

# Service Oriented Packet Forwarding in SDN

Rahul Patil

A Thesis Submitted to  
Indian Institute of Technology Hyderabad  
In Partial Fulfillment of the Requirements for  
The Degree of Master of Technology



Department of Computer Science and Engineering

June 2015

## Declaration

I declare that this written submission represents my ideas in my own words, and where ideas or words of others have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be a cause for disciplinary action by the Institute and can also evoke penal action from the sources that have thus not been properly cited, or from whom proper permission has not been taken when needed.

R Patil

(Signature)

\_\_\_\_\_  
(Rahul Patil)

CS12M1008

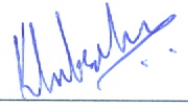
(Roll No.)

## Approval Sheet

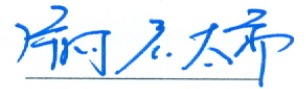
This Thesis entitled Service Oriented Packet Forwarding in SDN by Rahul Patil is approved for the degree of Master of Technology from IIT Hyderabad



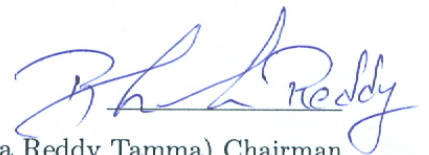
(Dr. Mohammed Zafar Ali Khan) Examiner  
Dept. of Electrical Engineering  
IITH



(.....*Internal*.....) Examiner  
Dept. of Computer Science and Engineering  
SUBRAMANYAM KAWANASUNDARAM IITH



(Dr. Kotaro Kataoka) Adviser  
Dept. of Computer Science and Engineering  
IITH



(Dr. Bheemarjuna Reddy Tamma) Chairman  
Dept. of Computer Science and Engineering  
IITH

## Acknowledgements

Foremost, I would like to express my sincere gratitude to my adviser Dr. Kotaro Kataoka for his valuable guidance, motivation, enthusiasm, and immense knowledge. Without his guidance and persistent help the completion of this dissertation would not have been possible. My sincere thanks to the entire faculty of the CSE department who inspired all the students towards Computer Science. I am also thankful to Prakash Pawar and Uttam Dhabas for their help in completing this dissertation. I would not forget to remember my colleagues, for their support, feedback and constant encouragement, without their cooperation this dissertation would not have been possible. I would like to make a special mention of DISANET project which gave me an opportunity to work on a live project and explore new areas of Computer Science.

Finally, I thank my family and friends for supporting me throughout my studies at the institute.

## Abstract

Today's enterprise networks rely on a variety of middleboxes. These middleboxes provide various improvements on services in the network such as security, management, performance and scalability. Some of the popular examples of middleboxes are Proxy, NAT, Load-Balancer, Firewall, DPI etc. Even though middleboxes offer significant benefits, they introduce new challenges such as difficulty in ensuring service chaining. Service chaining virtually inserts services into the flow of network traffic. As packets traverse the network, their headers and contents may get modified by middleboxes, that are deployed along the network path; e.g., NAT and Load-Balancer rewrites IP header, whereas proxy terminates sessions. These modifications eliminate the possibility of re-purposing available header fields such as DSCP bits in the IP header for realizing service chain.

To integrate middleboxes into SDN-capable network and leverage the benefits of both, this research proposes a novel approach using packet tagging, which addresses the challenge of enabling service chaining without modifying middleboxes. The proposed system tags packets of flows at the ingress switch, to determine the sequence of middleboxes for these packets and enforce them to follow the given service chain. Middleboxes do not need to be aware of the added tags and hence no modifications are required to the middlebox software to guarantee correct middlebox traversal. In addition the proposed solution is simple and lightweight because we do not require special techniques to detect the impact of service application on the packet. This thesis also discusses the feasibility to integrate with the existing infrastructure to support L<sub>4</sub>-L<sub>7</sub> capability. Future work includes use of 802.1ad for the scalability of the existing solution. A proof of concept of proposed system is implemented using open source protocol, OpenFlow version 1.0 and open source controller, floodlight and tested with the MiniNet network emulator by running network testing tool, iperf and tcpdump.

# Contents

Declaration . . . . .	ii
Approval Sheet . . . . .	iii
Acknowledgements . . . . .	iv
Abstract . . . . .	v
<b>Nomenclature</b>	<b>vii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Service Chaining . . . . .	1
1.2 Overview of the Work . . . . .	2
1.3 Thesis Outline . . . . .	2
<b>2 SDN and OpenFlow</b>	<b>4</b>
2.1 Software Defined Networking . . . . .	4
2.2 OpenFlow Protocol . . . . .	5
<b>3 Service Oriented Packet Forwarding in SDN</b>	<b>8</b>
<b>4 Related Work</b>	<b>12</b>
<b>5 System Design</b>	<b>14</b>
5.1 Shallow Packet Inspector . . . . .	15
5.2 Packet Tagging at Ingress & Egress Switch . . . . .	16
5.3 Flow Rule Compaction . . . . .	16
5.4 Deep Packet Inspector . . . . .	16
5.5 Policy Specification . . . . .	16
<b>6 System Implementation</b>	<b>18</b>
6.1 Data Structures . . . . .	18
6.2 Proactive Flow Rule Injection . . . . .	19
6.3 Encoding tag in 802.1Q header . . . . .	19
6.4 Network Configuration and Traffic Handling for Middleboxes . . . . .	19
6.5 Addition of a new Middlebox in the Existing Network . . . . .	21
6.6 Dynamic Policy Change . . . . .	22

<b>7</b>	<b>Evaluation</b>	<b>23</b>
7.1	Proof of Concept . . . . .	23
7.1.1	Time to Install Flow Rules . . . . .	24
7.2	Overhead : SPI and Additional 802.1Q Header . . . . .	24
7.3	Benefits of Our System . . . . .	24
7.4	Number of Flow Rules . . . . .	24
7.5	Discussion . . . . .	26
7.5.1	Double Tagging for Scalable Service Chaining . . . . .	26
7.5.2	Middlebox Placement Problem . . . . .	27
7.5.3	SPI Interface . . . . .	27
<b>8</b>	<b>Conclusions and Future Work</b>	<b>28</b>
	<b>References</b>	<b>29</b>
<b>A</b>	<b>VLAN Configuration</b>	<b>32</b>
A.1	IEEE 802.1Q . . . . .	32
A.2	IEEE 802.1ad (QinQ) [1]. . . . .	32

# Chapter 1

## Introduction

Today's networks are growing in terms of bandwidth, number of devices supported, various network access technologies and services used. Enterprise networks also have similar characteristics and these characteristics leads to real and pressing problems and the needs that are not likely to go away e.g. NAT to solve the problem of IP address space depletion, Firewall to deal with various attacks on internal network from the outside large network. So it is vital to have middleboxes in the network to address these problems. But the introduction of middleboxes cause new important and common problem, i.e. Middlebox Sequencing also known as Service Chaining. So it is natural to look for a solution which effectively solves this problem. Software-Defined Networking (SDN) is a networking architecture which has been gaining momentum in the past few years. Because of flexible and programmable feature of SDN, such technology is promising to solve the service chaining problem effectively. This thesis addresses the challenge of ensuring service chaining without modifying middleboxes in an enterprise network using SDN.

### 1.1 Service Chaining

End users are often unaware of the existence of middleboxes in their traffic's path. Middleboxes are inevitably deployed in enterprise networks, and more recently in data centers and cloud environments. While they have received a significant amount of attention in recent years [2-4], Citation there is still no satisfactory solution to the problem of directing the traffic through the desired sequence of middleboxes. It requires carefully planned network topology, manually set up rules to route traffic through the desired sequence of middleboxes, and implement safeguards for correct operation in the presence of failures and overload.

Service chaining is required when traffic needs to traverse through more than one middleboxes in a specific sequence (e.g. web traffic should be processed by a web proxy and then a firewall). If more than one sequence of middleboxes/services are possible, then the network configuration needs to be done so that the right traffic goes through right path of middlebox. This process of network configuration is complex and rigid, it often leads to the mis-configurations and errors. Lack of availability of protocols and tools to carry out this configuration makes the service chaining problem more complex. SDN offers a promising alternative for solving the service chaining problem. It uses logically centralized management, decouples the data and control plane, and provides programability



of forwarding flows.

This thesis tries to give a simple but effective and immediately deployable approach to service chaining problem by using existing technologies. In brief, we utilize VLAN ID field of 802.1Q header within each packet, which can be used for yet another purpose of forwarding packets in the network. We tag each packet at the ingress switch, this tag is used to control the forwarding of the packets in the network. Tagging decision is taken at controller with the help of Shallow Packet Inspector (SPI). Tagging avoids the need of having per flow basis rules at each switch in the network, this results, reduction in number of required flow rules. As the middleboxes may modify the contents of the upper layer packet headers we cannot preserve tag, if we use other available fields such as 6-bit DS field (part of the 8-bit ToS) at IP layer. In order to overcome this, We make use of VLAN ID field which is part of 802.1Q header to store the tag. The proposed solution is a novel approach and is easy to deploy, without much modifications to the existing network configuration. flow-unique rule at each switch in the network. Our proof-of-concept implementation can achieve service chaining with minimal changes in the network configuration and without modifying actual middleboxes. We can configure the middleboxes as per our convenience so that desired network configuration can help us achieve the service chaining. While the primary focus of this thesis is service chaining, the system evaluation also shows how it may be useful in different scenarios such as profiling and QoS control. Profiling can help to define different policy sets for different profiles, whereas QoS control can help in controlling resource allocation to the flows.

## 1.2 Overview of the Work

This work includes the study of SDN and OpenFlow [5] for solving the service chaining problem in the enterprise networks. OpenFlow was introduced by the Open Networking Foundation. This work utilizes SDN as an infrastructure to achieve service chaining. It is difficult to achieve service chaining when we have mangling middleboxes in the network. Mangling middleboxes modify the packet header contents. This modification causes context loss and makes it more difficult to determine the next hop for packet, in the path of a service chain. The work done in this thesis tries to solve the problem of service chaining by supporting mangling middleboxes and without making any modifications to the middleboxes. The proposed system includes SPI to define and apply the policies across the network. It gives a flow, the service chain corresponding to a path through which it should traverse. SPI also helps in dropping unnecessary traffic at the edge switches e.g. certain web site is blocked for some user. The solution includes tagging the packets at the ingress switches and use this tag for steering. VLAN ID field of 802.1Q header is used to store the tags. The proposed system is implemented using OpenFlow version 1.0 [5] and floodlight [6] and tested with the MiniNet [7] network emulator.

## 1.3 Thesis Outline

This thesis is structured as follows. Chapter 2 briefly explains about the SDN and OpenFlow. We introduce the problem and summarize the limitations of the existing approaches in chapter 3 and 4 respectively. Chapter 5 and 6 describes system design and implementation of the proposed system respectively. We evaluate our system in chapter 7. Other possible solution within the scope of SDN

and OpenFlow for solving service chaining problem are discussed in chapter 8. Conclusions and future work are presented in chapter 9.

## Chapter 2

# SDN and OpenFlow

Computer networks are large, complex and difficult to manage. Traditional networks are classified considering their non-programmable, vertically integrated, closed and vendor specific architecture. Due to the lack of centralized control it is difficult to control and manage the traditional network. Networking devices run complex, distributed control software that is typically closed and proprietary and each device needs to be configured separately.

Software Defined Networking (SDN) is an approach to networking in which control is decoupled from the physical infrastructure and it promises to simplify the control and management of the network. SDN makes networks more simple, dynamic, open and programmable. OpenFlow [8] is the first open standard interface for implementing the SDN.

### 2.1 Software Defined Networking

A traditional networking device consists of data plane and control plane as shown in Figure 2.1. Data plane is responsible for forwarding a packet whereas control plane is responsible to determine where to forward the packet. For instance, in a learning switch, data plane forwards a packet and control plane maintains a MAC table i.e. reachability of a MAC addresses to determine the output port for an incoming packet. In SDN architecture, control plane is programmable and logically centralized (known as the controller) which allows network administrators to control all the data-plane elements by writing a single control program. For control plane it is possible to have direct access and manipulation of the forwarding plane. Centralized controller has a global view of the network.

Data plane communicates with a centralized controller through an open standard (such as OpenFlow). SDN facilitates the deployment of new services and protocols in the network, due to its vendor independence architecture and network virtualization. It also reduces the capital and operational costs for deploying and managing the network. Common SDN applications are network virtualization, network monitoring, network security, network policy implementation, load balancing, user authentication and cloud or data center network etc.

Figure 2.2 shows a logical view of SDN architecture. With a global view of the network at the controller, applications and policy-engines which are built on top of the controller, view networking devices as a single, logical switch. Networking devices needs to implement only basic packet forward-

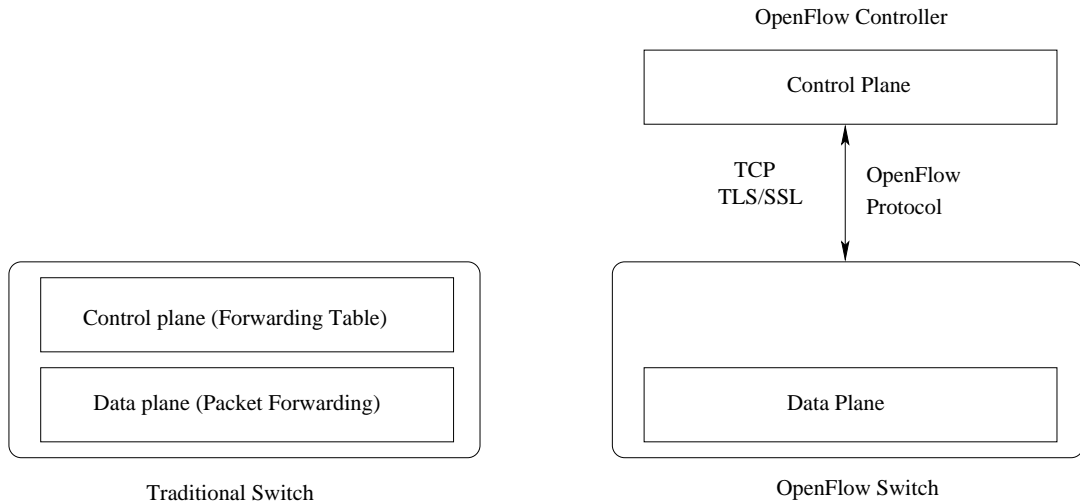


Figure 2.1: Traditional Switch vs OpenFlow Switch

ing mechanism and greatly simplifies network design and operation. It also facilitates vendor-neutral control over the network and real-time alteration of the network behavior.

SDN is not a new idea but has gained traction in recent times [10, 11]. Many vendors (such as Cisco) have their proprietary implementations of the concept of SDN. OpenFlow is a widely accepted implementation of SDN across the industry and academic research communities. The OpenFlow protocol is an open source and aims at making network programmable, innovative and vendor agnostic. One of the advantages of OpenFlow and its vendor independence is the rise of the concept of virtual switches. These are software switches which are implemented usually as user-space or kernel-space software. One such example is Open vSwitch [12, 13] which implements the OpenFlow protocol. This enables any regular computer to be used as a dedicated switching hardware and reduces the need of purchasing expensive hardware from proprietary vendors.

## 2.2 OpenFlow Protocol

OpenFlow [8] is a protocol designed by the Open Networking Foundation(ONF) which promotes and adopts SDN. OpenFlow was the first SDN standard to realize the concept of Software Defined Networking. The OpenFlow protocol is spoken between OpenFlow enabled switches and OpenFlow Controller as shown in Figure 2.1. OpenFlow provides flow based switching which allows to control the network on per-flow basis.

Match Fields	Priority	Counters	Actions	Timeouts	Cookie
--------------	----------	----------	---------	----------	--------

Table 2.1: A typical flow entry in a flow table

In OpenFlow enabled [5, 8], switches only consists forwarding plane equipped with flow tables which performs packet lookup and forwarding. A switch can have multiple flow tables containing several flow rules. Flow rules are similar to forwarding or routing rules in traditional switches and routers. Each packet is matched against flows rules present in the flow tables. A flow rule includes

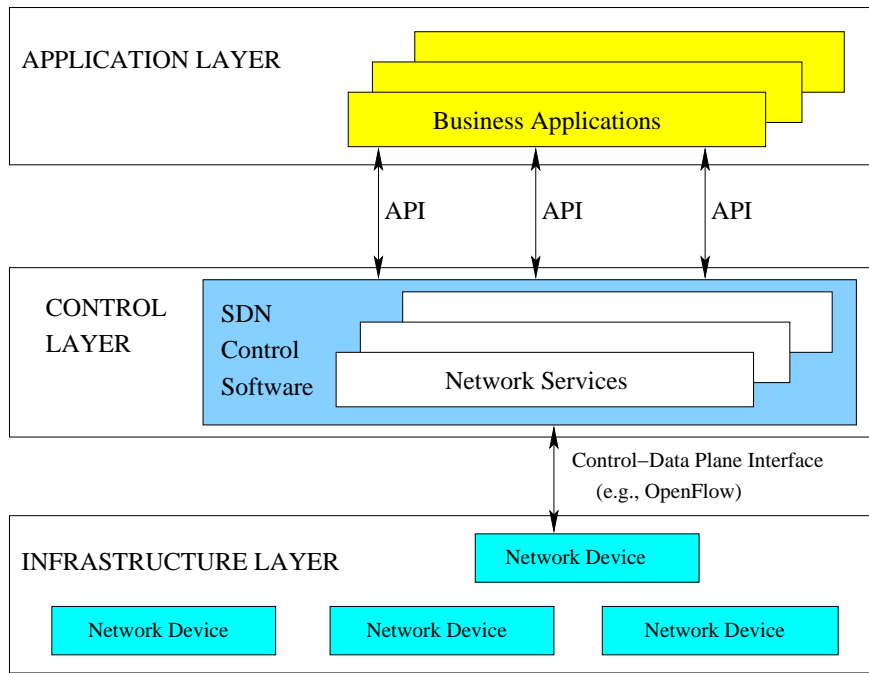


Figure 2.2: Software-Defined Network Architecture (Source: [9])

	Ethernet			VLAN		IP				TCP/UDP	
Ingress Port	Ether Src	Ether Dst	Ether Type	VLAN Id	VLAN Priority	IP Src	IP Dst	IP Proto	IP ToS bits	TCP/UDP Src Port	TCP/UDP Dst Port

Table 2.2: Match fields used to match against packets

a match, priority, counters, actions, timeouts and cookies as shown in Table 2.1. Match fields which are part of flow rules are used to compare against the incoming packets for matching. It currently supports matching up to the transport layer as shown in Table 2.2.

When a new flow comes to OpenFlow switch, it forwards the packet to the controller through the *packet\_in* message. Controller then determines the actions for this new flow depending on the logic implemented. Depending on this logic, an OpenFlow switch may function like a router, switch, firewall, or network address translator etc. To handle this new flow in future, controller either installs a flow rule on the switch by sending a *flow\_mod* message or sends a *packet\_out* message. In case of flow rule installation further *packet\_in* messages will not be sent to the controller for the same flow unless it is mentioned explicitly in the action. Each flow rule has set of actions to be taken on matching packets, e.g. it can forward the packet to a port or to the controller or it may simply drop the packet. These actions may also include modifications to packet header, e.g. changing the destination mac, changing the VLAN (Virtual Local Area Network) tag. Whenever a packer matches with the flow rule corresponding counters are updated and corresponding actions are carried out.

Each flow rules has two timeout values: *Idle\_timeout* and *Hard\_timeout*, which controls it's automatic expiration from the flow table. Flows can also be removed explicitly by the controller.

If there is a match on multiple flow rules then the match is defined to be with the highest-priority matching flow rule, where higher numbers are higher priorities. The cookie field is an opaque data value that is set by the controller at the time of flow rule installation. It is not used for processing packets but may be used by the controller to filter flow statistics, flow modification, and flow deletion. The OpenFlow protocol works on top of TCP and has support for TLS/SSL encryption which allows secure channel between controller and OpenFlow switches. Currently a few hardware vendors like Big Switch Networks, HP, and Pronto support OpenFlow in their hardware switches. Some of the available OpenFlow controllers are Floodlight [6], Ryu [14], Trema [15], NOX/POX [16] etc.

## Chapter 3

# Service Oriented Packet Forwarding in SDN

Service chain determines the set of services through which a particular flow of packets must pass before reaching its destination. Network policies typically require packets to go through more than one middleboxes at a time for example, network administrator might want to process HTTP traffic or web traffic, first through IDS followed by Proxy, because unmodified payload can be inspected at IDS. SDN can enforce such policies without the need of manually planning middlebox placements or statically configured routes, but at the same time SDN may lead to inefficient use of the available switch TCAM (e.g., we might need several thousands of rules)

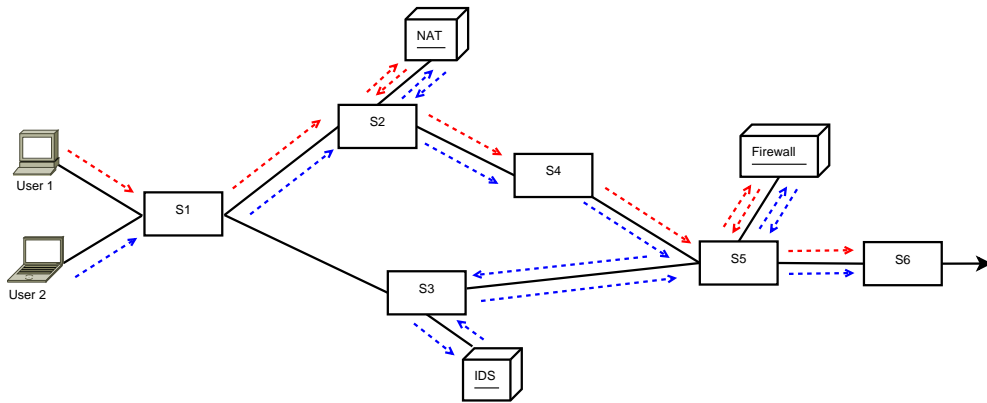


Figure 3.1: An Example of Service Chaining Problem

In Figure 3.1, there are three middleboxes each one providing a unique service. NAT translates private ip addresses to public ip addresses. Firewall allows and blocks access to certain destinations for internal users as well as to protect internal network from external attacks. IDS monitors users activities in the network for detecting malicious behaviour, it also alerts administrator for policy violations. Suppose there are two users in the network, network administrator has defined different policies for User<sub>1</sub> and User<sub>2</sub>. For User<sub>1</sub> all the traffic should follow middlebox policy NAT-Firewall whereas for User<sub>2</sub> all the traffic should follow middlebox policy NAT-IDS-Firewall. Unfortunately,

the traffic exiting the NAT will have same source ip thus, it is challenging to steer the User<sub>2</sub>'s traffic at S<sub>5</sub> so that it does not violate policy. Many middleboxes are stateful and require visibility into both forward & reverse flows for correctness, SDN needs to ensure that, a flow passes through same stateful middleboxes in both direction. Dynamic modification of the incoming traffic (especially the packet headers) at the middleboxes is the key issue for installing correct forwarding rules at the switches to steer the flow as desired. Once the packets belonging to a flow are modified at a middlebox, then the forwarding rules in downstream switches on the path of the flow must account previously done modifications e.g. NAT translation as explained in earlier example. So in this case it is important for the controller to know such translation and thus installs the correct forwarding rules to direct the traffic to the next middlebox or to the correct destination. But this requirement makes the controller dependent on the middlebox behavior, and due to the closed and proprietary nature of the middleboxes it further complicates the problem.

As the complexity and scale of enterprise network increases, it is becoming more difficult to rely on the manual configurations as it fails to ensure the following highly desirable properties which are necessary for service chaining.

- **Correctness [17]:** Right traffic should traverse through right sequence of middleboxes under all possible conditions, as specified by the network administrator in the form of network policies. Configuring layer-2 switches and layer-3 routers to enforce service chain involves tweaking large number of network devices. It is highly complex and error-prone process. For example, physical topology change such as the failure or addition of a network link may cause the network traffic getting routed through alternative path instead of the network path containing mandatory firewall, which violates data center security policy.
- **Flexibility [17]:** Adding, removing or updating the policy for a particular applications traffic i.e a flow in the network, should be easily configurable. Doing it manually requires significant engineering and configuration changes. e.g. adding an SSL offload box facing web traffic needs identification of a point in the network through which all web traffic passes and then manual insertion of the SSL offload box at that point.
- **Efficiency [17]:** Traffic should not travel across unwanted middleboxes. To force the traffic to pass through middleboxes, on-path deployment is carried out, which causes all traffic passing through a specific route in the network to traverse the same sequence of middleboxes which are deployed along the route. However, different applications have different requirements, e.g. A simple web application may require its traffic to pass through a firewall and then by a load-balancer, whereas an Enterprise Resource Planning (ERP) application may require all its traffic must be processed by a dedicated custom firewall followed by an intrusion prevention system. Since all traffic passes through the same middleboxes, the intrusion prevention box and the custom firewall will be utilized unnecessarily for the web traffic. This causes wastage of the valuable network resources.
- **Scalability:** The number of middleboxes deployed in an enterprise networks are increasing day by day and hence the system should scale according to hundreds-to-thousands of possible service chains. At the same time high-speed memory (such as TCAMs) of the SDN enabled switches is limited. Number of flow rules required for service chaining should be within the



limitations of the available resources and system should scale as per the requirements of the enterprise network for new middleboxes.

## Behaviour of various Middleboxes

Middlebox	Input	Actions	Timescale
FlowMon	Header	No change	-
IDS	Header, Payload	No change	-
IP Firewall	Header	Drop?	-
Redundancy eliminator	Payload	Rewrite payload	Per-packet
NAT	Flow	Rewrite header	Per-flow
Load balancer	Flow	Rewrite headers & reroute	Per-flow
Proxy	Session	Map Sessions	Per-session
WAN-Opt	Session	Rewrite header	Per-session

Table 3.1: Taxonomy of the dynamic actions performed by different middleboxes (Source: [4])

Table 3.1 summarizes the various types of middleboxes, commonly used in today’s enterprise network and their characteristics such as, the kind of input traffic they process, their actions, and the timescales at which the dynamic traffic transformations are carried out. For example, NAT rewrites source and destination IP and port fields after checking its state table while firewall checks packet header and payload information to decide whether to drop the packet or forward it ahead. It is important to note that middlebox nature may differ from vendor to vendor. For example, the NAT may assign the source port from the pool of available ports or it may increase it randomly or sequentially when a new host connects. This depends on the vendor specific implemented logic. In addition to this we can observe that middlebox processing takes place at different timescales, modifies content at different layers and operate at various granularity’s e.g. per-packet, per-flows or per-session. Fig 3.2 shows TCP Segment processing on the today’s Internet (Source: [18, 19]). All the fields in red color get modified by the middleboxes such as router, NAT and Application Level Gateways.

Lack of availability of protocols and tools to carry out necessary configuration to steer the traffic and various existing approaches to solve this problem such as utilizing various fields in the packet header to store the contextual information for steering [3, 20], heuristic correlation of the flows to track the flow for further steering, extensions to middleboxes in order to generate and consume contextual information in the form of tag at the middleboxes, providing separate dedicated headers for service chaining, fail to satisfy the highly desirable properties as mentioned earlier. Prior work (e.g. [3, 20]) has repurposed the DSCP bits in the IP header for tagging, which then used for correct steering. However this approach may work in practice but they are based on the assumption that DSCP bits are not used within the network and middleboxes emit them unmodified. The former assumption prevents QoS use within the network, and the latter may not be true at all.

This thesis proposes a novel approach to solve the service chaining problem by tagging the packets at the ingress switches and use this tag for steering. VLAN ID field of 802.1Q header is used for storing the tags. The proposed system includes Shallow Packet Inspector (SPI) to define and apply the policies across the network. These policies define the service chain corresponding to a flow

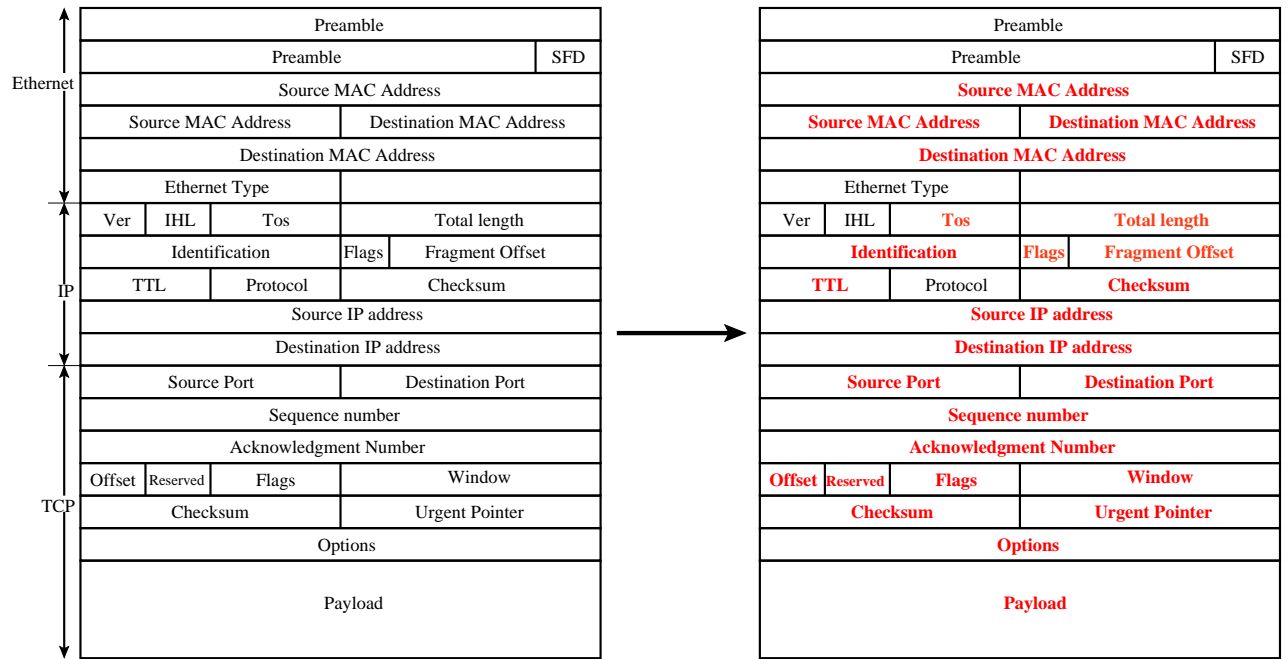


Figure 3.2: MAC, IP and TCP Segment processing on the today's Internet

through which it should traverse. In addition to successfully ensuring service chaining this system also helps in dropping unnecessary traffic at the edge switches e.g. certain web site is blocked for some user, then corresponding blocking rule will be installed at the user's ingress switch.

# Chapter 4

## Related Work

Service chaining or middlebox sequencing is a known problem and is of growing importance. Various approaches have been proposed to solve it in recent years.

Single box running multiple services: this approach combines multiple services into a single box. New services can be added by adding new service cards in the existing box, but in case of closed and proprietary middleboxes this approach makes it difficult for integration. It also suffers from the scalability issues as a single box can not be expanded beyond the practical limitations e.g. limited bandwidth capacity. On top of these issues it is vulnerable to single point of failure.

Network Service Header (NSH) [21] also tries to solve the service chaining problem by adding an additional header to each packet which can be used by middleboxes for traffic steering. However this approach necessitates the middleboxes to be aware of the additional headers and in turns creates the need to modify middleboxes. The main drawback is the increased overhead due to the additional header for each packet.

StEERING [2] utilizes pipeline feature introduced in OpenFlow version 1.1 for service chaining. OpenFlow version 1.1 supports 64-bit metadata field [8]. It's design requires one bit for the direction and remaining 63 bits are used to encode service chain, allowing a maximum of 63 distinct services in the network. It also talks about how to select the best locations for placing the services in the network in order to optimize the performance. However it does not clearly mention how to deal in case of mangling middleboxes, as well as how to ensure the processing of flow in a given order of services.

SIMPLE [4] introduces heuristic correlation of the traffic payloads while entering and leaving middleboxes to correlate flows. It requires multiple packets to be sent to the controller for correlating and is error-prone, SIMPLE has 19% error rate. Finally, this process has high overhead, as multiple packets per flow need to be processed at the controller in a stateful manner (e.g., when reassembling packet payloads). It mainly focuses on balancing the middlebox load, one major contribution of the paper is an approach to tracking packets when processed by service functions that modify the header information.

FlowTags [3] proposes simple extensions to middleboxes to add tags, carried in packet headers. It associates additional contextual information in terms of tags with a traffic flow as it traverses the network. It has two main drawbacks first one requires middlebox modifications due to tag generation & consumption at middleboxes. and second one introduces additional overhead due to rewriting of

packet headers at every middlebox.

<b>Framework</b>	<b>Handles Mangling</b>	<b>Maintains Affinity</b>	<b>No Middlebox Modifications</b>	<b>Minimal Rules</b>
pLayer [17]	✓	×	✓	?
SIMPLE [4]	≈	×	✓	✓
Per-Flow Rules	×	✓	✓	×
FlowTags [3]	✓	?	×	✓
StEERING [2]	?	✓	✓	×
SC Header [21]	✓	?	×	?

Table 4.1: Comparison of existing approaches for service chaining (Source: [20])

# Chapter 5

## System Design

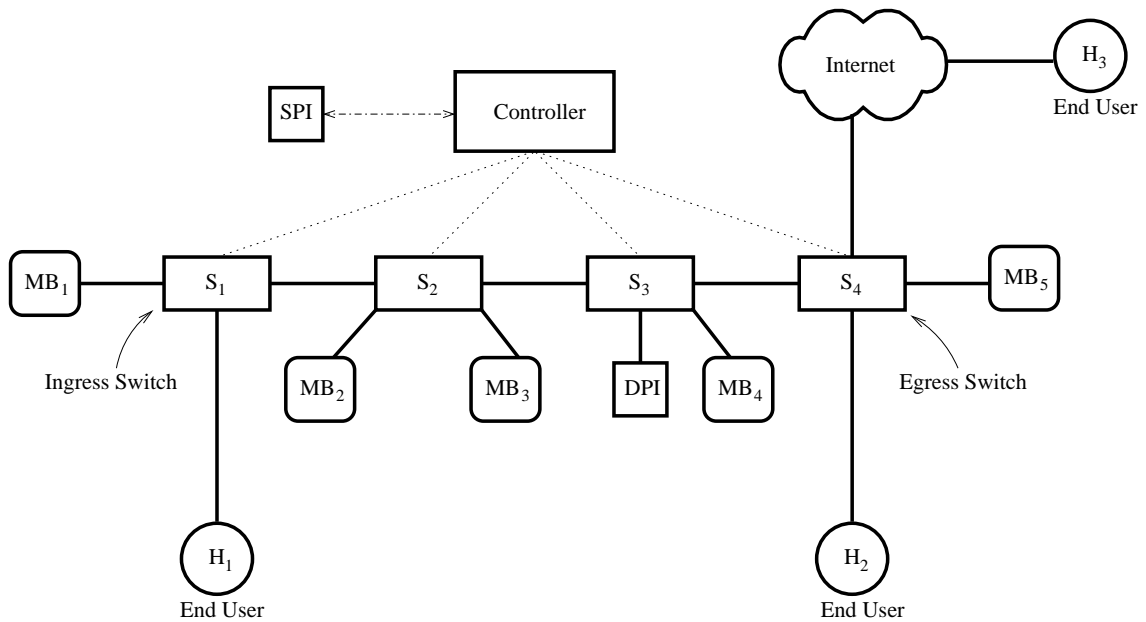


Figure 5.1: Expected Network Diagram for Designing System

Figure 5.1 shows expected network diagram for proposed system, which consists of SDN switches, SPI and a SDN Controller. Controller controls every switch in the network and communicates with SPI through an API. The forwarding plane can be configured in order to meet the requirement of flow steering. Controller is responsible for implementing network policies (service chain) through the tag-based flow rules. Controller installs such flow rules on the switches. Tag is stored in VLAN ID field of 802.1Q header. Each packet of the flow, is added with an additional 802.1Q header and forwarded based on VLAN ID stored in it.

Figure 5.2 shows the flow of messages in SDN controller. Controller receives the information about a new flow through *packet\_in* message. Steering module and Learning Switch module processes *packet\_in* message at the controller. Steering module extracts the original packet from the *packet\_in* message and feeds it to SPI. SPI processes original packet based on the admin defined policies.

Output generated by SPI contains either the drop action to drop the flow or the meta-information including middlebox sequence through which the flow should traverse. Steering module then installs appropriate flow rules on corresponding switches with the help of Tag Generator module. Tag Generator module is responsible for tag generation depending on the direction of the flow. The Tag Generator module is decoupled from the rest of modules for augmentation purpose.

Flow rules installed by steering module performs one of the following actions on the input packet,

1. Add VLAN ID and then forward (adding tag for forward direction flow)
2. Forward a packet through an appropriate outgoing port
3. Rewrite destination mac address, VLAN ID and then forward (forwarding to a middlebox)
4. Strip VLAN ID and then forward (done processing by all middleboxes, ready to deliver to destination)
5. Add VLAN ID, VLAN PCP and then forward (adding tags for reverse direction flow)

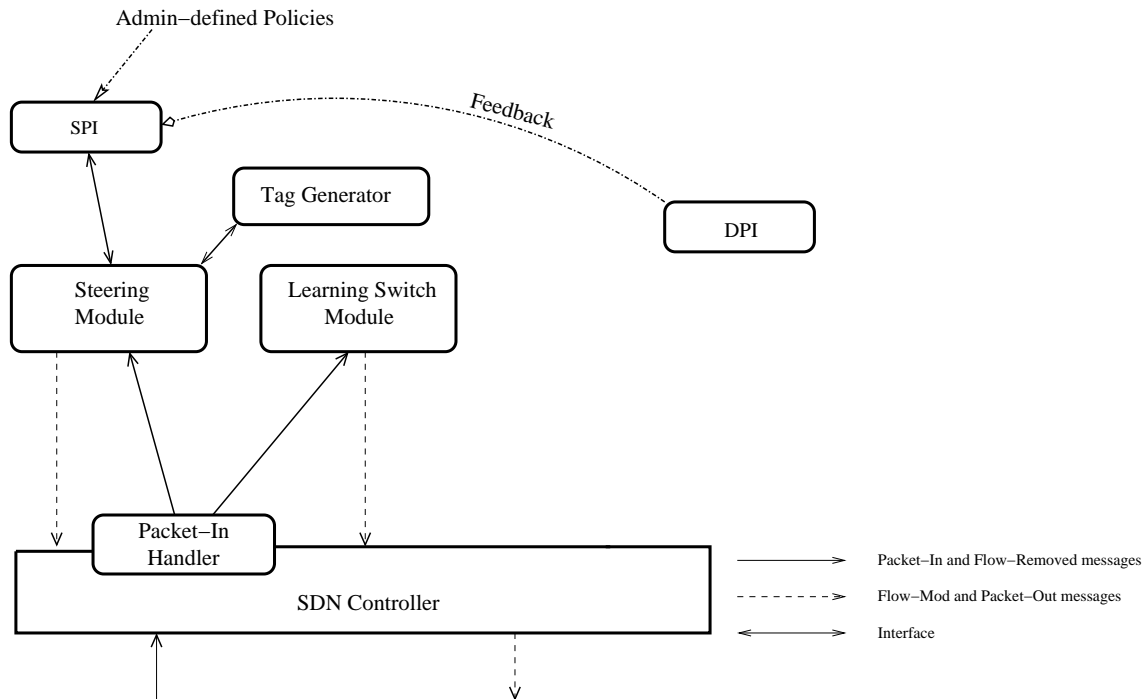


Figure 5.2: Flow of messages in the system

The basic design components of the system are explained below.

## 5.1 Shallow Packet Inspector

When new flow comes, *packet\_in* message is sent to the controller by an ingress switch. Controller utilizes SPI to classify the flow and to find related information e.g. middlebox sequence, the flow

should traverse through. SPI contains admin defined policies, these policies can be updated dynamically. The policies defined at SPI are lightweight for processing compared to DPI. Controller then calculates the appropriate tag values based on the middlebox sequence and installs the flow rules on corresponding switches. This requires only those primitives commonly available within SDN (e.g. those defined by the OpenFlow 1.0 standard).

## 5.2 Packet Tagging at Ingress & Egress Switch

The flow rules installed by controller on ingress switch tags the packets of the forward direction flow whereas the egress switch tags the packets of the reverse direction flow. This tag is used for forwarding the packet to pass through the middleboxes within the network, as specified by the policy. Some middleboxes are stateful and requires to process the flow in both direction. In order to maintain flow affinity controller proactively installs flow rules for the reverse flow. Reverse flow passes through the same middleboxes in reverse order.

## 5.3 Flow Rule Compaction

To reduce the number of forwarding entries in larger networks, we leverage the observation that intermediate switches of each segment of a physical chain of services do not need fine grained flow rules (e.g. 5-tuple flow rules). The only role they serve is to forward the packet toward the switch that is connected to the next middlebox in the sequence. In the proposed system intermediate switches use tag to forward the packet within the network. Tag corresponds to a service chain. Intermediate switches forward packets based on tag, until it is passed through all the services indicated by tag. Multiple flows can have same policy (service chain), which results into same tag for multiple flows. Same tag based flow rules are used for forwarding if these flows. This avoids the need of having flow specific rules on intermediate switches and results in compact flow table.

## 5.4 Deep Packet Inspector

Some flows may be classified as suspicious by SPI in the initial stage. DPI has more resource consuming policies compared to SPI e.g. policies which require application layer detection. Due to the exclusion of the lightweight policies DPI is fully utilized for the later seven. DPI will monitor network traffic and analyze it against a rule set defined by the administrator. DPI will then generate the log based on what has been identified. e.g. torrent traffic. The log generated by DPI after processing these flows is provided as a feedback to SPI. It is practically possible to generate the lightweight policies from the log generated by DPI. It leverages DPI's  $L_4$ - $L_7$  capability furthermore reduces processing of such flow in near future and improves DPI's utilization.

## 5.5 Policy Specification

Network policies can be viewed as rules. Each policy is a set of policy\_id, action, matching conditions, meta-information, priority, timestamp. Every policy has its own id i.e. policy\_id. Matching conditions includes protocol, source ip & port, destination ip & port and direction. Direction tells

which way the signature has to match i.e. single direction or both directions. Meta-information consists of middlebox sequence that the matching packet should follow. Policies can be prioritized. Priority is a numeric value which can range from 1 till 255. Policy with a higher priority will be examined first. The highest priority is 1. Timestamp follows unix time system which is defined as the number of seconds that have elapsed since 00:00:00 Coordinated Universal Time (UTC), Thursday, 1 January 1970.

An incoming packet to SPI is compared against the conditions of the policies. If a match occurs between the rule and the incoming packet, the action & meta-information defined in the rule are provided to controller. Controller takes the necessary actions based on what has been identified. In order to apply a network policy the incoming packet must match against the conditions mentioned in the policy.

- **Policy-1: allow tcp 10.0.0.1 any → 10.0.0.5 any (msg:“Service Chain:1-2-3”; priority: 10, timestamp:1434782226, sid:33333)**
- **Policy-2: drop tcp 192.168.0.10 any ↔ 182.6.45.3 any (msg:“bad-unknown”; priority: 1, timestamp:1434782226, sid:44444)**



## Chapter 6

# System Implementation

### 6.1 Data Structures

Following data structures is defined and maintained to efficiently store and retrieve the information about policy match results, middleboxes and their placement in the network. Controller feeds the actual *packet\_in* to SPI and determines the corresponding action as specified by the matched policy. Then flow rules are installed on the appropriate SDN switches with appropriate match and action primitive for steering flows.

- **RuleIdentifier** is a class which has 5 tuple which includes Source IP address, Destination IP address, Source Port number, Destination Port number and Protocol as a data member.
- **RuleIdentifierAction** is a class which contains middlebox sequence and action (allow/drop) as a data member.
- **MBCConnect** is a class whose data members give middlebox specific information such as attach point (Switch DPID, port), MAC address and its sequence number (ID) among all the available middleboxes in the network. This class can be extended for adding in more middlebox specific information.
- **RuleList** is a hash table where *RuleIdentifier* is used as a hash key to look up Switch DPID. It is mainly used to maintain flows for which SPI action is to be decided by SPI.
- **RuleListAction** is a hash table where *RuleIdentifier* is used as a hash key to look up middlebox sequence and action.
- **SwidMb** is a hash table where middlebox identifier (as specified in the sequence) is used as a hash key to look up the *MBCConnect* which provides middlebox specific information.

It is important to note that either *RuleIdentifierAction* or *RuleListAction* will be filled once we get the response from SPI, *RuleIdentifierAction* gives the action as per the matched policy for *RuleIdentifier*'s 5 tuple. *SwidMb* provides the middlebox specific information as well as its attachment point in the network. This attachment point is provided by administrator at the time of its introduction in the network.

## 6.2 Proactive Flow Rule Injection

Steering module as shown in 5.1 is responsible for injection of proactive flow rules. Proactive flow rules allow each switch to know how to process a particular flow in advance, rather than reacting to each new *packet.in*. Proactive flow rules eliminates the latency induced due to controller consultation, for obtaining the flow rule upon arrival of every new flow. Proposed system installs proactive flow rules on the intermediate switches along the end-to-end path for both direction flow. This helps in eliminating the expected delay that a known flow will experience, and all packets are forwarded at line rate.

## 6.3 Encoding tag in 802.1Q header

As stated in Section 5.2, packets belonging to a flow are tagged at ingress switches. In order to preserve tags across various middleboxes, we make use of VLAN ID field in 802.1Q [22] header. VLAN ID field in 802.1Q header is of 12 bits. We divide these 12 bits in 4 groups, each containing 3 bits. First 3 groups are for storing middlebox sequence and the last group is for storing the *count* of middleboxes as shown in Figure 6.1. In case of forward direction flow the last group shows *count* of middleboxes that the packet has already traversed. In case of reverse flow it shows the  $(count - 1)$  middleboxes that packet has yet to be traversed. Count starts with 0 for forward direction flow and 1 more than the number of middleboxes in the sequence, for the reverse direction flow.

3 bits in a group allow us to support 7 distinct middleboxes (after ignoring 000 combination) for sequencing, but only 3 can be sequenced at a time, as per the current algorithm. As defined by IEEE 802.1Q, Priority Code Point (PCP) can be used by QoS disciplines to differentiate traffic. Its usage is undefined and left to the implementation. We use PCP field to indicate the direction of the flow. In the current implementation PCP field is utilized only for indicating the direction of the flow. In case of reverse flow, this field is set with a unique combination out of the seven possible combinations, currently we are using binary 011 i.e. decimal 3. Both PCP field and VLAN ID are available in OpenFlow v1.0 to specify as part of the matching field in a flow rule. Using both of these fields the flow can be steered in the network, no matter how a middlebox may modify the packet contents, as these fields will be preserved.

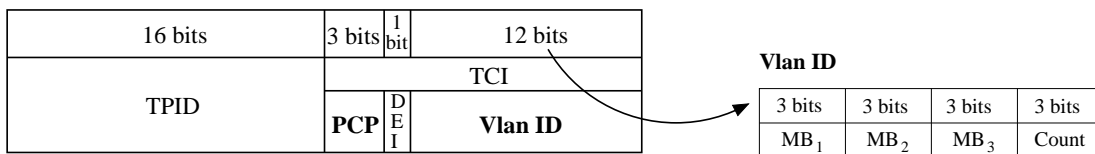


Figure 6.1: 802.1Q Header, utilized for storing tags

## 6.4 Network Configuration and Traffic Handling for Middleboxes

In the proposed system, VLAN ID is used to indicate the service chain through which a packet has to travel. The following Table 6.1 shows the mapping between a particular VLAN ID and the

middlebox sequence indicated by it. Each middlebox can have multiple VLAN IDs, that are used for serving to the flow in different service chains. Each of the logical interfaces is configured with VLAN ID which corresponds to the part of service chain, e.g. Table 6.2 shows the conversion of middlebox sequence 2-1-3/2 to the VLAN ID 1113. MB<sub>1</sub> should be configured with VLAN ID 1113. Refer 8 for how the vlan configuration is carried out on Linux based server. All the packets tagged with VLAN ID 1113 will be forwarded to MB<sub>1</sub>. VLAN ID 1113 indicates that packet has to traverse the service chain of MB<sub>2</sub>-MB<sub>1</sub>-MB<sub>3</sub>, and till now MB<sub>2</sub> and MB<sub>1</sub> has been traversed and next it should be forwarded to MB<sub>3</sub>. Because a logical interface can have multiple IP addresses, each from different subnets, even a service which, requires packet forwarding over multiple subnets, e.g. NAT and firewall, can be provided on only one logical interface using one VLAN ID. The following Figure 6.2 shows the configuration of a network interface on a middlebox that is serving as NAT.

Upon the arrival of a new flow, the SDN switch learns how it should tag the matching packets and forward. The switch refers the input port and VLAN ID of an incoming packet and converts the VLAN ID to the one which is owned by the corresponding middlebox's logical interface. The packet is sent to the middlebox with a proper VLAN ID for processing. Destination MAC address is rewritten depending on the requirement of the middlebox, as some middleboxes do not require an incoming packet to have destination mac address to be the same as that of the middlebox's interface, e.g. an Intrusion Detection System (IDS) monitors the network traffic, by putting its interface into promiscuous mode. An interface in promiscuous mode passes all traffic to operating system rather than passing only the frames that the CPU is intended to receive. All network traffic is passed to such an interface using port mirroring technique.

No matter how the corresponding packet is modified in IP header or other upper layer headers, it comes back from the middlebox using the same VLAN ID to the switch. Using the VLAN ID, the SDN switch can still determine the next middlebox that the incoming packet has to be destined. The new VLAN ID will remain same till it reaches switch which is connected to next middlebox in the sequence. After reaching the next switch the same procedure repeats till the egress switch where the VLAN ID is stripped and packet is forwarded to its destination. In the proposed system, the VLAN IDs to be assigned to middleboxes need to be carefully managed and configured together with IP address to be used for the service. In the current system network configuration for middleboxes is done manually, we plan to automate this procedure in the future work.

Middlebox-Sequence/Count	Vlan ID
1-2-3/0	664
1-2-3/1	665
1-2-3/2	666
1-2-3/3	667
1-2-3/4	668

Sequence MB<sub>1</sub>-MB<sub>2</sub>-MB<sub>3</sub>

Middlebox-Sequence/Count	Vlan ID
2-1-3/0	1112
2-1-3/1	1113
2-1-3/2	1114
2-1-3/3	1115
2-1-3/4	1116

Sequence MB<sub>2</sub>-MB<sub>1</sub>-MB<sub>3</sub>

Table 6.1: Middlebox-Sequence/Count and corresponding Vlan ID

Sequence									<i>count</i>			Representation		
MB <sub>2</sub>			MB <sub>1</sub>			MB <sub>3</sub>			2			2-1-3/2		
0	1	0	0	0	1	0	1	1	0	1	0	1113		

Table 6.2: Conversion of Middlebox-Sequence/Count (2-1-3/2) to VALN ID (1113)

Middlebox Sequence/Count	Vlan ID	PCP	Meaning
1-2-3/0	664	0	Forward Direction Packet has not traversed any middleboxes
1-2-3/1	665	0	Forward Direction Packet has traversed First middleboxes in the sequence
1-2-3/4	667	3	Reverse Direction Packet has to traverse (4 – 1) i.e. 3 middleboxes in reverse direction of the sequence
1-2-3/3	666	3	Reverse Direction Packet has to traverse (3 – 1) i.e. 2 middleboxes in reverse direction of the sequence

Table 6.3: Middlebox Sequence/Count, PCP and its meaning

## 6.5 Addition of a new Middlebox in the Existing Network

Whenever a new middlebox is added to the network, Administrator needs to update the database containing middlebox specific information such as attach point (Switch DPID, Port), MAC address and its sequence number (ID) among all the available middleboxes in the network. Middlebox-specific information may introduce the requirement of changing the destination MAC address or copying the packet to the port connected to the middlebox. In addition Administrator also needs to determine VLAN IDs to be assigned to the middlebox depending on its sequence number (ID). All possible VLAN IDs together with IP addresses should be carefully configured. This configuration can be done dynamically using *expect* script [23], *expect* scripts can help automate this configuration procedure. Current implementation of the system requires this configuration to be done manually.

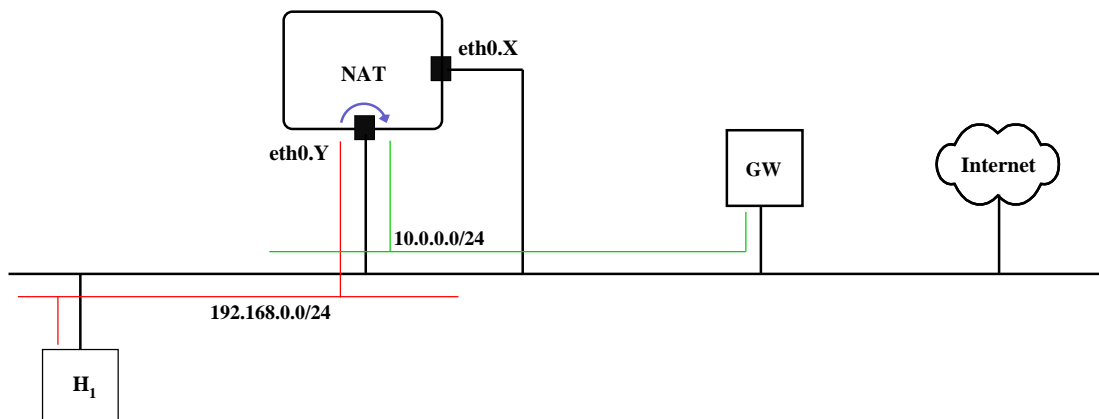


Figure 6.2: Example Configuration for NAT

## 6.6 Dynamic Policy Change

Administrator can change the network policies dynamically e.g. in Figure 3.1 administrator might want to change the initially defined policy for User<sub>2</sub> which is NAT-IDS-Firewall to IDS-NAT-Firewall. In case of such policy change the current implementation of system does not reflect the change immediately, but the modified policy will be applied after the completion of the existing flow. It is possible to incorporate the immediate reflection of policy change with current system design. We plan to extend current system implementation to include this feature in future work.

# Chapter 7

## Evaluation

### 7.1 Proof of Concept

As a proof of concept, service chaining was verified with virtualized topology created with MiniNet [7], enabling Floodlight 0.90 [6] as SDN controller and Open vSwitch 1.9.0 [12,13] as SDN Switch on MiniNet VM.

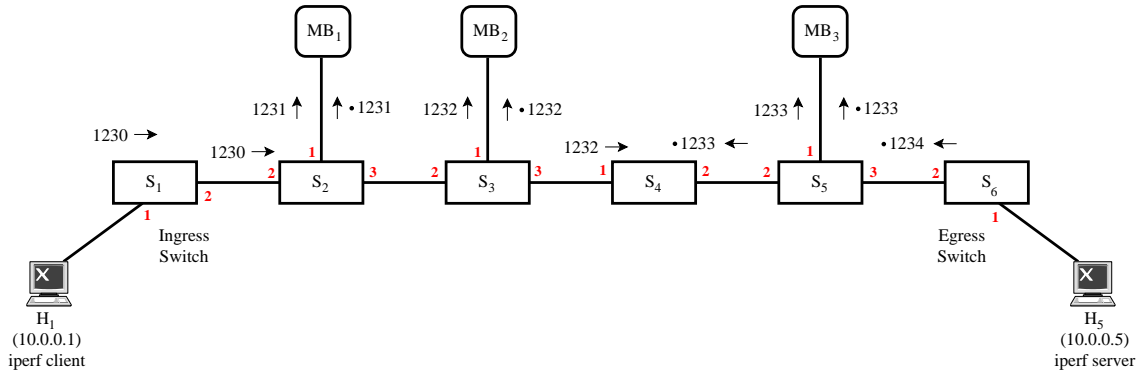


Figure 7.1: Test Scenario

As shown in Figure 7.1 test topology consists of six OpenFlow enabled switches, 3 synthetic middleboxes and two hosts H<sub>1</sub> & H<sub>5</sub>. Policy as shown below is defined at SPI. We implemented synthetic, libpcap based middleboxes for imposing various types of middlebox behavior on the network traffic.

- Policy : alert tcp 10.0.0.1 any → 10.0.0.5 any (msg:“1-2-3”; sid:33333)

Policy indicates, tcp traffic for Network Source 10.0.0.1 (H<sub>1</sub>) and Network Destination 10.0.0.5 (H<sub>5</sub>) should follow the service chain MB<sub>1</sub>-MB<sub>2</sub>-MB<sub>3</sub>. This policy has the policy\_id 33333. We used iperf to generate tcp flow between H<sub>1</sub> and H<sub>5</sub>. Upon arrival of *packet.in* message from the ingress switch S<sub>1</sub>, controller proactively installs the flow rules shown in Table 7.1. Installed flow rules also consider the reverse flow i.e. from H<sub>5</sub> to H<sub>1</sub>. Once the VLAN header is stripped from Ethernet

frame, e.g. for a flow from  $H_1$  to  $H_5$ ,  $S_5$  strips the 802.1Q header from Ethernet frame, packet can be forwarded toward the destination simply based on its destination mac address. Learning Switch module in Floodlight [6] can take care of such flows. Since there was no other traffic we examined per-interface logs including timestamp to verify that packet is following intended path.

### 7.1.1 Time to Install Flow Rules

The installation of flow rules was performed to all the switches along the path for both forward and reverse direction flow upon the arrival of a new flow on  $S_1$ . Time required for installing flow rules on all the switches was 637 ms i.e. actual time required to process *packet\_in* message. Out of the total time required to process *packet\_in* message significant amount of delay was introduced by SPI, it was 614 ms. Which means 27 ms were required by the steering and tag generator module to install the flow rules, once we receive response from SPI.

## 7.2 Overhead : SPI and Additional 802.1Q Header

Controller takes help of SPI to classify the flow and to find the related information, this introduces delay in the controllers decision making process. Each packet of the flow is added with an additional 802.1Q header, VLAN ID field is used to store tag required for steering. This causes the overhead of 4 Bytes with each packet of the flow. For 1 MB data the overhead incurred due to additional 802.1Q Header is approximately 3 KB. Efficiency of packet tagging is directly proportional to the packet size, Use of 802.1Q Header in an enterprise network is very common. does not introduce significant overhead on the operation of the enterprise network

## 7.3 Benefits of Our System

No changes are required to the middleboxes to enforce service chaining. The system is friendly with the existing Layer 2 switches because they also understand VLAN's and can be configured in truncated mode to forward the tagged packets. Due to the partitioning of policies between SPI and DPI the load on DPI is reduced. System exhibits distributed firewall nature due to the drop action performed by the controller on the ingress switch for the end host. Dropping unnecessary traffic at ingress switches improves bandwidth utilization and performance of the network. Middlebox placement does not affect the correctness of the solution, it allows flexibility in middlebox placements.

## 7.4 Number of Flow Rules

Total number of flow rules required across all the switches, for steering are directly proportional to the number of middleboxes. We can find out this number using,

$$X = \sum_{k=1}^n \frac{n!}{k!} \quad (7.1)$$

where  $X$  gives total number of flow rules required, when  $n$  number of middleboxes are considered for the service chaining in the network.

Table 7.1: Flow Rules installed on the switches for Sequence 2-1-3

Sw	Dir	Match	Action
S <sub>1</sub>	F	tcp,nw_src=10.0.0.1,tp_src=4786, nw_dst=10.0.0.5,tp_dst=5001	actions=mod_vlan_vid:1112,output:2
	F	dl_vlan=1112	actions=output:3
S <sub>2</sub>	F	dl_vlan=1113	actions=mod_vlan_vid:1114,mod_dl_dst:00:00:00:00:02,output:1
	F	dl_vlan=1114	actions=output:3
	R	dl_vlan=1115,dl_vlan_pcp=3	actions=mod_vlan_vid:1114,mod_dl_dst:00:00:00:00:02,output:1
	R	dl_vlan=1114,dl_vlan_pcp=3	actions=output:3
S <sub>3</sub>	F	dl_vlan=1112	actions=mod_vlan_vid:1113,mod_dl_dst:00:00:00:00:03,output:1
	F	dl_vlan=1113	actions=output:2
	F	dl_vlan=1114	actions=output:3
	R	dl_vlan=1115,dl_vlan_pcp=3	actions=output:2
	R	dl_vlan=1114,dl_vlan_pcp=3	actions=mod_vlan_vid:1113,mod_dl_dst:00:00:00:00:03,output:1
	R	dl_vlan=1113,dl_vlan_pcp=3	actions=strip_vlan,output:2
S <sub>4</sub>	F	dl_vlan=1114	actions=output:2
	R	dl_vlan=1115,dl_vlan_pcp=3	actions=output:1
S <sub>5</sub>	F	dl_vlan=1114	actions=mod_vlan_vid:1115,mod_dl_dst:00:00:00:00:04,output:1
	F	dl_vlan=1115	actions=strip_vlan,output:3
	R	dl_vlan=1116,dl_vlan_pcp=3	actions=mod_vlan_vid:1115,mod_dl_dst:00:00:00:00:04,output:1
	R	dl_vlan=1115,dl_vlan_pcp=3	actions=output:2
S <sub>6</sub>	R	tcp,nw_src=10.0.0.5,tp_src=5001, nw_dst=10.0.0.1,tp_dst=4786	actions=mod_vlan_vid:1116,mod_vlan_pcp:3,output:2



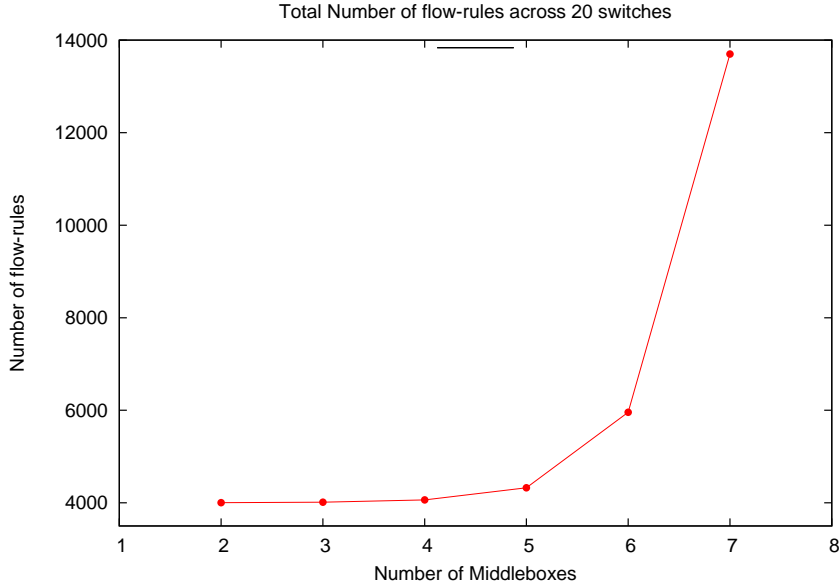


Figure 7.2

Graph 7.2 shows the comparison between number of flow rules required in case of flow specific i.e. 5-tuple flow rules and number of flow rules required by our system to steer the flow. We consider 200 distinct flows and 10 switches in the network. We assume that all the middleboxes are spread across the network and steering of flows requires to pass through all the switches. It is clear from the graph that this solution does not scale when we have more than 7 middleboxes in the network.

## 7.5 Discussion

### 7.5.1 Double Tagging for Scalable Service Chaining

As discussed in Section 7.4 the proposed system does not scale if number of middleboxes in the network are more than six. To overcome this, the similar approach can be implemented using IEEE 802.1ad. IEEE 802.1ad is a protocol for carrying VLAN traffic on an Ethernet. It is based upon 802.1Q, but allows for VLANs to be nested by adding two tags to each frame instead of one. The header is as shown in Figure 7.3. This allows us to use an additional 802.1Q header for the earlier discussed approach, it can solve the scalability problem.

The idea is, we can represent 4094 distinct service chains with *outer* VLAN ID, *inner* VLAN ID can be used to store the next middlebox in the sequence represented by *outer* VLAN ID. Lets consider a middlebox is providing service on an interface `eth0.b.x`. When the packet is tagged with double tag containing *outer* tag  $x$  and *inner* tag  $b$ ,  $x$  represents the middlebox sequence  $MB_a-MB_b-MB_c-MB_d$  and  $b$  represents that the packet has already traversed  $MB_a$ , next it should be forwarded to  $MB_b$ . *Match* and *Set* double-tagged VLANs (QinQ) is currently not supported in Open vSwitch [24]. As of linux kernel version 3.10 (released in June 2013) includes support for 802.1ad. Both inner and outer VLANs are handled by the attached machine (which sees double-tagged 802.1ad VLAN frames).

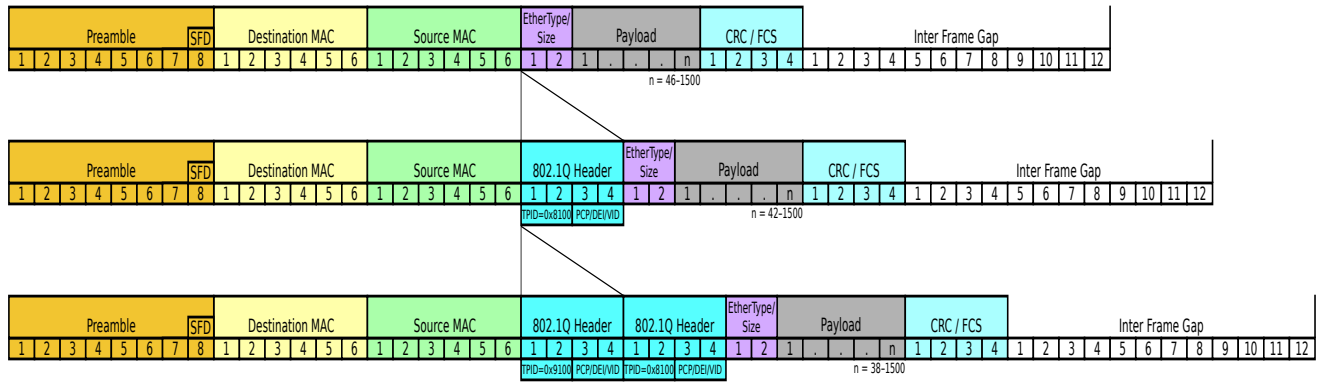


Figure 7.3: Insertion of 802.1ad double tag in an Ethernet frame [25]

## 7.5.2 Middlebox Placement Problem

Middlebox placement problem as explained in [2] clearly has an impact on network performance. Though the middlebox placement is not bounded by any constraints, certain placement strategies are better and affects end user experience. For example in Figure 3.1, lets consider that it is mandatory for all the traffic to pass through IDS, then placing IDS at  $S_5$  will avoid the delay introduced for steering between  $S_5$  and  $S_3$ .

## 7.5.3 SPI Interface

The delay introduced due to external entity (Suricata) can be reduced further by modifying the existing source code, this requires source code study. The goal is to have one single CPU to treat the packet from the start to the end. Suricata has different running modes which define how the different parts of the engine (decoding, streaming, signature, output) are chained.

As explained in [26], One of the mode is the *workers* mode where all the treatment for a packet is made on a single thread. This mode can be modified and adopted in our system. it permits to keep the work from start to end on a single thread. By using the CPU affinity system available in Suricata, we can assign each thread to a single CPU. By doing this the treatment of each packet can be done on a single CPU. Finally we need to consider the link between the CPU receiving the packet and the one used in Suricata. To do so we have to ensure that when a packet is received on a queue, the CPU that will handle the packet will be the same as the one treating the packet in Suricata, this can be done by tweaking the fanout mode of AF\_PACKET.

## Chapter 8

# Conclusions and Future Work

Service chaining or Middlebox sequencing is a challenge because middleboxes modify the packet headers and makes it difficult to ensure service chaining. Existing approaches fail in the presence of mangling middleboxes or they require middlebox modifications. We have designed and implemented a system that can guarantee correct middlebox traversal, without any modifications required to the middlebox software. We make use of VLAN ID field of 802.1Q header for storing the contextual information (tag) for service chaining. Our specific contribution in this research is to exploit existing technology with carefully managed and configured middleboxes to solve the service chaining problem. Our approach is lightweight and works for any type of middleboxes. Evaluation was done using virtualized topology created with MiniNet, enabling Floodlight 0.90 as SDN controller, Open vSwitch 1.9.0 as SDN Switch and synthetic, libpcap based middleboxes. Evaluation shows that we can achieve service chaining even in the presence of mangling middleboxes. We found performance issues, SPI introduces significant delay and scalability issues, system does not scale in case number of more than six number of middleboxes. Performance and scalability issues can be improved in the future work.

As future work, performance and scalability issues will be solved by improving SPI interface and using IEEE 802.1ad respectively. Usage of IEEE 802.1ad are discussed in Section 7.5.1. We will further analyze the impact of using IEEE 802.1ad in real network. Implementation of kernel module to preserve PCP bits for incoming and outgoing packets, which gives the direction of flow. We will perform field trial of the system in live environment. Our system shows feasibility for integration of DPI, it will be worthwhile to implement an interface between DPI and SPI which supports L<sub>4</sub>-L<sub>7</sub> capability. Currently network configuration for middleboxes is done manually, we will integrate the *expect* script based module with the existing system for dynamically adding middleboxes in the network.

# References

- [1] Configure an Ethernet interface as a 802.1ad (QinQ) VLAN trunk. [http://www.microhowto.info/howto/configure\\_an\\_ethernet\\_interface\\_as\\_a\\_qinq\\_vlan\\_trunk.html](http://www.microhowto.info/howto/configure_an_ethernet_interface_as_a_qinq_vlan_trunk.html).
- [2] Y. Zhang, N. Beheshti, L. Beliveau, G. Lefebvre, R. Manghirmalani, R. Mishra, R. Patneyt, M. Shirazipour, R. Subrahmaniam, C. Truchan et al. StEERING: A software-defined networking for inline service chaining. In *Network Protocols (ICNP)*, 2013 21st IEEE International Conference on. IEEE, 2013 1–10.
- [3] 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 *Proc. USENIX NSDI*. 2014 .
- [4] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. SIMPLE-fying middlebox policy enforcement using SDN. In *ACM SIGCOMM Computer Communication Review*, volume 43. ACM, 2013 27–38.
- [5] ONF Specifications OpenFlow Switch Specification Version 1.0.0. <http://archive.openflow.org/documents/openflow-spec-v1.0.0.pdf> 2009.
- [6] Project Floodlight. <http://www.projectfloodlight.org/floodlight/>.
- [7] MiniNet. <http://mininet.org/>.
- [8] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review* 38, (2008) 69–74.
- [9] O. M. E. Committee et al. Software-Defined Networking: The New Norm for Networks. *ONF White Paper. Palo Alto, US: Open Networking Foundation* .
- [10] N. Feamster, J. Rexford, and E. Zegura. The Road to SDN. *Queue* 11, (2013) 20.
- [11] B. A. A. Nunes, M. Mendonca, X.-N. Nguyen, K. Obraczka, and T. Turletti. A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks. *IEEE Communications Surveys and Tutorials (Under Review)* .
- [12] B. Pfaff, J. Pettit, K. Amidon, M. Casado, T. Koponen, and S. Shenker. Extending Networking into the Virtualization Layer. In *Hotnets*. 2009 .
- [13] Open vSwitch. <http://openvswitch.org/> 2014.

- [14] Ryu. <http://osrg.github.io/ryu/>.
- [15] Trema. an open source modular framework for developing openflow controllers in ruby/c. <https://github.com/trema/trema> 2013.
- [16] POX. <http://www.noxrepo.org/pox/about-pox/>.
- [17] D. A. Joseph, A. Tavakoli, and I. Stoica. A policy-aware switching layer for data centers. *ACM SIGCOMM Computer Communication Review* 38, (2008) 51–62.
- [18] G. Detal, B. Hesmans, O. Bonaventure, Y. Vanaubel, and B. Donnet. Revealing middlebox interference with tracebox. In Proceedings of the 2013 conference on Internet measurement conference. ACM, 2013 1–8.
- [19] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda. Is it still possible to extend TCP? In Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference. ACM, 2011 181–194.
- [20] A. Gember, A. Krishnamurthy, S. S. John, R. Grandl, X. Gao, A. Anand, T. Benson, A. Akella, and V. Sekar. Stratos: A Network-Aware Orchestration Layer for Middleboxes in the Cloud. *CoRR* abs/1305.0209.
- [21] P. Quinn, R. Fernando, J. Guichard, S. Kumar, P. Agarwal, R. Manur, A. Chauhan, M. Smith, N. Yadav, B. McConnell, and C. Wright. Network Service Header. *Internet-Draft draft-quinn-nsh-03, IETF Secre-tariat* .
- [22] N. Ek. Ieee 802.1 p, q-qos on the mac level. *Apr* 24, (1999) 0003–0006.
- [23] D. Libes. Exploring Expect: a Tcl-based toolkit for automating interactive programs. ” O’Reilly Media, Inc.”, 1995.
- [24] OpenFlow 1.1+ support in Open vSwitch. <https://github.com/openvswitch/ovs/blob/master/OPENFLOW-1.1+.md>.
- [25] IEEE 802.1ad. [https://en.wikipedia.org/wiki/IEEE\\_802.1ad](https://en.wikipedia.org/wiki/IEEE_802.1ad).
- [26] Possible Suricata Improvements. <https://home.regit.org/2012/07/suricata-to-10gbps-and-beyond/>.
- [27] J. Blendin, J. Rückert, N. Leymann, G. Schyguda, and D. Hausheer. Position Paper: Software-Defined Network Service Chaining .
- [28] S. S. John and A. Akella. Active Switching: Packet Steering Flow Annotations. *arXiv preprint arXiv:1403.7115* .
- [29] M. Boucadair, C. Jacquenet, R. Parker, D. Lopez, P. Yegani, J. Guichard, and P. Quinn. Differentiated Network-Located Function Chaining Framework. *Internet-Draft draft-boucadair-network-function-chaining-02, IETF Secre-tariat* .
- [30] Z. Qazi, C.-C. Tu, R. Miao, L. Chiang, V. Sekar, and M. Yu. Practical and incremental convergence between sdn and middleboxes. *Open Network Summit, Santa Clara, CA* .

- [31] A. A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella. ElastiCon: an elastic distributed sdn controller. In Proceedings of the tenth ACM/IEEE symposium on Architectures for networking and communications systems. ACM, 2014 17–28.
- [32] Open vSwitch. <http://www.openvswitch.org/>.
- [33] Open Networking Foundation. OpenFlow switch specification version 1.3 2013.
- [34] A. Tirumala, F. Qin, J. Dugan, J. Ferguson, and K. Gibbs. Iperf: The TCP/UDP bandwidth measurement tool. *http://dast.nlanr.net/Projects* .
- [35] V. Jacobson, C. Leres, and S. McCanne. The tcpdump manual page. *Lawrence Berkeley Laboratory, Berkeley, CA* .

# Appendix A

## VLAN Configuration

### A.1 IEEE 802.1Q

1. Install the vlan package if it is not already installed:

```
sudo apt-get install vlan
```

2. Load the 8021q module into the kernel

```
sudo modprobe 8021q
```

3. To add vlan interface with vlan id 10 to the physical interface eth0

```
sudo vconfig add eth0 10
```

4. Assign an address to the new interface

```
sudo ip addr add 10.0.0.1/24 dev eth0.10
```

A static configuration can be added to `/etc/network/interfaces` file in Linux based server.

### A.2 IEEE 802.1ad (QinQ) [1]

To configure an Ethernet interface as an IEEE 802.1ad (QinQ) VLAN trunk we need to use *ip link* command, which supports both 802.1Q and 802.1ad instead of *vconfig*.

1. Select the required service VLAN from the service VLAN trunk

Let say we want to select VLAN 24 from interface eth0 and present it as eth0.24. This can be achieved using the following command:

```
ip link add link eth0 eth0.24 type vlan proto 802.1ad id 24
```

By default, the type vlan argument would create an 802.1Q VLAN tagged using an EtherType of 0x8100. The proto 802.1ad argument overrides this, causing the VLAN to be tagged using an EtherType of 0x88a8

2. Select the required customer VLAN from the service VLAN trunk.

Let say we want to select VLAN 371 from interface eth0.24 and present it as eth0.24.371. This can be achieved using the following command:

```
ip link add link eth0.24 eth0.24.371 type vlan proto 802.1Q id 371
```

The proto 802.1Q argument can be omitted, since this type of VLAN is the default, but it has been included here in the interests of clarity. The resulting VLAN will be tagged using an EtherType of 0x8100 (which is correct for a customer VLAN).