /* "Packet" is the packet processed and passed down from the
   middlebox                                                  */;
2  /* "action" is the result(drop or forward) from the middlebox */;
3  /* "maskOfNF" is the identifier of the middlebox, which is also
   the middlebox bit mask                                     */;
4  Function PacketHandler(Packet, action, maskOfNF)
5  |  Packet ⟵ UpdateHandler(Packet,action,maskOfNF) ;
6  |  if isSufficientToDecide ( Packet ) then
7  |  |  decision ⟵ MakeDecision(Packet) ;
8  |  |  return ActionImplementer( decision,Packet ) ;
9  |  else
10 |  |  return Packet ;
11 |  end
```

Algorithm 2: Updates actions at NF's designated bit position.

```
1  Function UpdateHandler(Packet, action, maskOfNF)
2      /* "Packet : foo" indicates the value of field "foo" in the
       packet header                                              */;
3      category ⟵ CATEGORIES[maskOfNF] ;
4      /* "CATEGORIES" is a (key,value) pair of the middlebox
       identifier (maskOfNF) to middlebox category                */;
5      index ⟵ indexOfLeastSignificant1Bit( maskOfNF ) ;
6      action ⟵ action ≪ index ;
7      category⟵ category ≪ index ;
8      Packet:category ⟵ Packet:category | category;
9      Packet:visited ⟵ maskOfNF ⊕ Packet:visited ;
10     Packet:action ⟵ action | Packet:action ;
11     return Packet ;
```

Algorithm 3: Checks if part or all of SLA have been met

```
1  Function IsSufficientToDecide(Packet)
2      visited ⟵ Packet : visited ;
3      counter ⟵ Packet : counter;
4      destIP ⟵ Packet : destination;
5      orderedListOfNFs ⟵ SlaPolicies[destIP] ;
6      /* "SlaPolicies" is a (key,value) pair of subscribers
       (identified by the destination IP) to policy chain         */;
7      /* "orderedListOfNFs" is a list containing a bit mask of
       middleboxes in the policy chain in order                   */;
8      mask ⟵ orderedListOfNFs[counter] ;
9      visited ⟵ visited ⊕ mask ;
10     Packet : visited ⟵ visited ;
11     Packet : counter ⟵ counter + 1 ;
12     /* Returns true if sufficient to decide else False        */
13      if visited > 0 then
14         return False;
15     else
16         return True ;
17     end
18
```

Algorithm 4: Makes decision by examining category and action bits

```
1  Function MakeDecision(Packet)
2      category ⟵ Packet:category ;
3      visited ⟵ Packet:visited ;
4      action ⟵ Packet:action ;
5      while category do
6          index ⟵ indexOfLeastSignificant1Bit( category );
7          category ⟵ category ≫ index;
8          action ⟵ action ≫ index;
9          /* Returns Drop if category is 1 and action is 1          */
10         if action then
11             return DROP;
12         end
13     end
14     /* Optimize by setting already examined category bits to 0  */;
15     return FORWARD ;
```

Algorithm 5: Drops or forwards a packet.

```
1  Function ActionImplementer(decision, Packet)
2      if decision is FORWARD then
3          return Packet ;
4      else
5          return NULL ;
6      end
```

Appendix B

Mininet Setup and Configuration

```
# HOST->SERVER

ACTIVE
in_port=1
dl_type=0x0800
actions=2

DEFAULT
in_port=1
dl_type=0x800
actions=3

# MBX->SW

ACTIVE
in_port=4
dl_type=0x0800
actions=3

# SERVER->HOST

DEFAULT
in_port=3
dl_type=0x0800
actions=1
```

Figure 25: Sample RCM Config File Format.

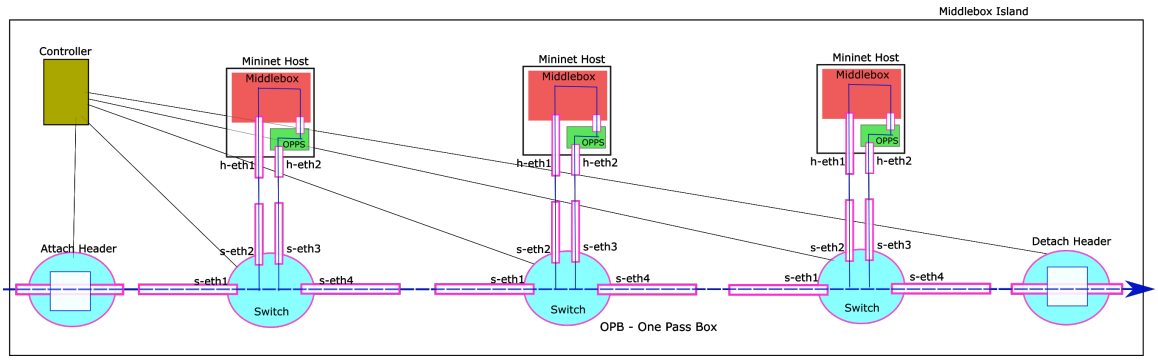

Figure 26: Sample setup of packet flow for all subscribers

Bibliography

[1] G. Gibb, H. Zeng, and N. McKeown, "Initial thoughts on custom network processing via waypoint services," in *Proceedings of the WISH-3rd Workshop on Infrastructures for Software/Hardware co-design, CGO*, Chamonix, France, 2011.

[2] B. Anwer, T. Benson, N. Feamster, and D. Levin, "Programming slick network functions," in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*. Santa Clara, California: ACM, 2015, pp. 14:1–14:13.

[3] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 4, pp. 63–74, Aug. 2008.

[4] P. Quinn and T. Nadeau, "Problem Statement for Service Function Chaining," Internet Requests for Comments, RFC Editor, RFC 7498, April 2015.

[5] D. A. Joseph, A. Tavakoli, and I. Stoica, "A policy-aware switching layer for data centers," in *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*. Seattle, WA, USA: ACM, 2008, pp. 51–62.

[6] S. K. Fayazbakhsh, V. Sekar, M. Yu, and J. C. Mogul, "Flowtags: Enforcing network-wide policies in the presence of dynamic middlebox actions," in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*. Hong Kong, China: ACM, 2013, pp. 19–24.

[7] Y. Zhang, N. Beheshti, L. Beliveau, G. Lefebvre, R. Manghirmalani, R. Mishra, R. Patneyt, M. Shirazipour, R. Subrahmaniam, C. Truchan, and M. Tatipamula, "Steering: A software-defined networking for inline service chaining," in *Proceedings of the 2013 21st IEEE International Conference on Network Protocols (ICNP)*, Goettingen, Germany, Oct 2013, pp. 1–10.

[8] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu, and J. C. Mogul, "Enforcing network-wide policies in the presence of dynamic middlebox actions using flowtags," in *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association, 2014, pp. 543–546.

[9] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu, "Simple-fying middlebox policy enforcement using sdn," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*. Hong Kong, China: ACM, 2013, pp. 27–38.

[10] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi, "Design and implementation of a consolidated middlebox architecture," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2012, pp. 24–24.

[11] R. Potharaju and N. Jain, "Demystifying the dark side of the middle: A field study of middlebox failures in datacenters," in *Proceedings of the 2013 Conference on Internet Measurement Conference*. Barcelona, Spain: ACM, 2013, pp. 9–22.

[12] J. Chukwu, A. Matrawy, and D. Makrakis, "One pass packet steering (OPPS) for stateless policy chains in multi-subscriber SDN," in *Proceedings of the 2017 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS): SWFAN 17: International Workshop on Software-Driven Flexible and Agile Networking (INFOCOM17 WKSHPS SWFAN'17)*, Atlanta, USA, May 2017.

[13] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar, "Making middleboxes someone else's problem: Network processing as a cloud service," in *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. Helsinki, Finland: ACM, 2012, pp. 13–24.

[14] B. Carpenter and S. Brim, "Middleboxes: Taxonomy and Issues," Internet Requests for Comments, RFC Editor, RFC 3234, February 2002.

[15] K. Edeline and B. Donnet, "Towards a middlebox policy taxonomy: Path impairments," in *Proceedings of the 2015 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, Hong Kong, China, April 2015, pp. 402–407.

[16] D. Joseph and I. Stoica, "Modeling middleboxes," *Netwrk. Mag. of Global Internetwkg.*, vol. 22, no. 5, pp. 20–25, Sep. 2008.

[17] M. Kablan, B. Caldwell, R. Han, H. Jamjoom, and E. Keller, "Stateless network functions," in *Proceedings of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*. London, United Kingdom: ACM, 2015, pp. 49–54.

[18] A. Lara, A. Kolasani, and B. Ramamurthy, "Simplifying network management using software defined networking and openflow," in *Proceedings of the 2012 IEEE International Conference on Advanced Networks and Telecommuncations Systems (ANTS)*, Dec 2012, pp. 24–29.

[19] M. C. S. Blake, D. Black, E. Davies, Z. Wang, and W. Weiss, "An Architecture for Differentiated Services ," http://www.rfc-editor.org/rfc/rfc2475.txt, RFC Editor, RFC 2475, December 1998.

[20] P. Quinn and J. Guichard, "Service function chaining: Creating a service plane via network service headers," *Computer*, vol. 47, no. 11, pp. 38–44, Nov 2014.

[21] J. Liu, Y. Li, Y. Zhang, L. Su, and D. Jin, "Improve service chaining performance with optimized middlebox placement," *IEEE Transactions on Services Computing*, Nov 2015.

[22] A. Bremler-Barr, Y. Harchol, and D. Hay, "Openbox: Enabling innovation in middlebox applications," in *Proceedings of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*. London, United Kingdom: ACM, 2015, pp. 67–72.

[23] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "Clickos and the art of network function virtualization," in *Proceedings*

of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14). Seattle, WA: USENIX Association, Apr. 2014, pp. 459–473.

[24] Z. Cao, M. Kodialam, and T. V. Lakshman, "Traffic steering in software defined networks: Planning and online routing," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, Aug. 2014.

[25] A. Leivadeas, M. Falkner, I. Lambadaris, and G. Kesidis, "Dynamic traffic steering of multi-tenant virtualized network functions in sdn enabled data centers," in *Proceedings of the 2016 IEEE 21st International Workshop on Computer Aided Modelling and Design of Communication Links and Networks (CAMAD)*, Toronto, ON, Canada, Oct 2016, pp. 65–70.

[26] T. W. Kuo, B. H. Liou, K. C. J. Lin, and M. J. Tsai, "Deploying chains of virtual network functions: On the relation between link and server usage," in *Proceedings of the IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, San Francisco, CA, USA, April 2016, pp. 1–9.

[27] A. Mohammadkhan, S. Ghapani, G. Liu, W. Zhang, K. K. Ramakrishnan, and T. Wood, "Virtual function placement and traffic steering in flexible and dynamic software defined networks," in *the Proceedings of the 21st IEEE International Workshop on Local and Metropolitan Area Networks*, Beijing, China, April 2015, pp. 1–6.

[28] G. Lee, M. Kim, S. Choo, S. Pack, and Y. Kim, "Optimal flow distribution in service function chaining," in *Proceedings of the The 10th International Conference on Future Internet*. Seoul, Republic of Korea: ACM, 2015, pp. 17–20.

[29] S. Deering and R. Hinden., " Internet Protocol, Version 6 (IPv6) Specification," http://www.rfc-editor.org/rfc/rfc2460.txt, RFC Editor, RFC 3234, December 1998.

[30] J. Postel, "Internet Protocol," http://www.rfc-editor.org/rfc/rfc791.txt, RFC Editor, RFC 791, September 1981.

[31] "Mininet," http://mininet.org/, accessed on 10/03/2017.

[32] K. Wehrle, *The Linux Networking Architecture: Design and Implementation of Network Protocols in the Linux Kernel*. Pearson Prentice Hall, 2004.

[33] "Scapy," http://www.secdev.org/projects/scapy/, accessed on 11/01/2017.

[34] "Pox, python-based openflow controller," http://www.noxrepo.org/pox/about-pox/, accessed on 17/02/2017.

[35] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: Rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. Monterey, California: ACM, 2010, pp. 19:1–19:6.

[36] C. E. Oren Ben-Kiki and I. döt Net, "YAML Ain't Markup Language (YAML™) Version 1.2," http://yaml.org/spec/1.2/spec.html, accessed on 11/3/2017.

[37] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat, "Portland: A scalable fault-tolerant layer 2 data center network fabric," *SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 4, pp. 39–50, Aug. 2009.

[38] J. Blendin, J. Rückert, N. Leymann, G. Schyguda, and D. Hausheer, "Position paper: Software-defined network service chaining," in *Proceedings of the 2014 Third European Workshop on Software Defined Networks*, London, UK, Sept 2014, pp. 109–114.

[39] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network," *SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, pp. 123–137, Aug. 2015.

[40] S. Herker, X. An, W. Kiess, S. Beker, and A. Kirstaedter, "Data-center architecture impacts on virtualized network functions service chain embedding with high availability requirements," in *Proceedings of the 2015 IEEE Globecom Workshops (GC Wkshps)*, San Diego, CA, USA, Dec 2015, pp. 1–7.

[41] C. H. Hsieh, J. W. Chang, C. Chen, and S. H. Lu, "Network-aware service function chaining placement in a data center," in *Proceedings of the 2016 18th Asia-Pacific Network Operations and Management Symposium (APNOMS)*, Kanazawa, Japan, Oct 2016, pp. 1–6.