Alireza Ranjbar

Domain Isolation in a Multi-Tenant Software-Defined Network

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 15.4.2015

Thesis Supervisor:

Prof. Jukka Manner, Aalto University, Finland

Thesis Instructor:

Kristian Slavov M.Sc. (Tech.), NomadicLab, Ericsson Research, Finland

Aalto University
School of Electrical
Engineering

AALTO UNIVERSITY
SCHOOL OF ELECTRICAL ENGINEERING

ABSTRACT OF THE
MASTER'S THESIS

Author: Alireza Ranjbar

Title: Domain Isolation in a Multi-Tenant Software-Defined Network

Date: 15.4.2015 Language: English Number of pages: 10+97

Department of Communications and Networking

Professorship: Networking Technology Code: S-38

Supervisor: Prof. Jukka Manner, Aalto University, Finland

Instructor: Kristian Slavov M.Sc. (Tech.), NomadicLab, Ericsson Research, Finland

Software-Defined Networking (SDN) has evolved as a new networking paradigm to solve many of current obstacles and limitations in communication networks. The SDN technology is going to be implemented in multi-tenant environments like data centers where several customers, which are called "tenants", share network resources. In fact, the integration of SDN allows tenants in a shared network to have higher levels of control over available resources. While this approach has several advantages, the isolation between the tenants of a shared network becomes a vital factor which has not been discussed clearly so far.

This thesis discusses multi-tenancy and explains current isolation approaches in a multi-tenant SDN. For increasing isolation between tenants, this thesis proposes a scalable solution that provides traffic isolation, address space isolation, control isolation and performance isolation. In the new system architecture, tenants are not limited to their own networks and they are able to make interaction with each other and external resources. Indeed, while tenants are isolated from each other, they are allowed to access special services offered by other tenants or external services outside of a shared network.

The evaluation of the prototype proves that the new architecture provides a high level of isolation in a multi-tenant SDN and it is scalable enough to be implemented in large networks with millions of tenants.

Keywords: Domain, Multi-Tenancy, Policy, SDN, OpenFlow, Traffic Isolation, Packet Rewriting, Address Space Isolation, Control Isolation, Monitoring

Acknowledgements

I am grateful to all of those who supported me throughout this thesis work. I would like to thank my supervisor, Prof. Jukka Manner for reviewing my thesis and giving me valuable feedbacks. Also, I would like to give my special thanks to Prof. Tuomas Aura who guided me throughout this thesis work. I deeply appreciate his constant support and excellent suggestions during this work.

I would like to show my gratitude to my instructor, Kristian Slavov, for supporting me throughout this work. Special thanks to Bengt Sahlin for providing me an opportunity to work on my field of interest in Ericsson Research, NomadicLab. Also, I would like to thank all my colleagues from NomadicLab particularly Patrik Salmela, Abu Shohel Ahmed, Oscar Novo, Miika Komu and Kazi Wali Ullah for guiding me and creating such a friendly and enjoyable working environment.

Last but not least, I would like to thank my family who supported me morally all through my life.

Espoo, April 2015

Alireza Ranjbar

Contents

Abstract ii
Acknowledgements iii
Contents iv

1 Introduction **1**
1.1 Research Goals 2
1.2 Contributions 2
1.3 Limitations 3
1.4 Structure of the Thesis 3

2 Software Defined Networking **4**
2.1 SDN Architecture 4
2.1.1 Application Plane 4
2.1.2 Controller Plane 6
2.1.3 Data Plane 6
2.1.4 Management 7
2.2 OpenFlow Specification 8
2.2.1 Flow Table 8
2.2.2 Matching 10
2.2.3 OpenFlow Protocol 11
2.2.4 Secure Channel 12
2.3 Chapter Summary 12

3 Isolation in a Multi-Tenant SDN **13**
3.1 Multi-Tenancy in SDN 13
3.1.1 SDN Provider with Connected Tenant Applications 13
3.1.2 SDN Provider with Directly Connected Tenant Controllers . . 14
3.1.3 SDN Provider with Non-Recursively Connected Tenant Controllers 15
3.1.4 SDN Provider with Recursively Connected Tenant Controllers 16
3.2 Current Isolation Techniques in a Multi-Tenant SDN 16
3.2.1 Slicing 17
3.2.2 Encapsulation 17
3.2.3 Packet Rewriting 17
3.3 Available Multi-Tenant SDN Solutions and their Isolation Approaches 19
3.3.1 FlowN 19
3.3.2 Splendid Isolation 20
3.3.3 FlowVisor 20

3.3.4 AutoSlice . . . 22
3.3.5 OpenVirtex . . . 23
3.4 Chapter Summary . . . 23

4 Proposed System Design **25**
4.1 Design Goals . . . 25
4.2 Design Pattern . . . 26
4.3 Principles . . . 27
4.3.1 SDN Provider . . . 27
4.3.2 Tenant . . . 27
4.3.3 Tenant Network Domain . . . 27
4.3.4 Policy . . . 28
4.4 Architectural Components . . . 28
4.4.1 Northbound Interface . . . 29
4.4.2 Service Manager . . . 29
4.4.3 DHCP Server . . . 30
4.4.4 ARP Handler . . . 30
4.4.5 Isolation Manager . . . 30
4.4.6 Routing Manager . . . 31
4.4.7 Monitoring . . . 33
4.5 Communication Patterns in the System Architecture . . . 33
4.5.1 Intra-Tenant Communications . . . 33
4.5.2 Inter-Tenant Communications . . . 36
4.5.3 External Communications . . . 40
4.6 Chapter Summary . . . 44

5 Implementation of the Prototype **45**
5.1 Implementation Environment . . . 45
5.2 Implementation of the Northbound Interface . . . 46
5.3 Implementation of the Service Manager . . . 49
5.4 Domain Discovery Mechanism . . . 51
5.4.1 Domain Discovery for Northbound Requests from Tenants . . . 51
5.4.2 Domain Discovery for Flow Requests from the Data Plane . . . 51
5.5 Implementation of the DHCP Server . . . 53
5.5.1 Host Detection with the DHCP Server . . . 53
5.5.2 Expired Leased Addresses . . . 53
5.6 Isolation Mechanism in the Prototype . . . 54
5.6.1 Rule Matching . . . 54
5.7 Forwarding at the Data Plane . . . 56
5.8 Implementation of Monitoring . . . 61
5.8.1 sFlow-RT . . . 62
5.8.2 Protection Application . . . 62
5.8.3 Monitor Manager . . . 62
5.9 Implementation Issues and Challenges . . . 62
5.10 Chapter Summary . . . 63

6 Evaluation and Experimental Results **64**
6.1 Analysis of Isolation Enforcement 64
6.1.1 Traffic Isolation 64
6.1.2 Address Space Isolation 64
6.1.3 Control Isolation 65
6.1.4 Performance Isolation 65
6.2 Functional Testing 65
6.2.1 Test Scenario 1: Isolation in Intra-Tenant Communications . . 66
6.2.2 Test Scenario 2: Isolation in Inter-Tenant Communications . . 67
6.2.3 Test Scenario 3: Isolation in External Communications 69
6.2.4 Test Scenario 4: Performance Isolation 71
6.3 Scalability 73
6.3.1 Scalability at the Core Network 73
6.3.2 Scalability in IP Address Assignment 74
6.4 Control Traffic Overhead 75
6.4.1 Size of OpenFlow Messages 75
6.4.2 Control Overhead in the Prototype 79
6.5 Latency of Rule Matching Process 80
6.6 Discussion 81
6.7 Chapter Summary 82

7 Conclusion **84**

Appendices **92**

A List of REST APIs **93**

List of Figures

2.1 SDN Architecture 5
2.2 Service Abstraction Layer in the OpenDaylight controller 7
2.3 Simplified diagram of flow matching in an OpenFlow-enabled device . 10

3.1 SDN provider with application tenants 14
3.2 SDN provider with SDN tenants connected directly to the data plane 15
3.3 SDN Provider with Non-Recursively Connected Tenant Controllers . 15
3.4 SDN Provider with Recursively Connected Tenant Controllers 16
3.5 Packet rewriting at the controller plane 18
3.6 Packet rewriting at the data plane 18
3.7 FlowN Architecture 19
3.8 Slicing approach (topology slicing) in FlowVisor 21
3.9 Architecture of AutoSlice 22
3.10 Isolation at the controller and data planes in OpenVirtex 23

4.1 Design pattern 27
4.2 System Architecture 29
4.3 Routing in the System Architecture 32
4.4 The operational flow for the intra-tenant communication 35
4.5 An example of the service advertisement for inter-tenant communications 36
4.6 The operational flow for the inter-tenant communication 39
4.7 The operational flow for the communication to the external network . 41
4.8 The operational flow for the communication from the external network 43

5.1 The operational states of a bundle in the OSGi framework 46
5.2 Domain management in the service manager 50
5.3 Implemented algorithm for finding the destination domain 52
5.4 Host detection using the DHCP server 54
5.5 Proactive rule installation at the core network 57
5.6 Reactive forwarding using the routing manager at the edge of the network 59
5.7 Message diagram for monitoring process in the prototype 61

6.1 Test network for intra-tenant communications 66
6.2 A packet captured between end-hosts A-1 and A-2 at the core network 67
6.3 A packet captured between end-hosts B-1 and B-2 at the core network 67
6.4 A packet captured between end-hosts C-1 and C-2 at the core network 67
6.5 Test network for inter-tenant communications 68

6.6 A captured packet on the link between end-host B-1 and edge switch S1 . . . 68
6.7 A captured packet on the link between edge switch S1 and core switch S3 . . . 69
6.8 A captured packet on the link between edge switch S3 and end-host A-1 . . . 69
6.9 The output of Iperf server on the end-host A-1 . . . 69
6.10 Test network for external communications . . . 70
6.11 A packet captured on the link between switch S1 and S2. The source of the packet is end-host A-1 . . . 71
6.12 A packet captured on the link between switch S1 and S2. The source of the packet is end-host B-1 . . . 71
6.13 A packet captured on the link between the core switch S2 and the edge switch S1. The destination of the packet is end-host A-1 . . . 71
6.14 A packet captured on the link between the core switch S2 and the edge switch S1. The destination of the packet is end-host B-1 . . . 71
6.15 Testing the prototype without the monitoring module . . . 72
6.16 Testing the prototype by enabling the monitoring module . . . 72
6.17 A sample network for testing the scalability . . . 73
6.18 The number of flow entries at the ingress edge switch and the core switch . . . 74
6.19 Control messages (OpenFlow) for processing a new flow request in our prototype . . . 79
6.20 The amount of control traffic based on the number of new flow requests 80
6.21 The latency of rule matching technique for installing forwarding rules 81

7.1 Multi-location approach for future work . . . 86

List of Tables

2.1 An example of the REST API definition in the OpenDaylight controller . . . 5
2.2 Match fields in the flow table . . . 9

5.1 REST request for registering a tenant . . . 47
5.2 REST request for setting a subnet . . . 48
5.3 REST request for creating a policy group . . . 48
5.4 REST request for adding rules to a policy group . . . 48
5.5 REST request for service advertisements . . . 49
5.6 REST request for setting network configurations . . . 49
5.7 Forwarding rules at the core of the network . . . 56
5.8 Match fields of forwarding rules at the edge of the network . . . 58
5.9 Actions for forwarding rules at the edge of the network (the source and destination end-hosts are not connected to the same edge switch) 60
5.10 Actions for forwarding rules at the edge of the network (the source and destination end-hosts are connected to the same switch) . . . 60

6.1 The number of available IP addresses and port numbers for different types of communications . . . 75
6.2 Size of header fields for all transmissions between the controller and the switches . . . 76
6.3 Size of Packet-in message . . . 76
6.4 Size of Packet-out message . . . 77
6.5 Size of Flow-Mod message for different types of flows . . . 78
6.6 Size of Barrier messages . . . 78
6.7 Size of Flow-Removed message . . . 78
6.8 Total control overhead for handling a new flow . . . 79

Abbreviations

ACK	Acknowledgment
API	Application Programming Interface
ARP	Address Resolution Protocol
CFI	Canonical Format Identifier
CPU	Central Processing Unit
CPX	Controller Proxy
CRC	Cyclic Redundancy Check
DHCP	Dynamic Host Configuration Protocol
HTTP	Hypertext Transfer Protocol
ICMP	Internet Control Message Protocol
ID	Identifier
IP	Internet Protocol
JAX-RS	Java API for RESTful WEB Services
JSON	JavaScript Object Notation
LLDP	Link Layer Discovery Protocol
LXC	Linux Container
MAC	Media Access Control
MM	Management Module
MPLS	Multiprotocol Label Switching
NAT	Network Address Translation
NIC	Network Interface Card
OSGi	Open Services Gateway initiative
PCP	Priority Code Point
QOS	Quality of Service
REST	Representational State Transfer
RFC	Request for Comments
SAL	Service Abstraction Layer
SDN	Software-Defined Networking
sFlow	sampled Flow
sFlow-RT	sFlow Real Time
SNMP	Simple Network Management Protocol
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TLS	Transport Layer Security
ToS	Type of Service
UDP	User Datagram Protocol
VID	Virtual Local Area Network Identifier
VIF	Virtual Interface
VLAN	Virtual Local Area Network
VM	Virtual Machine
VXLAN	Virtual Extensible LAN
XML	Extensible Markup Language

Chapter 1

Introduction

During the past few years, Software-Defined Networking (SDN) has emerged as a new networking paradigm by decoupling the control layer in the network from the forwarding layer. The separate centralized controller has led to better virtualization and programmability in today's networks. In the architecture of SDN, applications send forwarding requests to the controller by the Application Programming Interface (API) commonly called northbound API. Consequently, the controller installs the necessary forwarding rules in the network elements in the data plane. The OpenFlow protocol has been accepted as the first standard protocol for communication between the controller plane[1] and the data plane. The use of OpenFlow allows the SDN controller to control different networking devices from different vendors with an open standard interface.

Both the academia and the industry are investing in the development of SDN. Ericsson, Google, Juniper and Cisco are few of the vendors that have contributed to the development of SDN and programmable networks. Despite all the advantages of SDN, the implementation of multi-tenancy with SDN is still a research problem. Multi-tenancy is an appealing feature of todays networks which allows different customers and organizations share the same resources of a SDN network while they are logically isolated from each other. Since the resources are shared, isolation becomes one of the most important requirements of multi-tenant SDN networks [12,14]. In SDN, we need isolation at both of the controller and data planes. Consequently, a combination of different isolation techniques should be used to provide effective isolation at both layers. On the other hand, isolation is a general concept and it needs to be explored in every aspect. For instance, we can have traffic isolation between tenants which limits tenants to only accessing their own traffic, or we may need performance isolation which restricts tenants from affecting the performance of other tenants in a shared network.

Several solutions have been implemented for bringing multi-tenancy into SDN networks. FlowVisor [39] is one example which provides multi-tenancy by making slices of a shared network for different tenants. Another example is FlowN [38], which allows to define several tenant networks by using encapsulation technique. However, there is not any comprehensive research about the possible configurations for a multi-tenant SDN. Likewise, different isolation approaches in a multi-tenant SDN have not been discussed extensively so far. Therefore, during this thesis, we

[1] According to [3], the control plane in SDN is called the controller plane

investigate the possible configurations for a multi-tenant SDN and then, we explore the different isolation techniques that have been used in the available multi-tenant SDN solutions. In addition, we propose a new architecture for providing higher levels of isolation in a multi-tenant SDN. While our solution provides isolation for tenants, it increases interoperability between tenants and it allows tenants to connect to the available services outside of a shared network.

1.1 Research Goals

This thesis mainly focuses on isolation in a multi-tenant SDN. The goals of the thesis are:

- Defining and explaining possible configurations in a multi-tenant SDN. At first, the concept of multi-tenancy in SDN is discussed and then the possible configurations between tenants and a service provider are explained.
- Surveying the current isolation techniques in a multi-tenant SDN to understand the design space. This includes a motivation for isolation in a multi-tenant SDN and an analysis of the state-of-the-art isolation techniques.
- Designing a new architecture to enhance isolation in a multi-tenant SDN. Our solution covers various isolation requirements in a multi-tenant SDN.
- Implementing a prototype of the designed architecture and evaluating the functionality and the performance of the prototype. The prototype is implemented on the OpenDaylight controller.

1.2 Contributions

The first contribution of this thesis is to discuss possible configurations in a multi-tenant SDN network. We consider four different types of interaction between tenants and a service provider in a SDN network. Furthermore, we explain the isolation approaches in existing multi-tenant SDN solutions.

The next contribution of this thesis is to propose and implement a scalable solution which provides traffic isolation, address space isolation, control isolation and performance isolation between the tenants of a shared network. The traffic isolation provides isolation between different flows at the data plane and the address space isolation allows to have overlapped addresses between tenants. With the control isolation, we allow tenants to securely join the network and make their own configurations. Moreover, we have integrated the monitoring feature in our architecture which leads to the performance isolation between tenants. Additionally, our solution increases the isolation level between tenants by removing the broadcast ARP requests and allowing tenants to join the network using a DHCP server.

While our solution provides isolation, it allows tenants to make connection with offered services by other tenants or external services outside of a shared network. Indeed, tenants are not limited to their own networks and we have provided a possibility for tenants to interact with each other and with the external resources.

Although we have implemented our architecture on the OpenDaylight controller, it can be extended to other SDN controller platforms.

1.3 Limitations

Our architecture is mainly based on the OpenFlow 1.0 specification. However, the OpenFlow protocol is not mature and has several drawbacks. OpenFlow does not include features to configure the network elements in the data plane from the controller plane, so we need to configure the devices manually. Moreover, it does not provide effective QOS features, and we rely on available capabilities. OpenFlow 1.3, supports more QOS features, and the OpenDaylight controller implements this version, but it is not yet supported by the Open vSwitch at the time of writing of this thesis. Our approach is also based on the Open vSwitch features[2], and so it is not possible to change it to the other virtual switch implementations.

1.4 Structure of the Thesis

In chapter 2, we explain all details about SDN that are used in the subsequent chapters. The architecture of SDN networks will be discussed and the OpenFlow protocol will be explained in detail in this chapter. In the next chapter, we introduce the possible configurations for a multi-tenant SDN. Moreover, the isolation approaches in existing multi-tenant SDN solutions will be explained. In chapter 4, we explain our new architecture and the functionality of our solution. The details of each component in our architecture will be discussed in this chapter. We start to explain about the implementation of our prototype in chapter 5. The structure of our prototype will be explained in detail in this chapter. In chapter 6, we evaluate our approach for the different isolation requirements and we run several test scenarios for testing the functionality of our solution. Also, we evaluate the solution in terms scalability, overhead and latency and we discuss about the the advantages and the disadvantages of our proposed solution in this chapter. In the last chapter, we summarize our work based on the topics discussed throughout the thesis and make conclusion in this chapter. Additionally, the possible directions for future works will be introduced in the last chapter.

[2]We use sFlow with the Open vSwitch for monitoring the traffic at the data plane.

Chapter 2

Software Defined Networking

Today's networks are growing in size, the amount of traffic and more complex features. While available technologies are limited to address the current limitations, SDN has emerged as a new networking paradigm to solve many of current obstacles in today's networks. SDN removes the complexity and provides an abstract view of the network with a central control point. In this way, we have a central controller over the network which eases the management of networking resources. The programmable and the open nature of SDN leads to a high level of innovative and independent developments. Also, a centralized view of the network allows the controller to define uniform security policies over network resources [1]. In this chapter, at first we explain about the architecture of SDN and different components in the SDN architecture. Then, we continue this chapter by explaining the use of OpenFlow protocol in SDN and the definition of different messages in this protocol.

2.1 SDN Architecture

The SDN architecture includes the application plane, the controller plane, the data plane and the management. The architecture of SDN is depicted in Figure 2.1. The application plane communicates with the controller plane through the northbound API and the controller plane connects to the data plane with the southbound API [3, 1]. The first standard protocol for SDN networks is OpenFlow [2] protocol which is used as a southbound API for connecting the controller plane to the data plane.

2.1.1 Application Plane

The application plane consists of one or more applications that send their requests to the controller with the northbound interface. Network services are defined at the controller plane and they provide APIs for applications to use the available functionalities without considering the complexity of the underlying network. Consequently, it is possible to offer several services for different purposes through the northbound APIs [3]. The northbound interface is defined by open APIs to provide the common interface between different vendors.

One of the popular APIs that has been implemented widely is Representational State Transfer API (REST API) [20]. The SDN controller uses the REST API to provide an interface for communication with the application plane. By using the

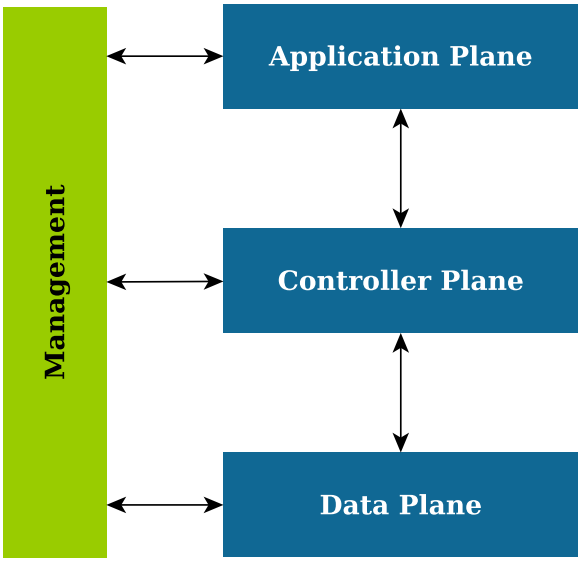

Figure 2.1: SDN Architecture

REST API, the SDN controller is able to hide the details of the network components and provides the main roles and major features. It shows an abstract view of available services to the application layer using the REST API and hides the complexity of the network services [20]. In the Opendaylight controller, the REST API can be defined in the JSON or XML format. Table 2.1 shows an example of the REST API definition for the flow configuration in JSON format after sending a GET command to the OpenDaylight controller [17].

Table 2.1: An example of the REST API definition in the OpenDaylight controller

```
{
   "flowConfig": {
      "installInHw": "true",
      "name": "flow",
      "node": {
         "type": "OF",
         "id" : "00:00:00:00:00:00:00:01"
       }
      "ingressPort": "1",
      "priority": "500",
      "etherType": "0x800",
      "nwSrc":"10.10.1.1",
      "actions": [
          {"OUTPUT=2"}
       ]
    }
}
```

2.1.2 Controller Plane

The controller plane provides a logically centralized view of the underlying network. In fact, the SDN controller at the controller plane becomes the network intelligence to control the entire network resources from a central point. The SDN controller works with the abstract view of the network resources and since the controller is implemented as a software solution, it is possible to implement a variety of services for different applications based on the available network resources. Today, we have many controllers operating on this layer such as NOX, POX, Opendaylight and Floodlight. The internal functional processes of these controllers are different but most of them allow programmers to define new services based on their structural languages such as Java, Python or C/C++ [3].

At the following, we briefly explain the functionality and the structure of the OpenDaylight controller as an example to clarify the operation of the SDN controllers.

OpenDaylight

OpenDaylight is a joint project between various vendors to provide a reference framework and an open source controller. It allows programmers to work on an open source project in an open community to develop business and technical subjects. According to [16], the mission goal of this group is: *"Facilitate a community-led, industry-supported open source framework, including code and architecture, to accelerate and advance a common, robust Software-Defined Networking platform"*.

OpenDaylight is a modular controller based on Service Abstraction Layer (SAL). Indeed, the modular design by SAL leads to the definition of rich services for modules and applications. Network services are deployed in SAL based on the features presented by the southbound plugins. Moreover, OSGi [28] is used as a component based framework to dynamically develop network services as new modules on the controller. Generally, SAL provides services like Device Discovery or Packet Data and OSGi uses these services for making new modules. Figure 2.2 illustrates the architecture of the OpenDaylight controller based on SAL [17].

2.1.3 Data Plane

The data plane consists of one or more network elements. Each network element includes a set of traffic processing and forwarding functions. This layer is responsible for providing appropriate virtualization, security, connectivity, quality and availability for processing the customer's traffic. Each network element contains a virtualization component named the virtualizer. The virtualizer on the network elements is responsible for providing an abstract view of resources for the controller and enforcing the policy. The data plane receives instructions and forwarding requests from the controller and it cannot process the traffic by itself. However, it is possible that the controller delegates specific capabilities to the data plane [3].

At the following, we briefly explain Open vSwitch as an example of network elements working at the data plane.

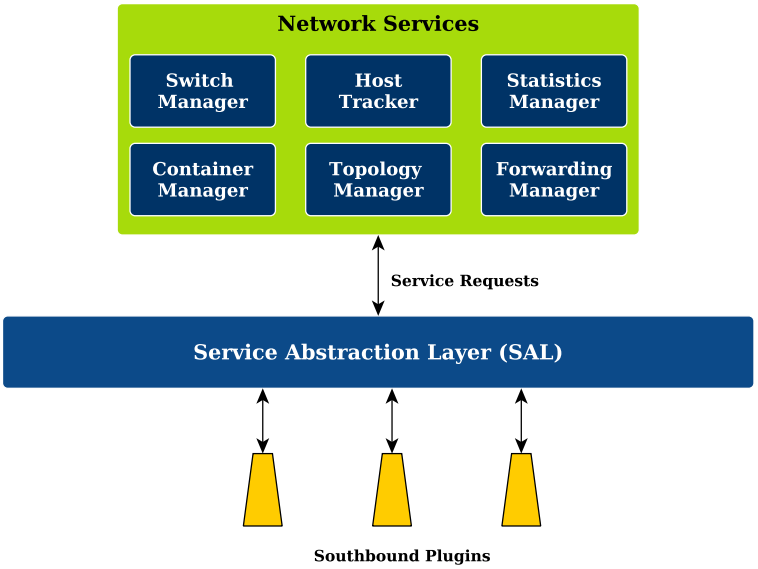

Figure 2.2: Service Abstraction Layer in the OpenDaylight controller

Open vSwitch

The Open vSwitch [8] is a software solution that provides connection between VMs and physical devices. Unlike physical hosts that communicate with NIC cards, VMs use Virtual Interfaces (VIFs) and virtual switches provide the connection between VIFs and physical interfaces. Furthermore, the Open vSwitch has the ability to be controlled by the SDN controller using the OpenFlow protocol. The operation of the Open vSwitch is based on two fundamental elements: kernel resident "fast path" and userspace "slow path". Forwarding operations including packet look up, counting and forwarding are implemented by the fast path. Forwarding logic operations such as: MAC address learning, load balancing and configurations are implemented with the slow path. Additionally, management protocols like OpenFlow are implemented in the slow path [33].

2.1.4 Management

The Management is responsible for all management functionalities in all layers. As it is demonstrated in Figure 2.1, this layer is connected to the application plane, the controller plane and the data plane. Generally, the management layer is responsible for implementing functionalities that are not supported by other layers. Also, the management is responsible for tasks that other layers are prohibited from implementing, for example, according to the policy. The management consists of SDN specific tasks such as defining the policies between the service provider and clients and the configuration and initialization of separate SDN units. Moreover, the responsibilities of this layer include the arrangement of information about the handoff points of the data plane, identification, credentials and protocols between physical and logical SDN units [3].

2.2 OpenFlow Specification

OpenFlow [1, 2] is the first standard protocol for interaction between the controller plane and the data plane. It allows the controller and the network elements of the data plane to transfer the configuration and statistical messages directly. The OpenFlow benefits from the concept of flows and in OpenFlow, each flow is processed according to the pre-defined match rules. Since OpenFlow is based on per flow analysis, it provides high granular control which leads to react in real-time to changes in the applications, users and sessions.

2.2.1 Flow Table

The flow table is one of the major parts of the OpenFlow-enabled switches. Each entry in the flow table of OpenFlow 1.0 includes a set of fields. The most relevant fields are[1]: *Header (Match) Fields*, *Counters*, *Timeouts*, *Priority* and *Actions*. At the following, we explain each field briefly.

Header (Match) Fields

An incoming packet to the switch matches according to the header fields illustrated in Table 2.2. Each of these 12-tuples can be a specific or ANY value. In case of choosing ANY as a value for one field, it matches all possible values [58].

Counter

Counter field indicates statistics information for each table, flow, port and queue on the switch. This field includes the amount of traffic (in bytes) and the number of received, transmitted and dropped packets. Also, it includes information about the possible errors like CRC, overrun and frame errors.

Timeouts

Each flow in the flow table has an Idle and Hard timeout. The idle timeout declares the time that the flow can be in the inactive state and after this period of inactivity, the flow should be removed. The hard timeout declares the maximum time that the flow is placed in the flow table and after that period, even if the flow is still active, it should be removed.

Priority

In the OpenFlow-enabled switches, the flows match the forwarding rules based on the priority of the forwarding rules in the flow table. The forwarding rules with the higher priority will match first.

[1]Depending on the implementation of the OpenFlow 1.0, these fields might be different. However, in this part we introduce the most relevant details to our work

Table 2.2: Match fields in the flow table

Field	Description	Layer
Ingress Port	Incoming port for the frame	Physical
Ether source	Source MAC address	Data Link
Ether dst	Destination MAC address	
Ether type	Encapsulated Protocol in the payload. For IPv4, it is 0x0800	
VLAN id	VID field in the VLAN tag	
VLAN priority	PCP field in the VLAN tag	
IP src	Source IP address	Network
IP dst	Destination IP address	
IP proto	Transport layer protocol	
IP ToS bits	Type of Service field	
TCP/UDP src port	Source transport port (or ICMP type field)	Transport
TCP/UDP dst port	Destination transport port (or ICMP code field)	

Actions

The Action field indicates a list of actions related to a flow. The list of actions can be empty or it might include a set of actions to be implemented. If the list is empty, then the packet is dropped. If the list includes a set of actions, then they will be implemented according to the order in the list. Some of the possible actions are explained at the following [2]:

- *Forward:* OpenFlow supports forwarding actions for the flows on the physical or virtual switches. The forwarding actions may specify to send the packet to the SDN controller or sending the packet out of a specific switch port. It is also possible to flood the packets out of all the switch ports.
- *Enqueue:* This action is used to place a packet in a specific queue on the switch port.
- *Drop:* This action is used to drop a flow. By default, if the action list is empty, the flow will be dropped.

- *Modify-Field:* This option is used to change the packet header fields. The OpenDaylight controller supports the following actions [15] for modifying the header fields:
  - PopVlan: Pop the VLAN tag from the packet.
  - PushVlan: Push the VLAN tag to the packet.
  - SetVlanCfi: Set Cfi value in the VLAN field.
  - SetVlanId: Set VLAN Id (VID) value in the VLAN field.
  - SetVlanPcp: Set Pcp value in the VLAN field.
  - SetDlDst: Set the destination MAC address for the packet.
  - SetDlSrc: Set the source MAC address for the packet.
  - SetDlType: Set the ethertype for the packet.
  - SetNwDst: Set the destination IP address for the packet.
  - SetNwSrc: Set the source IP address for the packet.
  - SetNwTos: Set the Type of Service (TOS) field in the packet header.
  - SetTpDst: Set the destination transport port number for the packet.
  - SetTpSrc: Set the source transport port number for the packet.

2.2.2 Matching

When an OpenFlow-enabled switch receives a new flow, it checks the packet headers with the existing forwarding rules in the flow table. If there is not any matching entry for the new flow, the switch generates a packet-in event and forwards the first packet of the received flow to the controller to decide about the actions for the flow. Consequently, the controller checks the packet-in and installs the rules on the switch for the received flow. From this point, the switch applies actions to the packets that match the rule in the flow table. Figure 2.3 demonstrates this process. This type of flow installation on the OpenFlow-enabled switches is called *reactive* flow installation. However, it is possible to implement *proactive* flow installation. In this case, the controller installs proper forwarding rules for possible flows on the switch before the switch receives a new flow. As a result, the switch forwards the packets according to the pre-installed rules [31].

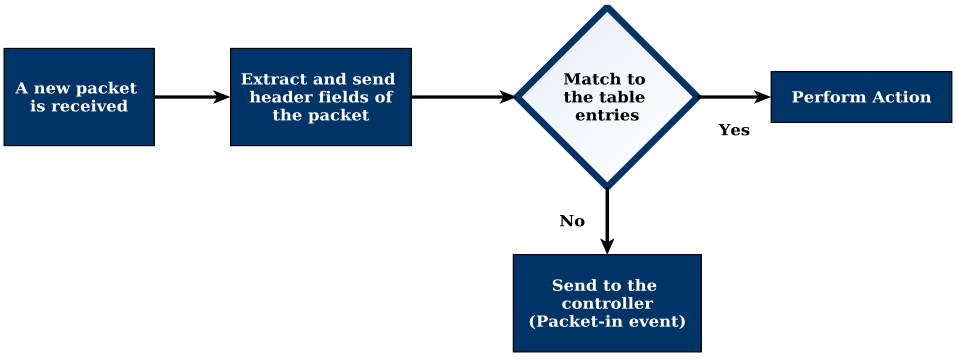

Figure 2.3: Simplified diagram of flow matching in an OpenFlow-enabled device

2.2.3 OpenFlow Protocol

The OpenFlow protocol [2] uses different types of messages for interaction between the switches and the SDN controller. At the following, we explain in brief these messages and their functionalities.

Controller-to-Switch:
The controller sends the following messages to the switch. These messages may not need a reply from the switches.

- Features: The feature messages include the supported functionalities by the switches. The controller initiates these messages and the switch must respond to the request message from the controller.
- Configuration: The controller may set configurations on the switch with this type of message. Additionally, using these messages, it is possible for the controller to query the configurations of the switch.
- Modify-State: These messages are used by the controller to add, remove or modify the entries in the switch flow tables. It should be noted that this type of messages are known as *flow-mod* messages.
- Read-State: The controller sends these messages to query the statistics data for the flow entries, ports and flow tables of the switch.
- Send-Packet: The controller uses these messages to forward a packet out of a particular switch port. Also, this type of messages are known as *packet-out* messages.
- Barrier: The barrier messages might be used as a request/reply message to assure all sent messages have been received and all requested operations have been implemented correctly.

Asynchronous:
The switch initiates and sends these messages regardless of receiving any request from the controller.

- Packet-in: The packet-in messages are generated when a switch receives a new flow request and it cannot find any matching entry for the new flow. These packet-in messages are forwarded to the controller to decide about the new flow request. If the switch supports buffering and has enough memory for buffering, the packet-in includes a fraction of the packet (128 bytes by default) and a buffer ID. The buffer ID is used for forwarding the packet after receiving a response from the controller. In case the switch does not support buffering, the whole packet is forwarded to the controller.
- Flow-removed: The flows might be removed, for instance, because of timeouts. As a result, the flow-removed is used by the switch to inform the controller about the removed flows.

- Port-status: If the status of the switch port changes, the switch initiates this message and sends it to the controller.
- Error: In case of any problems, the switch sends error messages to the controller.

Symmetric:
These messages are transferred in both directions regardless of receiving any request from the controller or the switches.

- Hello: These messages are used at the starting point of the connection between the controller and the switch and they are transferred in both directions.
- Echo: Either the switch or the controller can send echo request messages to understand link properties like bandwidth and latency. The other side of the connection should respond to the request.
- Vendor: Vendor messages are used by vendors to provide additional features in OpenFlow messages.

2.2.4 Secure Channel

The secure channel is established between the controller and the switches at the data plane. SSL/TLS [25] is a secure protocol that has been proposed for the encryption of messages between the controller and the data plane. While the security of this channel is important, because of the configuration and technical issues, there is lack of interest in the implementation of TLS on this channel in today's products [32]. It should be noted that the OpenDaylight controller supports TLS but it is not activated by default.

2.3 Chapter Summary

This chapter presents an overview of the SDN architecture. The SDN architecture consists of the application plane, the data plane, the controller plane and the management. The applications from the application plane interact with the SDN controller by using the northbound interface and the SDN controller controls the switches at the data plane using the southbound protocol. The OpenFlow protocol is the first southbound protocol that has been implemented in the SDN networks and it makes it possible to program the switches reactively or proactively by installing forwarding rules. OpenFlow defines each forwarding rule with a set of fields including the match field, actions, counters, priority and timeouts.

Chapter 3

Isolation in a Multi-Tenant SDN

Today's networking trend is changing to offer more shared services. An example of recent technologies is the cloud computing which provides a possibility for different customers to share resources. The emergence of SDN takes this feature one step further by allowing the customers of a shared network to decide about their own traffic and route it in the network. While SDN effectively gives a higher level of functionality to the customers of a shared network, there should be a way to limit and restrict the customers to their own resources. In this chapter, we want to explore the possibility of multi-tenancy in SDN and digging into the existing isolation approaches in such environments. We start this chapter by explaining multi-tenancy in SDN network and then we focus on isolation approaches in a multi-tenant SDN. In the last section of this chapter, we discuss current multi-tenant solutions and their contributions to provide isolation in a multi-tenant SDN.

3.1 Multi-Tenancy in SDN

Multi-tenancy allows different customers to share the same infrastructure and resources and it provides the efficient way of maintenance and management. As a result, this model leads to a significant improvement in the hardware utilization and reduction in the operational costs. We define a tenant as a customer or organizational entity which rents the resources of a SDN provider [35, 36, 59]. Based on our definition, in a multi-tenant SDN, tenants should be able to communicate with the controlling services offered by the SDN provider to control their own networks.

Tenants are able to connect to the SDN provider in different ways. At the following, the possible configurations for a multi-tenant SDN are explained.

3.1.1 SDN Provider with Connected Tenant Applications

In this case, all SDN services and network resources are under control of a single provider which controls and configures the network elements in the data plane. The provider offers a possibility to tenants to connect to the SDN controller for configuring network resources. In this configuration, the provider may offer additional features to the tenants or restrict the access of tenant applications to specific resources or services [3]. Figure 3.1 demonstrates this configuration.

Currently, the connection between tenant applications and the SDN controller is possible in two different ways: *Northbound APIs* [18] and *Containers* [38]. The northbound interface provides APIs for connecting applications to the controller. Most of the available SDN controllers support the northbound APIs for applications and it is part of the SDN architecture. Alternatively, it is possible to run the applications using the containers. The container provides a space for an application to be running on the SDN controller. As a result, the controller system is shared between tenant applications. An example of these containers is Linux Containers (LXC). LXC is a lightweight virtualization mechanism that allows running multiple copies of Linux operating system on a single machine. The details of Linux containers are explained in [10].

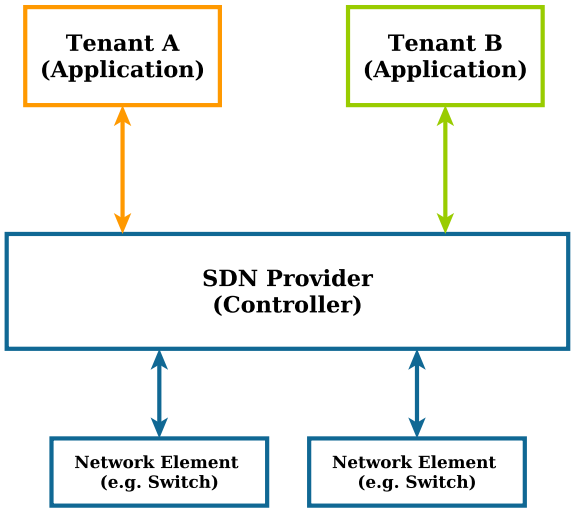

Figure 3.1: SDN provider with application tenants

3.1.2 SDN Provider with Directly Connected Tenant Controllers

In this configuration, the SDN provider allows different tenants to connect directly to the underlying network. The SDN provider configures and controls the network using the master controller and offers a virtual network to each of tenants. The virtual network is an abstract of the offered resources and ports of the network elements in the data plane. After assigning resources by the SDN provider, tenants can use their own controllers to connect directly to the network elements [3, 46, 47]. Figure 3.2 illustrates this configuration.

It should be noted that according to the SDN architecture [3], this configuration is not recommended since it exposes the underlying network and creates a vulnerability point in the network.

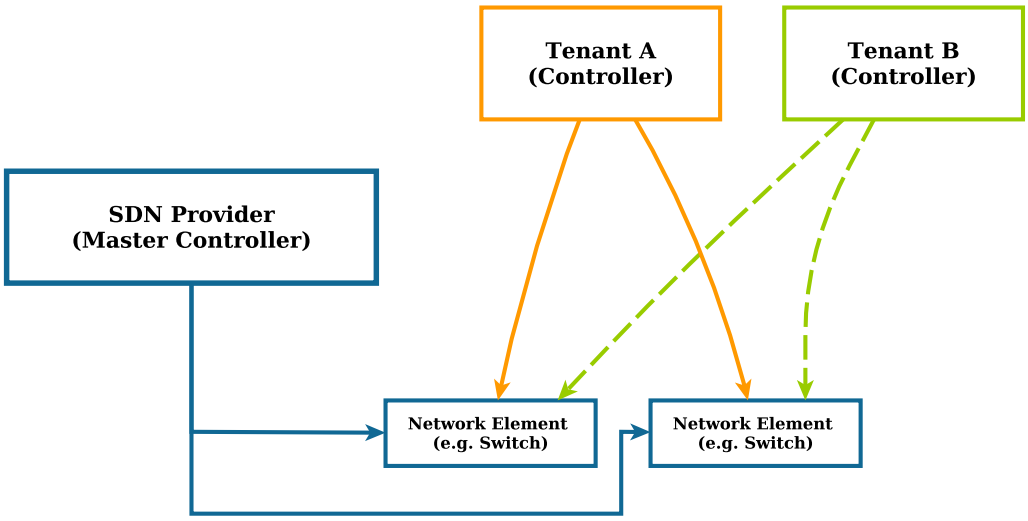

Figure 3.2: SDN provider with SDN tenants connected directly to the data plane

3.1.3 SDN Provider with Non-Recursively Connected Tenant Controllers

In this configuration, the SDN provider allows tenants to connect their own controllers to the network through another controller (or hypervisor). In fact, the SDN provider assigns network resources to tenants and tenants are able to access the resources through another controller provided by the SDN provider. According to Figure 3.3, the SDN provider offers an abstract of resources through the controller and tenants are connected to the SDN provider controller with their own controllers [3, 39, 49].

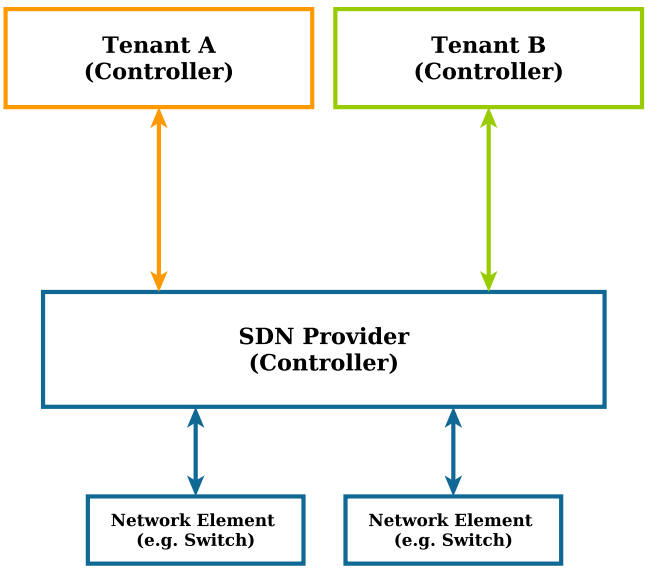

Figure 3.3: SDN Provider with Non-Recursively Connected Tenant Controllers

3.1.4 SDN Provider with Recursively Connected Tenant Controllers

SDN provider is able to be placed in a recursive order with tenants. Figure 3.4 demonstrates this configuration. According to the figure, the SDN provider offers services to tenant B and tenant B offers services to tenant A. As a result, in this configuration, tenant A is using some of the offered services by the SDN provider but it is not directly connected to the SDN provider. The assigned services might be physical or virtual resources. For example, tenant B may offer physical resources which are tunneled to use the services from the SDN provider [3].

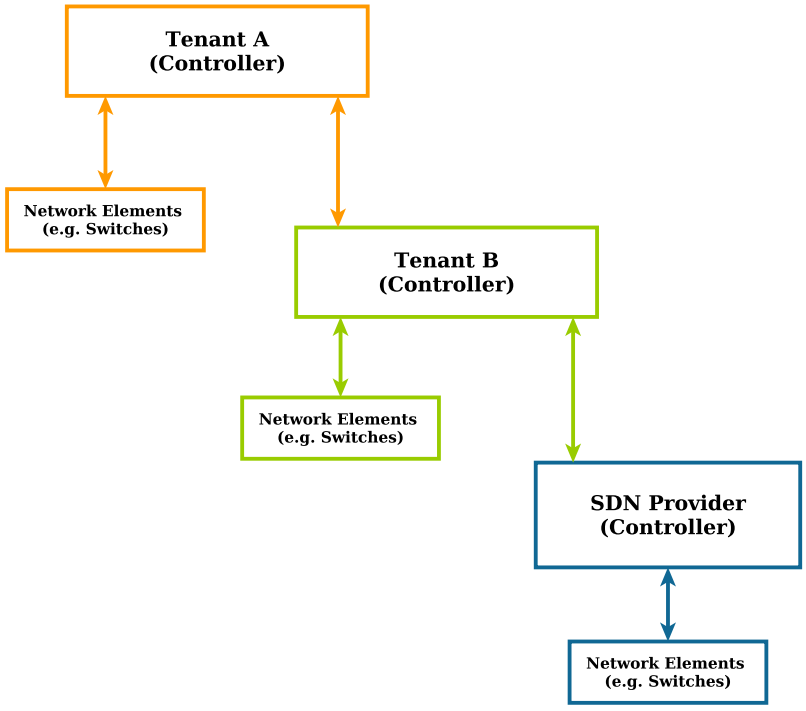

Figure 3.4: SDN Provider with Recursively Connected Tenant Controllers

3.2 Current Isolation Techniques in a Multi-Tenant SDN

Isolation is one of the most important security aspects of multi-tenant environments. Since the resources are shared, it is vital to provide isolation between tenants. Isolation in a multi-tenant SDN can be achieved by implementing the isolation at the controller plane and the data plane. Tenants connect to the controller to send the configuration requests for forwarding at the data plane. Hence, there should be a possibility to differentiate these requests at the controller plane between all tenants. Moreover, at the data plane, we need to distinguish and forward the traffic of different tenants. At the following, we explain isolation techniques that have been used in current multi-tenant SDN solutions. It should be noted that different solutions

may use the combination of these approaches and they are not limited to only one technique. In Section 3.3, we explain some of the existing solutions that are using these techniques to provide isolation in a multi-tenant SDN network.

3.2.1 Slicing

In this approach, depending on the policy and the requirements of different tenants, we can slice a shared network [39, 40]. Consequently, each tenant is assigned a slice of network resources and it is limited to work on its own slice. A slice can be based on the 12-tuples of header fields discussed in Section 2.2.1. For example, slices can be based on source IP addresses. As a result, while the network resources are shared, based on the source IP addresses, the traffic of different tenants is distinguishable and all flows with a certain source IP address are placed in a specific slice. Additionally, it is possible to define slices based on the topology of the underlying network. This means a slice may consist of a set of switches, ports or links on the data plane. Consequently, all the traffic from those switches, ports or links belongs to a slice of a particular tenant. The slices are created by the administrator of the network who defines the slicing policy for all tenants.

3.2.2 Encapsulation

The encapsulation methods are used in a multi-tenant SDN network for traffic isolation at the data plane [38, 44]. By using the encapsulation method in multi-tenant networks, the traffic of each tenant is tagged with a new header (label) at the ingress switch and it is removed at the egress switch. One of the well-known example of encapsulation methods is Virtual Local Area Network (VLAN) which allows to define several virtual networks on a shared physical network. VLAN tagging based on IEEE 802.1q standard, adds a 12-bit tag to the frame. In a multi-tenant SDN, the controller tags the traffic of each tenant with a unique VLAN at the ingress switch and removes the tag at the egress switch. As a result, the usage of VLAN leads to traffic isolation at the data plane among tenants. While this approach is simple, it has the scalability problem and it is limited to 4096 number of VLANs [50].

Another example which is based on the encapsulation technique is Multiprotocol Label Switching (MPLS). The length of MPLS label is 20-bit which allows to define over a million labels [12, 26, 50]. The other way of encapsulation is the use of overlay network technologies like Virtual Extensible LAN (VXLAN). VXLAN allows the providers to define about 16 million virtual networks by adding 24-bit VXLAN Network Identifiers (VNIs) to frames [23].

3.2.3 Packet Rewriting

Packet rewriting at the controller plane

Considering a case where several tenants are connected to a shared controller plane. In this case, we need to isolate control requests (OpenFlow messages) between different tenants. One way to isolate the requests that belong to different tenants is rewriting the control messages. In fact, for providing isolation between tenants,

the controller rewrites the control messages and delivers the requests to the corresponding tenant [39, 49]. Figure 3.5 illustrates a packet rewriting at the controller plane.

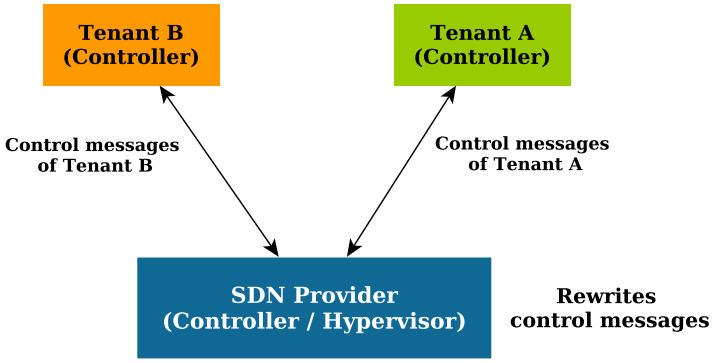

Figure 3.5: Packet rewriting at the controller plane

Packet rewriting at the data plane

In this approach, the isolation is achieved by mapping the information in the packet header and rewriting the headers at the data plane [49]. In fact, for providing isolation, we change header fields to make packets distinguishable. In SDN networks, the controller is responsible to keep mapping between the original and the mapped information and usually, the packet rewriting is enforced at the edge of the network.

For instance, it is possible to be implemented one type of mapping based on the source IP address. Consider the SDN network depicted in Figure 3.6. According to the figure, when one end-host sends traffic to the network with any IP address, the ingress switch, maps the host's IP address to a predefined mapped IP address. When the traffic leaves the network at the egress switch, the mapped IP address will be changed to the original IP address. The IP address mapping in this example leads to the traffic isolation between different sources and makes it possible to choose overlapped IP addresses for end-hosts.

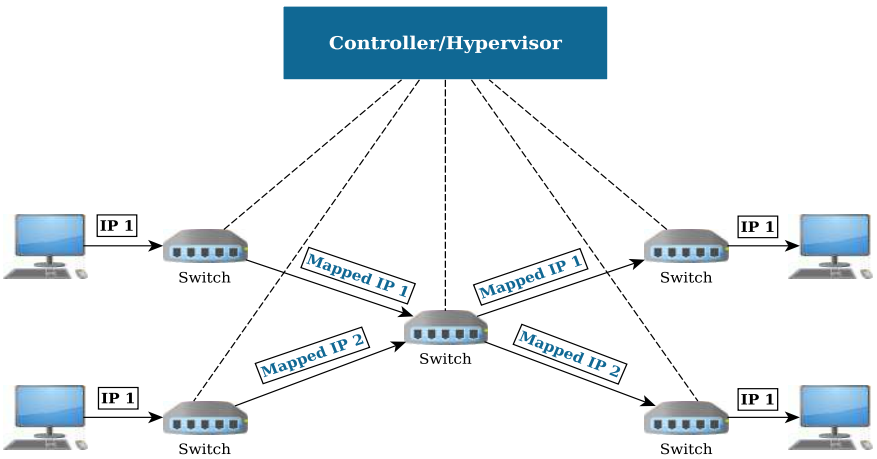

Figure 3.6: Packet rewriting at the data plane

3.3 Available Multi-Tenant SDN Solutions and their Isolation Approaches

In this section, we explain some of the available solutions that are used in a multi-tenant SDN network. These solutions provide multi-tenancy based on the configurations explained in Section 3.1. For providing isolation, these solutions use the isolation approaches discussed in Section 3.2.

3.3.1 FlowN

FlowN [38] is a virtualization solution for multi-tenant SDN environments. It provides the ability to define customized virtual networks for tenants. Tenants run their applications on the controller and the controller maps the traffic of each tenant from the virtual network at the controller plane to the physical network elements in the data plane. Figure 3.7 illustrates the architecture of this approach.

FlowN allows tenants to run their applications on LXC containers introduced in Section 3.1.1 and using these containers, the applications are able to communicate by function call and callback APIs with the controller system. The controller is responsible to assign an abstract view of the resources at the underlying network to the applications. It receives the requests from the tenant applications and translates them to the forwarding rules on the network elements in the data plane. Alternatively, the controller receives the requests from the data plane and forwards them to the tenant applications.

The isolation method at the data plane is based the encapsulation method introduced in Section 3.2.2. In fact, it assigns a VLAN tag for each tenant and these tags are appended to the traffic at the ingress switch and will be removed at the egress switch.

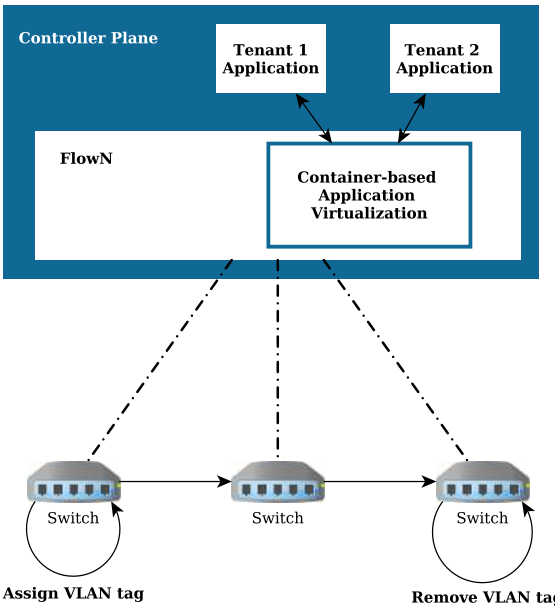

Figure 3.7: FlowN Architecture

3.3.2 Splendid Isolation

Stephen Gutz et al.[44] introduced an isolation approach based on the slice abstraction in SDN networks. The isolation is achieved by making slices based on an abstraction of the underlying network resources and then, the defined slices and their corresponding programs are compiled with a compiler, which guarantees isolation between slices. The slices and programs are defined with NetCore [45], a high level language for OpenFlow-enabled networks.

The concept of slicing in this method is defined based on three factors: *topology*, *mapping* and *predicates*. The topology includes a set of switches and links that are used in a slice. The mapping defines the relation between the defined topology and the network elements in the data plane and the predicates are used to differentiate the traffic of different slices. It should be noted that the slices may be separated from each other (physical isolation) or they may have shared resources.

In this solution, tenants make programs according to their own topology. The compiler receives the slice definition and a program for that slice and compiles them together. The compiler chooses an unused VLAN tag for the slice and the associated program and it installs rules on the switch at the data plane to push and pop VLANs on packets that match the predicate of that slice. When a packet reaches an edge switch at the data plane, the predicates in the packet header are used to find a slice that the packet belongs to. After finding the slice, the traffic will be tagged with a VLAN tag. Later, this label is removed when the packet leaves the slice.

To verify the isolation between slices, a verification tool is developed. Instead of relying on compiler functionalities, the verification tool analyzes the output of the compiler. The isolation is verified in two aspects: Traffic isolation and Physical isolation. The traffic isolation ensures a packet that is forwarded in a slice is allowed to only traverse switches, ports and links for the corresponding slice. Moreover, the physical isolation is used in a case that one slice should not have any switch or links shared with other slices.

3.3.3 FlowVisor

FlowVisor [39, 40, 41, 42] creates a virtualization layer between the controller plane and the data plane in SDN networks. FlowVisor is placed between the tenants and network resources at the data plane and it allows to attach several controllers of different tenants to a shared network. FlowVisor allocates the resources of a shared network based on the slicing approach discussed in Section 3.2.1. It defines the concept of slice to divide the network resources between different tenants. A slice consists of a set of flows on the data plane which is named FlowSpace. To isolate the control traffic of different tenants, FlowVisor intercepts all OpenFlow messages and based on the slicing policy, it rewrites and forwards OpenFlow messages to the corresponding tenant controller. As a result, FlowVisor provides isolation and allows tenants to control their dedicated resources. Moreover, control traffic translation in FlowVisor leads to complete transparency for tenants and they do not notice the existence of this layer. Figure 3.8 demonstrates an example of the slicing approach based on the topology of the underlying network using FlowVisor.

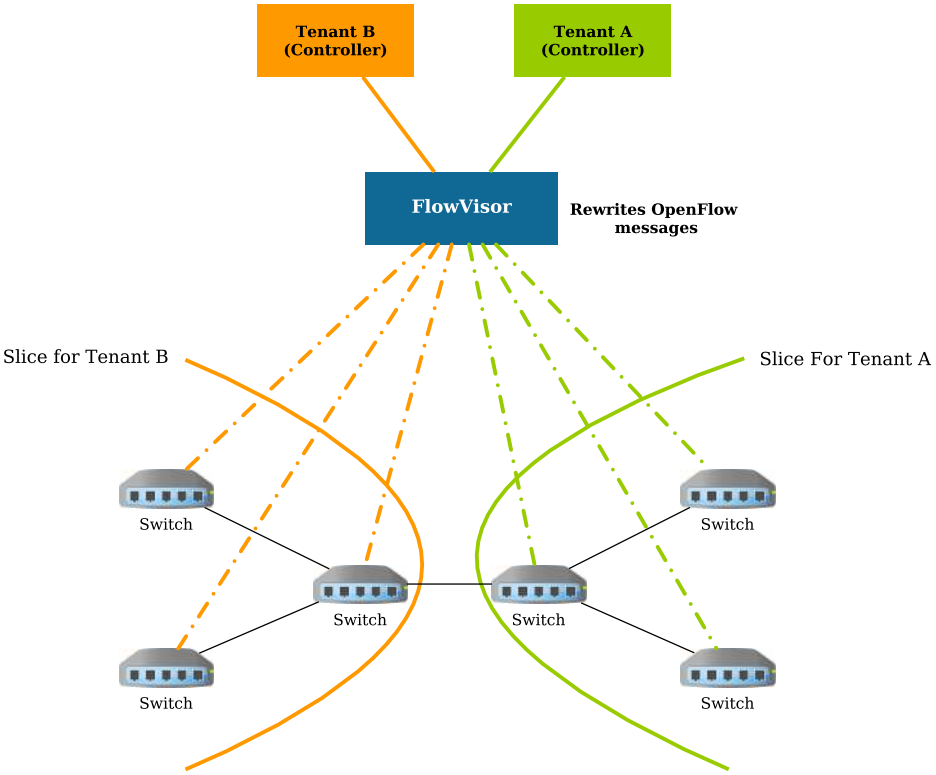

Figure 3.8: Slicing approach (topology slicing) in FlowVisor

The isolation in FlowVisor can be classified into the following aspects:

- **Bandwidth Isolation:** FlowVisor provides bandwidth isolation based on the concept of the minimum bandwidth guarantee between slices. The minimum bandwidth for each slice is defined in slice's policy. As a result, each slice can only consume a fraction of the bandwidth of the link based on the value specified in the definition of the policy for that slice. Consider an example where two slices are competing for the bandwidth on a shared link. FlowVisor may allow one slice to use only 30 percent of the bandwidth while the other slice may consume the 70 percent of the bandwidth of the link.

- **Topology Isolation:** To provide transparency, FlowVisor acts as a proxy and receives the connections from the switches at the data plane and forwards them to the related slices. Moreover, it receives OpenFlow messages that include information about switch ports and rewrites and forwards them to the corresponding tenant controller.

- **Switch CPU Isolation:** Switches with commodity hardware on the data plane are normally built based on low-power processors. Therefore, they can not process large amount of requests and they will be easily overloaded. To keep these devices responsive, there should be an isolation level between slices.

- **FlowSpace Isolation:** Slices should only affect the flows of their FlowSpaces. FlowVisor ensures isolation between slices by rewriting the OpenFlow messages to only affect the flows of the corresponding slice.

- **Flow Entries Isolation:** FlowVisor calculates the number of flow entries corresponding to each slice. If it surpasses the threshold, the error message "table full" is sent to the corresponding tenant controller.
- **OpenFlow Control Isolation:** The OpenFlow protocol uses a 32-bit transaction identifier. FlowVisor rewrites this value for different controllers to prevent them from receiving an identical transaction identifier. Additionally, in case of packet-in events, the switch may keep the packet in an internal queue and send a request to the controller with a buffer id. A buffer id in OpenFlow messages should be changed for each slice. Furthermore, other OpenFlow messages like status messages need to be duplicated for all slices.

3.3.4 AutoSlice

AutoSlice [43, 48] is placed between the controller plane and the data plane and allows tenants to have an arbitrary virtual view of the underlying network. In this solution, tenants are able to connect their own controllers to AutoSlice and control their own virtual network. Figure 3.9 depicts the architecture of AutoSlice. The architecture of AutoSlice is based on two main components: *Management Module (MM)* and *Controller Proxy (CPX)*. MM is used for receiving tenant network requests (virtual networks) and CPXs are used to manage the load from different tenant controllers. Furthermore, the underlying network is sliced into multiple SDN domains and each CPX is responsible to manage one SDN domain. When MM receives a network request, it maps it to the resources on the underlying network and configures CPXs to process the tenant's traffic for the corresponding SDN domain.

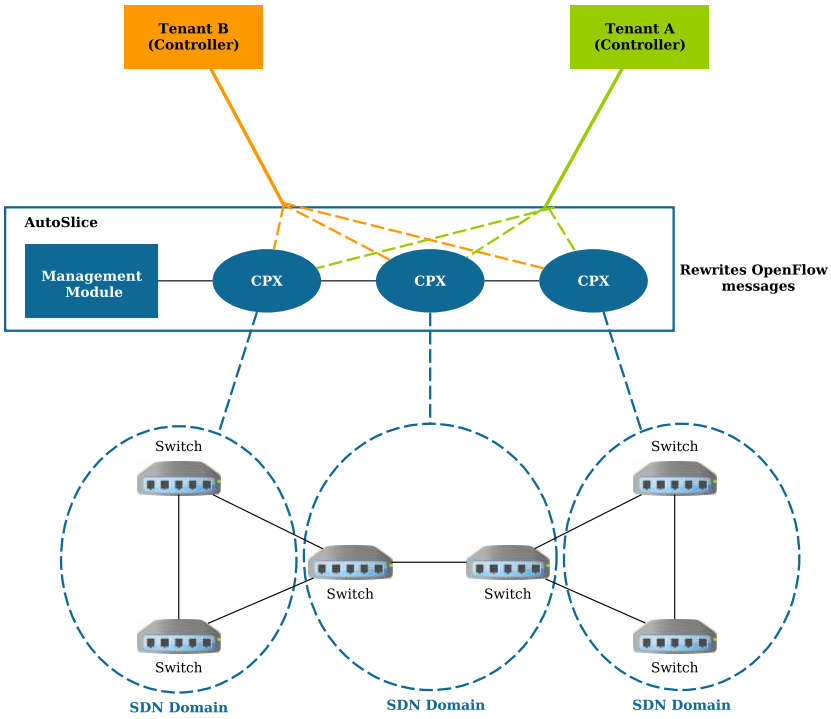

Figure 3.9: Architecture of AutoSlice

All control messages from the tenant's controller are redirected to the CPX and it rewrites the control messages. Similar to FlowVisor, each CPX in AutoSlice checks the forwarding rules to be non-overlapped before installing on the switches. As a result, it guarantees the isolation between the flow entries of different tenants. To provide traffic isolation between different tenants at the data plane, AutoSlice embeds identifiers to packets. These identifiers are embedded in the packets with the encapsulation methods such as VLAN or MPLS. The encapsulation method is explained in Section 3.2.2.

3.3.5 OpenVirtex

OpenVirtex [49, 19] is the next generation of virtualization solutions. OpenVirtex is placed between the controller and the data plane and allows several tenants to connect to a shared network. Figure 3.10 shows the isolation approach at the controller plane and the data plane in OpenVirtex. Similar to FlowVisor, OpenVirtex rewrites the control messages (OpenFlow messages) for providing isolation between different tenant controllers. At the data plane, OpenVirtex rewrites the packet headers at the edge of the network and embeds the information about the virtual networks of each tenant in the packet headers. Since OpenVirtex rewrites the packet headers at the data plane, it provides traffic isolation and allows tenants to have overlapped IP addresses.

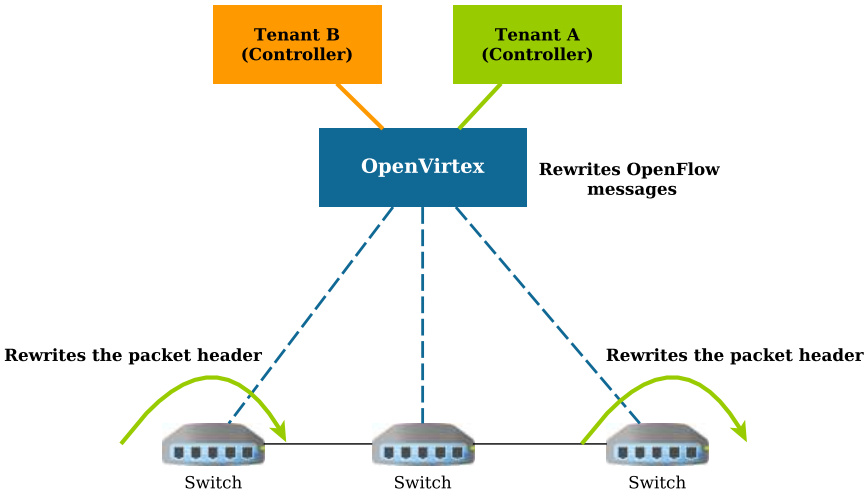

Figure 3.10: Isolation at the controller and data planes in OpenVirtex

3.4 Chapter Summary

This chapter discusses the multi-tenancy in SDN and particularly, the isolation in a multi-tenant SDN network. In a multi-tenant SDN, tenants are able to connect to the service provider network with their SDN controllers or applications and control their own network resources. Since, the network resources are shared, isolation becomes a necessary requirement in a multi-tenant SDN. Current isolation techniques

for a multi-tenant SDN are slicing, encapsulation and packet rewriting. With slicing technique, the service provider slices network resources and assigns a slice to each tenant. The encapsulation method provides isolation by tagging the data packets and each tag (or label) in this method is assigned to one tenant. In the packet rewriting approach, the packet headers are changed by the switches at the data plane to provide isolation between tenants. FlowN, FlowVisor and OpenVirtex are examples of current solutions which provide isolation by the aforementioned techniques.

Chapter 4

Proposed System Design

In this chapter, we propose an architecture that provides isolation in a multi-tenant SDN network. In the proposed system, while a tenant is able to stay completely isolated from other tenants, it is possible to connect to the offered services by other tenants or allow others to use its own services. Moreover, since we are working in a multi-tenant network, the scalability is another challenge that we have considered in our solution. It should be applicable to deploy our isolation architecture in large networks with millions of tenants and hosts. At the following, at first, we present our design goals and then we start to explain about the principles that we have used in our architecture. These principles are the fundamental concepts that are used to explain the relation between tenants and provider network. Subsequently, we demonstrate a high level explanation of our solution. This includes a description on the functionality of the main components in our architecture. At the end of this chapter, we focus on the communication patterns and how tenants may interact with each other.

4.1 Design Goals

Our solution should fulfill the following requirements.

- **Effective isolation in a multi-tenant network**
  Our system should provide complete isolation between tenants in a shared SDN network. We classify the isolation requirements into the following aspects:

  Traffic isolation: The traffic of each tenant should completely be isolated from other tenants and there should not be any information leakage between tenants. Each tenant should be able to send any type of traffic to the network without affecting other tenants.

  Address space isolation: In our solution, it should be possible to assign overlapped IP Addresses and MAC addresses to end-hosts of different tenants. In fact, while a tenant is working in its own subnet, it should not be possible to affect other tenants in the same address space in the network.

  Control isolation: Tenants should be able to control their own traffic. There should be a possibility for tenants to connect to the network and define their

own policies for their traffic. Moreover, it should not be possible for a tenant to control others or affect the configurations made by other tenants.

Performance isolation: The performance isolation is a broad concept which can be defined in different perspectives. In our solution, we define performance isolation between tenants by setting a threshold on network resources for each tenant. As a result, tenants should not be able to violate the maximum allocated capacity of the network resources and all violations from tenants should be detected.

- **Supporting intra-tenant, inter-tenant and external communications**
  It should be possible for tenants to make connection between their own hosts. Additionally, tenants may need to share specific services with each other within a shared network. While in existing solutions it is not simply feasible for a tenant to share services with other tenants, we aim to make a possibility for tenants to share and offer services to each other. Besides, tenants should be able to connect to the external services outside of a shared network (e.g. Internet).

- **Satisfactory level of scalability**
  The solution should be scalable to be implemented in large networks with millions of hosts and tenants. Moreover, the scalability of the solution for deployment on today's network devices should be taken into the consideration.

4.2 Design Pattern

Our solution separates the isolation layer from the routing layer. While the isolation layer guarantees that only verified flows are able to be transmitted, the routing layer decides about how to transmit a flow. The isolation layer matches the received flow requests with a set of rules and defines if the flow is allowed or not. The routing layer is concerned with the forwarding tasks at the data plane. It routes the verified flows in a specific path from the source to the destination.

Moreover, while the software switches can save millions of forwarding rules, today's hardware switches cannot hold large number of forwarding rules in their flow tables [51, 52]. As a consequence, it is not possible to install fine-grained rules for each flow request in all switches available in the network. To tackle this issue, in our design we separate the edge from the core network. In fact, we isolate flow requests at the edges of the network and we route them at the core network based on the routing labels. This approach is in accordance with the recent research efforts. Several of recent papers [51, 53] proposed to separate the edge and core networks by making edges more intelligent and providing label switching at the core. Our solution is similarly based on the same concept. While we enforce isolation at the edge of the network, we use routing labels to aggregate several flows together and route them at the core network. At the edge network, since we should implement fine-grained rules, we use software solutions and at the core network we use hardware switches to increase speed and efficiency for forwarding the flows. For enforcing isolation at the edge network, we use the concept of policy and isolation rules. Figure 4.1 illustrates the design of our solution.

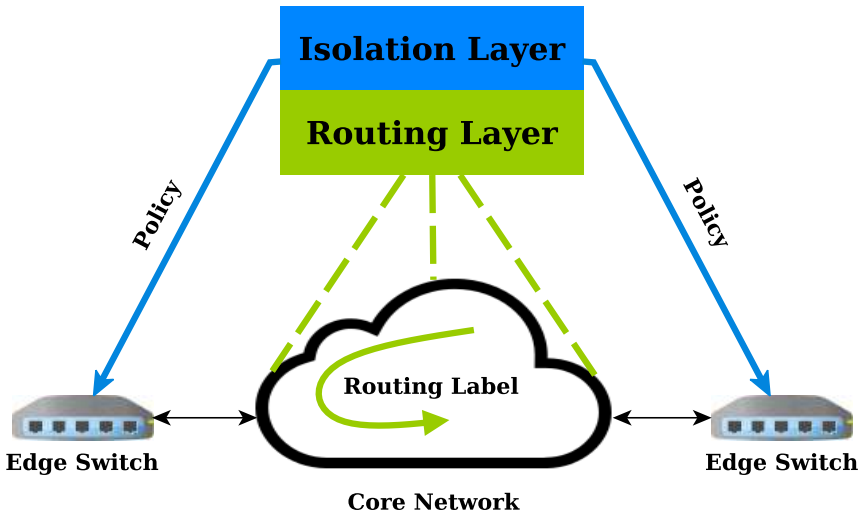

Figure 4.1: Design pattern

4.3 Principles

In this section, we explain the concepts that are used in our architecture. For the rest of this thesis, we use these concepts to explain the functionality of our architecture.

4.3.1 SDN Provider

The SDN provider owns all network resources and it is responsible for all management tasks in the infrastructure network. The SDN provider is responsible to implement the infrastructure network, configure the SDN controller and network elements (switches, gateway) in the data plane. Moreover, the SDN provider should register tenants to the network by creating an account and assigning network resources to each tenant.

4.3.2 Tenant

Referring to Section 3.1, we have defined a tenant as a customer or organizational entity which rents the resources of a SDN provider network. In our architecture, a tenant interacts with the SDN provider by connecting its application to the controller of the SDN provider. Tenants are able to create their own configurations, for instance, creating subnets for their hosts, creating policy group for their traffic and advertise their services to the other tenants.

4.3.3 Tenant Network Domain

Tenant Network Domain, which we will call it simply a *domain*, provides a way of managing network resources and creating boundaries between tenants [54]. In fact, a tenant is identified by its domain and all configurations related to a tenant are placed in its domain in the SDN provider network. At the controller plane, a domain

is uniquely identified by a domain label. These labels are generated randomly and they are unique between all domains in the network.

The SDN provider is responsible to create domains and assign each domain a set of input ports and a range of IP addresses. After the initialization process is done by the administrator, tenants are able to create, change or remove their own configurations in their domain.

4.3.4 Policy

Tenants create the policy for their traffic and services and this policy is mapped to the corresponding domain of a specific tenant. In our architecture, the definition of policy is based on *policy groups*. Each policy group has a set of accessibility levels. It can be defined as *Intra-Tenant*, *Inter-Tenant* and *External*. The intra-tenant means that the scope of this group is limited to one tenant. The inter-tenant group indicates that the scope of this group is between tenants in a shared SDN network. The external groups are used for communications with the resources outside of a shared network (e.g. Internet). A policy group may be defined to only have one accessibility level (for example Intra-tenant) or it can be defined to have more accessibility levels (for example Intra-tenant and Inter-tenant).

Additionally, every policy group consists of two main parts: *isolation policy* and *routing policy*.

Isolation Policy

The isolation policy includes a set of *rules*. These rules are defined similar to firewall rules. The isolation policy is used for matching with the received flows. As a result, only a flow that matches a rule will be forwarded. Each rule includes source and destination IP addresses, source and destination transport port numbers and the protocol for the communication.

Routing Policy

The routing policy includes information that needs to be used for routing a flow at the underlying network. It includes information about *routing decision* and *type of communication*. By default, the routing decision is to *drop* all flows. However, the routing decision might be changed if the flow is allowed for forwarding based on the isolation policy. If the routing decision is changed to *allow*, then the flow is forwarded based on the type of communication.

4.4 Architectural Components

In this section, we explain the main architectural components of our system. The northbound interface is used for making the configurations and interacting with the SDN controller. The Service Manager is responsible for processing the received configuration from the northbound interface. To respond to the received requests from the data plane five modules named *Isolation Manager*, *Routing Manager*, *ARP*

Handler, *DHCP Server*, and *Monitoring* are used. Each of these modules is responsible for handling specific tasks on the underlying network. The OpenFlow and sFlow protocols have been used as southbound interfaces for providing interaction between the SDN controller and the underlying network. Figure 4.2 shows the main components of our architecture.

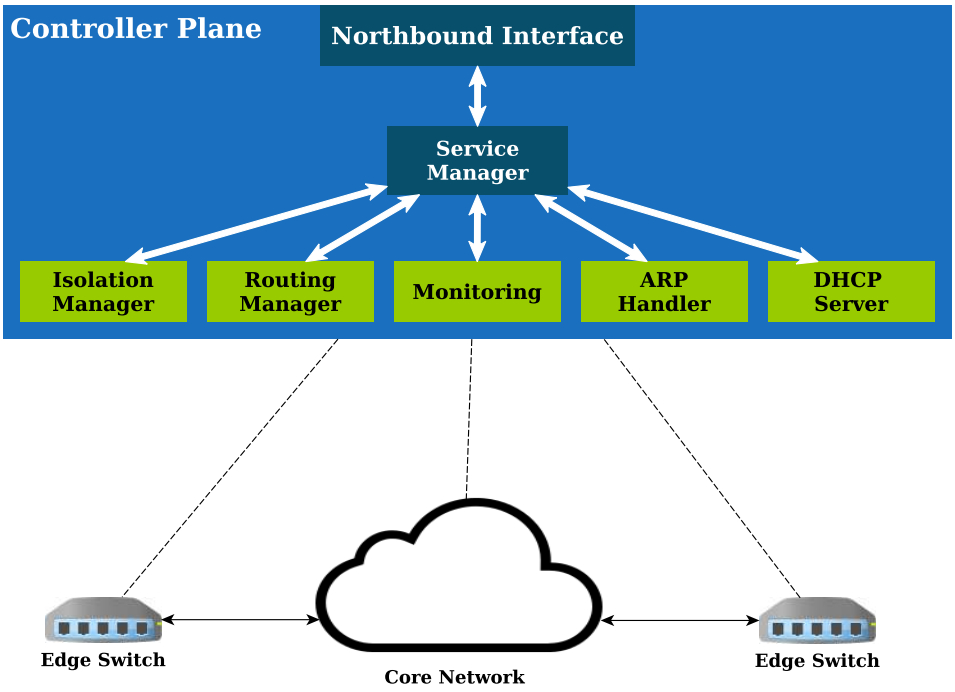

Figure 4.2: System Architecture

4.4.1 Northbound Interface

The northbound interface is used for interaction with the controller plane. The administrator of the network uses the northbound interface for configuring the network resources and adding or removing tenants. Furthermore, tenants use the northbound interface to create subnet for their end-hosts, policy groups for their traffic and advertise services to each other. To use the northbound interface by tenants, the administrator of the network should register tenants beforehand. After the registration of tenants is done by the administrator, tenants are able to set their policy and configurations. All tenants are limited to operate on their own domains and they cannot control any of other resources.

4.4.2 Service Manager

The service manager is placed between the northbound interface and other modules at the controller plane. It handles the received configuration requests from the northbound interface. Depending on the received information, it might inform other modules. One of the main roles of the service manager is to create and manage domains and assign configurations to each domain. It receives the request to create

a domain and adds all configuration information related to a tenant such as policy groups, internal subnet range, inter-tenant and external addresses in the corresponding domain. Other modules like Isolation Manager or DHCP server, use this service to find information about the domains and configurations of the network.

4.4.3 DHCP Server

The Dynamic Host Configuration Protocol (DHCP) server is used for assigning an IP address to end-hosts in the network. The DHCP server stores information about all detected end-hosts, their location in the topology and their domains. Based on the domain and the IP address, it is possible to find any end-host detected by the DHCP server. Our DHCP server operates in two modes: the dynamic mode and the static mode. In the dynamic mode, the IP addresses are assigned to hosts dynamically and this IP addresses are valid for a specific period of time. The other mode is the static mode in which IP addresses are statically reserved by a tenant. In this configuration, the IP address does not change and there is not any timeout for the allocated IP address.

4.4.4 ARP Handler

The Address Resolution Protocol (ARP) is used for discovering the MAC address corresponding to an IP address of an end-host. The default ARP handler mechanism in SDN is in a way that each end-host sends ARP request to the network. Then, the edge switch receives this ARP request and forwards it to the SDN controller. The controller checks the database to find the destination end-host. If it does not find the destination end-host, it broadcasts the packet to the network to find the destination end-host and after finding the destination end-host, the controller sends the ARP reply to the requested end-host. However, the default ARP handler mechanism is not usable in a multi-tenant network. The broadcast requests allow tenants to receive reachability information about other end-hosts in a shared network and it violates the isolation concept of our solution. Therefore, we have changed the default ARP handling mechanism in SDN for our solution. The ARP handler in our solution does not broadcast any traffic to the network, instead, it sends the MAC address of the SDN controller for every ARP request in the network. In this way, all end-hosts have the MAC address of the controller on their ARP table. The controller is responsible for capturing the packets and forwarding them in the network with the isolation manager and the routing manager.

4.4.5 Isolation Manager

The isolation manager is responsible for enforcing isolation in a multi-tenant network. The isolation manager checks the isolation policy for a received flow request and in a case of success, it sends the packet to the routing manager to route the packet in the network. As we described in Section 2.2, when a switch receives a flow, it checks the flow table to find any corresponding entry for the received flow. If it cannot match the received flow with the existing flow entries, it forwards the first packet of the flow to the controller (packet-in event) to decide about the new

flow request. In our system, the isolation manager in the SDN controller receives the packet-in requests and queries the service manager for finding the source and destination domains based on the source and destination IP addresses and the input switch port of the originating end-host at the edge of the network. Depending on the relation of domains, the isolation manager checks the isolation policy in three cases:

- **The source and destination domains are found and they are equal:** In this case, the packet belongs to the same domain. This type of connection is intra-tenant and it is between the end-hosts of the same tenant network. Accordingly, the isolation manager finds the intra-tenant policy groups for the source domain and checks the isolation policy in each policy group.
- **The source and destination domains are found but they are not equal:** In this case, the packet belongs to two different domains. This type of connection is inter-tenant and the isolation policy in the inter-tenant policy groups for the source and destination domains should be checked.
- **The source domain or the destination domain does not exist:** In this case, an end-host is making the external connection to the outside of the network or a response from an end-host outside of the network is coming back to the network. Therefore, the isolation policy in external policy groups for the source or destination domains should be checked.

After checking the isolation policy, the decision (allow or deny a flow) is added to the corresponding routing policy. In addition, in case the flow is allowed for forwarding, the type of communication (intra-tenant, inter-tenant or external) is set in the routing policy and the packet with the routing policy is delivered to the routing manager for forwarding or dropping the flow at the data plane.

4.4.6 Routing Manager

The routing manager is in charge of routing tasks. However, since the design of our system is different at the edge and core network, the routing manager has different functionality for forwarding at the edge and core network. At the core network, we install forwarding rules proactively and at the edge network, we install forwarding rules reactively. Figure 4.3 depicts the functionality of the routing manager at the core and edge network. At the following, we explain more details about the forwarding process based on our routing manager.

Forwarding at the Core Network

Forwarding at the core network is based on hardware switches. However, today's hardware switches cannot support the large number of flow entries in their flow tables. It fact, for sake of scalability, we need to decrease the number of forwarding rules at the core network. Authors in [51, 52], proposed a new way to increase the scalability at the core network. Similarly, we use this technique with some changes for our architecture. In our solution, depending on a path between two edge switches,

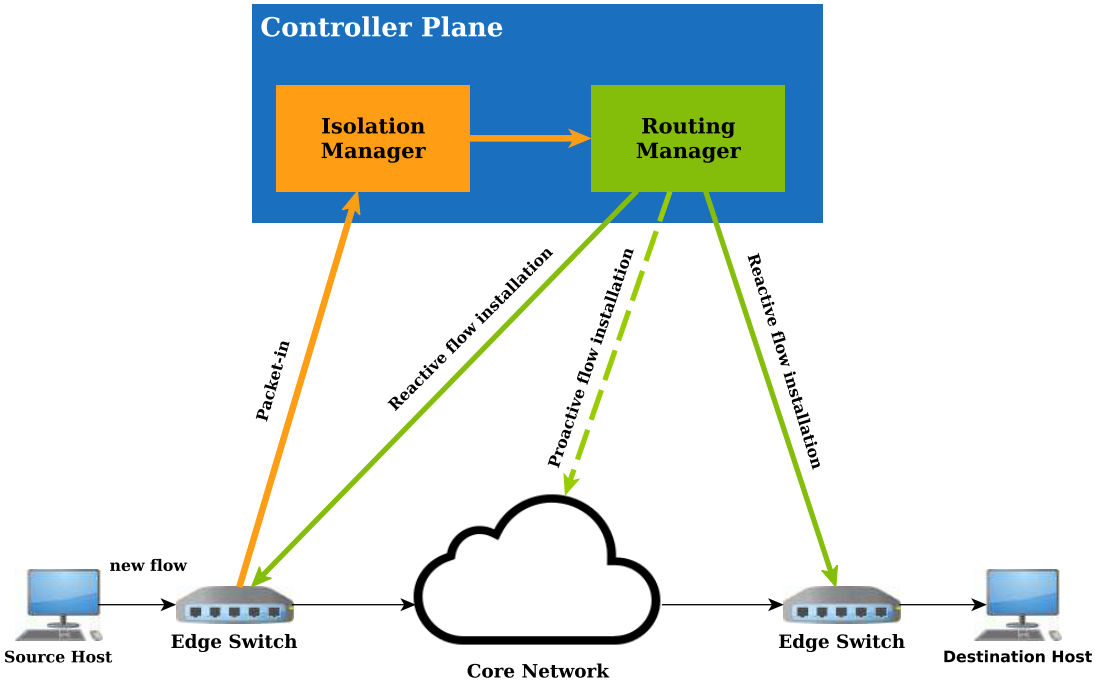

Figure 4.3: Routing in the System Architecture

we consider a routing label. As a result, we aggregate and route several flows in a same path between two edge switches at the core network. This approach leads to high reduction in the number of forwarding rules at the core network. Moreover, the forwarding rules at the core network are installed proactively between all edge switches. In fact, we install forwarding rules based on the routing labels between edge switches after detecting a new edge switch in the network.

Forwarding at the Edge Network

For forwarding at the edge network, the routing manager must receive the requests from the isolation manager. When the routing manager receives a request from the isolation manager, depending on the routing policy, this unit forwards or drops the flow at the edge of the network. The routing manager uses the information in the routing policy for forwarding a flow. If the routing policy is to deny the flow, then the routing manager installs a rule on the edge switch connected to the source end-host to drop the flow at the edge of the network. If the routing policy is to allow the flow, depending on the type of communication (intra-tenant, inter-tenant or external), the routing manager installs forwarding rules at the ingress edge switch and the egress edge switch for forwarding the flow through the network. As we explain in Section 2.2, in OpenFlow, each forwarding rule includes a list of actions. It means that if the flow is matched with a forwarding rule in the switch, the corresponding actions of the forwarding rule will be implemented on the flow. In our architecture, the routing manager installs forwarding rules with a list of actions at the edge switches to rewrite the header of the packets. In Section 4.5, we explain how the packet rewriting is implemented for different communication patterns.

4.4.7 Monitoring

The monitoring is responsible for detecting unusual traffic from tenants at the data plane and informing the detected issues to the administrator. It receives the statistics from the switches at the data plane and then the statistics are analyzed at the controller plane and in case of any violation, the monitoring informs the administrator. Based on these requirements, we demonstrate the functionality of the monitoring in three steps:

- **Collection of statistics:** In this case the monitoring unit collects the statistics from all network elements in the data plane. The statistics data are sampled from received packets and the result is transferred in a periodic time interval to the collector unit at the controller plane.
- **Analysis of statistics:** After receiving the statistics, in each time interval, the monitoring unit analyzes the received statistics information to check if any type of malicious activities happened in the network. It checks the number of received frames per flow for each switch at the data plane. If the number of received frames in a specific period of time goes beyond the threshold, then the monitoring reacts to the new event.
- **Reaction:** After detecting any unusual activity, the result is informed to the administrator. The administrator may have defined specific actions in case of anomaly detection. For example, it may define the default action in case of violation to drop the malicious traffic or log the detected event.

4.5 Communication Patterns in the System Architecture

In this section, we discuss all types of communications in our system architecture. In our solution, tenants are able to make three types of communications: *intra-tenant*, *inter-tenant* and *external*. In all these cases, we should provide isolation. When a tenant limits a traffic type to a special communication pattern, we should guarantee that type of communication is only possible.

4.5.1 Intra-Tenant Communications

The scope of intra-tenant communications is limited to one domain and it leads to strict isolation for tenant's traffic. In fact, the source and destination end-hosts in this type of connection are placed in the same domain. For making this type of communication, the source and destination IP addresses for end-hosts must be in a range of 10.0.0.0/9. We have reserved this subnet for each tenant and tenants are allowed to assign overlapped IP addresses in any subnet in this range.

Intra-Tenant Communication between two End-Hosts

At the following, we give an example for explaining the functionality of our system architecture for the intra-tenant communications. Figure 4.4 demonstrates the op-

erational flow for this type of connection. For this example, we assume that the end-hosts received the IP address from our DHCP server and all ARP requests are processed by the ARP handler module.

1. For this type of communication, tenants should make a policy group with the accessibility level set to "intra-tenant". The isolation policy (rules) in a policy group should allow the communication between two end-hosts.

2. End-host A sends a new flow to end-host B.

3. When a new flow is received by the ingress edge switch, the edge switch checks its flow table. If there is not any matching entry for the received flow in the flow table, then the first packet of the flow is forwarded to the controller to decide about it.

4. The isolation manager at the controller plane receives the packet. The isolation manager asks the service manager for finding the source and destination domains based on the received packet. After finding the domains, the isolation manager checks the policy groups. Since in this type of connection the source and destination domains are equal, the isolation manager checks the policy groups with the accessibility level set to "intra-tenant". Subsequently, based on the results from checking the policy groups, the isolation manager changes the routing policy and then, the packet and the corresponding routing policy are delivered to the routing manager.

5. The routing manager checks the routing policy and based on the routing policy, it installs forwarding rules on edge switches. If the routing policy is to drop the flow, then it only installs a forwarding rule at the ingress edge switch to drop the flow. Otherwise, the routing manager installs forwarding rules on the ingress and egress edge switches to forward the flow through the network.

6. If the flow is allowed for forwarding, at the ingress edge switch, the source and destination MAC addresses for the flow will be changed. The source MAC address is changed to the domain label and the destination MAC address is changed to the routing label.

7. At the core network, the flow is routed based on the routing label embedded in the destination MAC address.

8. At the egress edge switch, the destination MAC address will be changed to the real MAC address of the destination end-host (otherwise, the destination end-host does not accept flow). Thereafter, the flow is forwarded to the destination.

It should be noted that in this example, the source end-host and the destination end-host are connected to different edge switches. However, if the source end-host and the destination end-host are connected to the same switch, since we do not forward the flows through the core network, we only install one forwarding rule on the ingress edge switch to change the destination MAC address and forward it to the destination end-host.

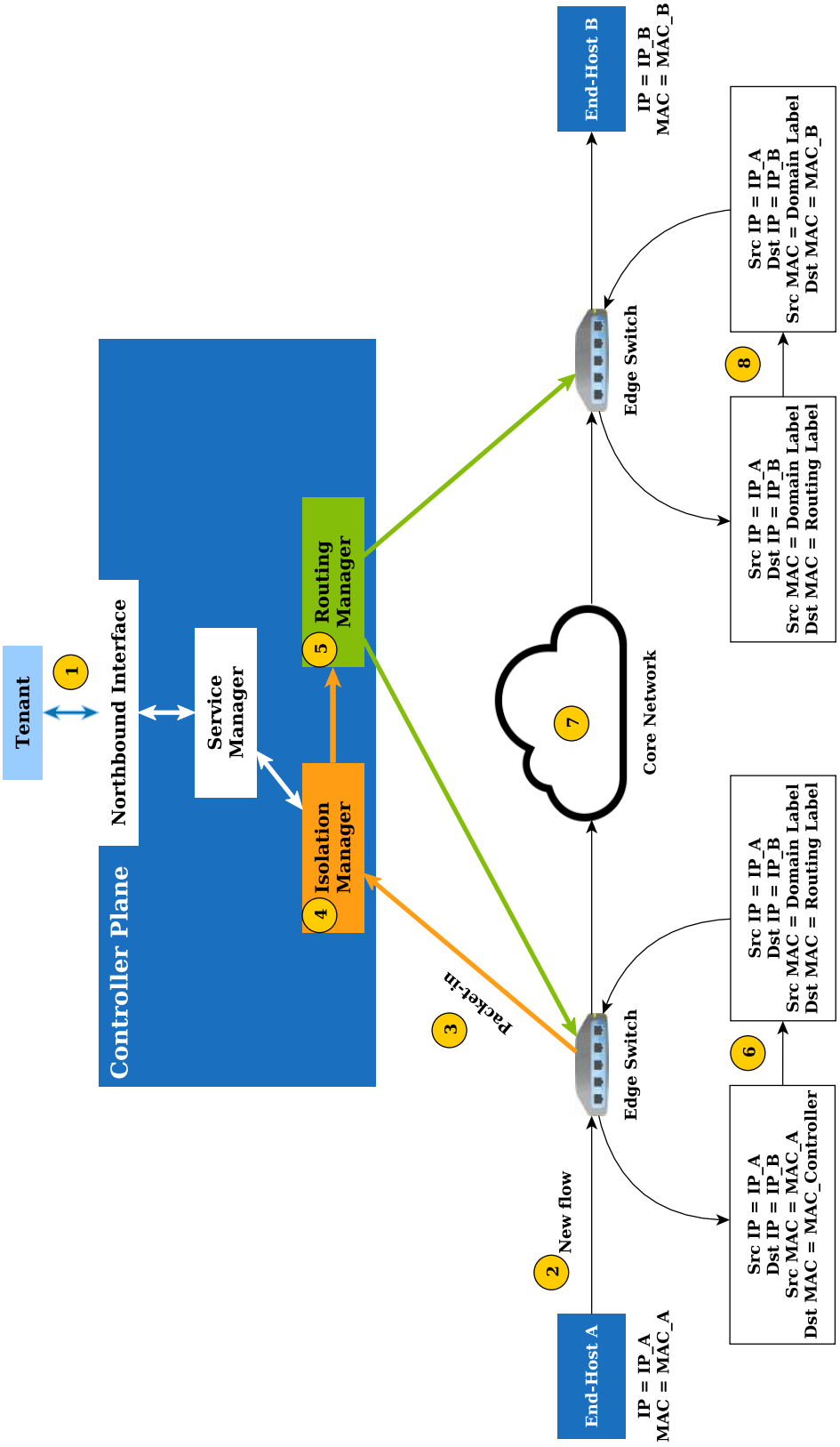

Figure 4.4: The operational flow for the intra-tenant communication

4.5.2 Inter-Tenant Communications

While the existing solutions discussed in Section 3.3 do not provide interoperability between tenants, our solution makes it possible for tenants to connect to each other and use special services offered by other tenants. In fact, while our solution provides isolation between tenants, it makes it possible for tenants to interact with each other and use special services offered locally within the service provider network. Each tenant can advertise services to the network and other tenants are able to query for finding offered services by other tenants. Since inter-tenant services are shared between tenants, they need to be assigned unique private IP addresses to be accessible by other tenants. These unique IP addresses can be assigned by the administrator to each tenant. The IP addresses assigned for inter-tenant communication must be in range of 10.128.0.0/9 and 172.16.0.0/12.

Service Advertisement

For this type of communication, tenants must advertise their services to the network. The service advertisement registers the service to the network and allows others to find the offered services. For advertising a service, tenants can use the northbound interface to set the reachability information for their services to the network. The reachability information includes the description of a service, the source IP address and port number and the protocol for a specific service. Since in the inter-tenant connection the IP addresses should be internally unique, the controller maps the source IP address and port number to an available inter-tenant IP address and a port number. Other tenants can send a query to the network for discovering available services. Accordingly, other tenants will receive the mapped source address and source port accompanied by other reachability information of the service. Figure 4.5 depicts an example of the service advertisement. According to Figure 4.5, tenant A advertises a service by the northbound interface and subsequently, this information is stored in the corresponding domain for tenant A. When tenant B requests for offered services, the service manager sends the reachability information for this service by the northbound interface to the tenant B. As it is demonstrated in Figure 4.5, tenant B receives the mapped IP address and port number for reaching the service offered by tenant A.

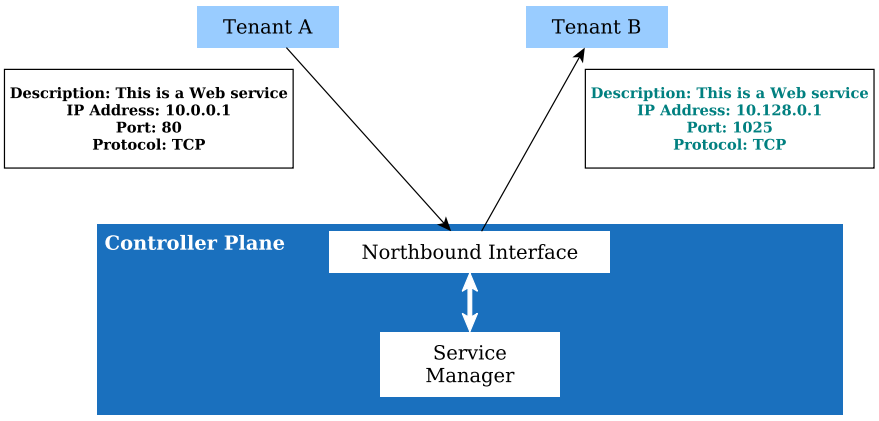

Figure 4.5: An example of the service advertisement for inter-tenant communications

Inter-Tenant Communication between two End-Hosts

At the following, we give an example to clarify the operational steps of our system architecture for the inter-tenant communications. Figure 4.6 depicts the operational steps for inter-tenant communications. We assume that two tenants in this network want to communicate with each other. End-host A belongs to tenant A and end-host B belongs to tenant B. End-host B is offering an inter-tenant service. Moreover, we assume that both of the end-hosts received the IP address from the DHCP server and the ARP handler responds to ARP requests.

1. For inter-tenant communications in this example, tenant B creates a service and advertises it to the network and tenant A queries the network to find offered services. Moreover, tenants should create inter-tenant policy groups to allow this type of communication.

2. End-host A initiates a connection to reach end-host B. End-host A is using the mapped IP address and port number for reaching end-host B. The mapped IP address and port number are provided through the service advertisement phase.

3. The flow request from end-host A is received by the edge switch. Subsequently, the edge switch checks its flow table to find any matching entry for the new flow. If it cannot find any matching entry, the first packet of the flow is forwarded to the controller (packet-in message).

4. When the packet-in is received by the controller, the isolation manager receives the packet and asks the service manager for finding the source and destination domains based on the received packet. If the source and destination domains exist in the network and they are different, the "inter-tenant" policy groups for the source and destination domains are checked. Based on the results from checking policy groups, the isolation manager changes the routing policy for forwarding the flow through the network. Then, the isolation manager sends the packet and the corresponding routing policy to the routing manager.

5. Based on the routing policy, the routing manager drops the flow or forward the flow at the data plane. If the routing policy is to drop the flow, then the routing manager installs a forwarding rule at the ingress edge switch to drop the flow. Otherwise, the routing manager installs forwarding rules on the ingress and egress edge switches to forward the flow through the network.

6. At the ingress edge switch, the source MAC address is changed to the domain label and the destination MAC address is changed to the routing label for each flow. Additionally, since it is inter-tenant communication, the routing manager changes the source IP address and source transport port to a free mapped IP address and port number for each flow. In addition, the destination IP address and port number are changed to the real address and port number for end-host B.

7. At the core network, the flow is routed according to the routing label embedded in the destination MAC address of all packets.

8. At the egress edge switch, the destination MAC address is changed to the real MAC address of end-host B and then the flow is forwarded to the destination end-host.

It should be mentioned that in this example, the source and destination end-hosts are connected to different edge switches. However, if the source and destination end-hosts are placed on the same edge switch, since we do not route the flow through the core network, we only install one rule to change the source and destination IP addresses and port numbers, and then we change the destination MAC address according to the real MAC address of the end-host B and forward the flow to the destination.

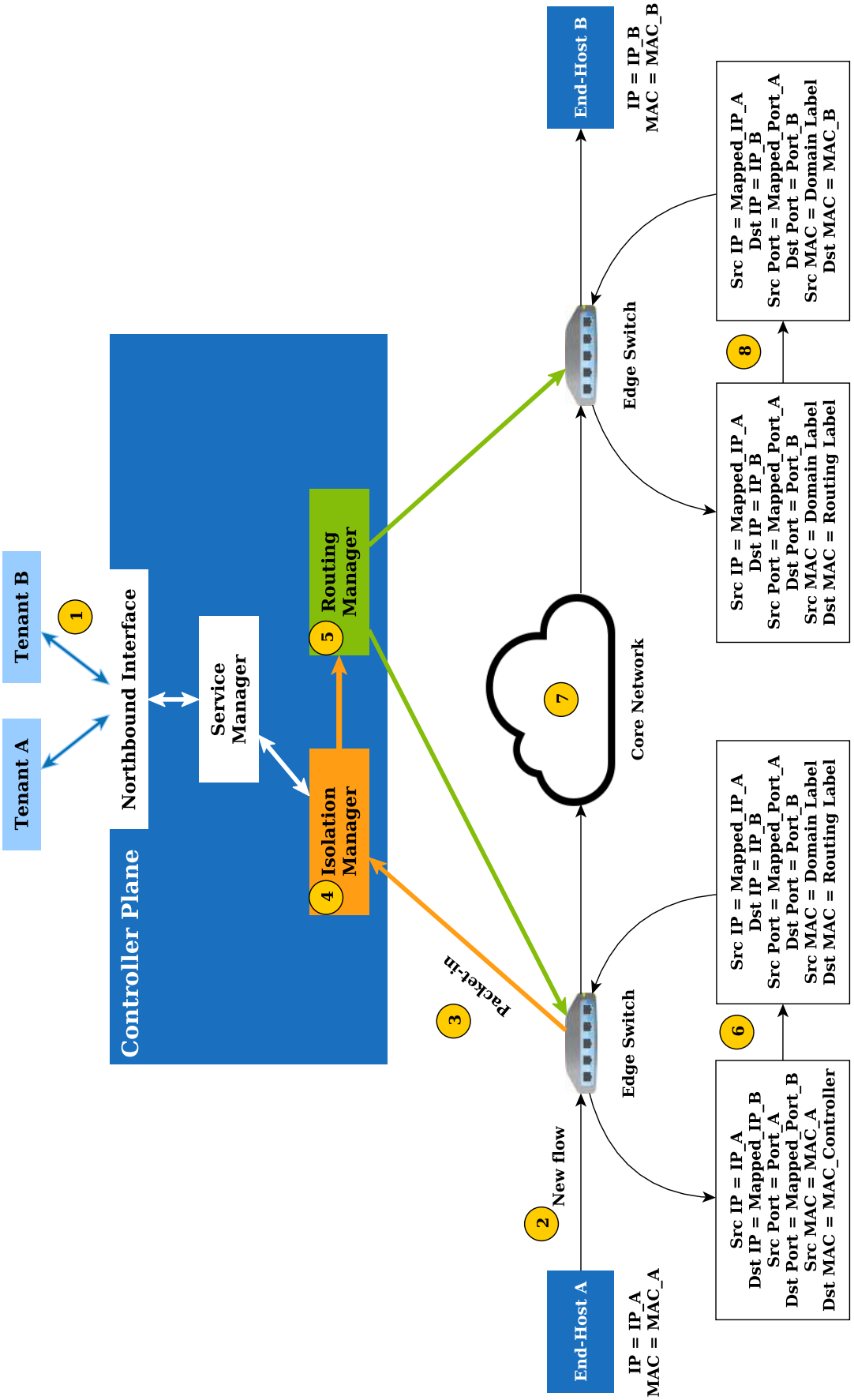

Figure 4.6: The operational flow for the inter-tenant communication

4.5.3 External Communications

Tenants might need to make connection to the external network (e.g. Internet). For this type of communication, tenants need to create a policy group with the accessibility level set to "external". Moreover, tenants need the public IP address to be routable through the Internet. Therefore, for external communications, an unused IP address and port are chosen from the pool of IPv4 public addresses.

Connection from the Internal Network to the Internet

At the following, we give an example for connections from the end-hosts inside of the shared network to the public services accessible through the Internet. Figure 4.7 illustrates the operational flow for this type of communication. In this network, we assume that the end-host received the IP address from the DHCP server and the ARP handler is responding to the ARP requests.

1. Tenants should create policy groups with the accessibility level set to "external". The policy group should allow this type of communication.

2. End-host A initiates a connection to reach a service outside of the shared network.

3. The edge switch receives the new flow and subsequently, if it does not find any matching entry for the new flow, the first packet of the flow is forwarded to the controller to decide about it.

4. When the packet is received at the controller, the isolation manager queries the service manager for finding the source and destination domains. If the source domain exists but the destination domain does not exist within the shared network, it assumes that the host is trying to access a service which is not available locally. Consequently, the isolation manager checks the policy groups with the accessibility level equal to "external". Based on the results from checking the policy groups, it changes the routing policy. Subsequently, the packet and the corresponding routing policy are delivered to the routing manager.

5. The routing manager checks the routing policy. If the flow is allowed to be forwarded then, the routing manager installs forwarding rules at the ingress and egress edge switches to forward the flow. Otherwise, the flow will be dropped at the ingress switch.

6. In case the flow is allowed for forwarding, at the ingress edge switch, the source and destination MAC addresses should be changed before forwarding the flow to the core network. The source MAC address is changed to the domain label and the destination MAC address is changed to the routing label. Moreover, since the IP addresses for the external communication should be from the public IP addresses, the routing manager maps the source IP address and port number to a free public IP address and port number.

7. The flow is forwarded at the core network based on the routing label.

8. The flow is received at the egress edge switch. The egress edge switch changes the destination MAC address to the MAC address of the next router on the path and forwards the packet.

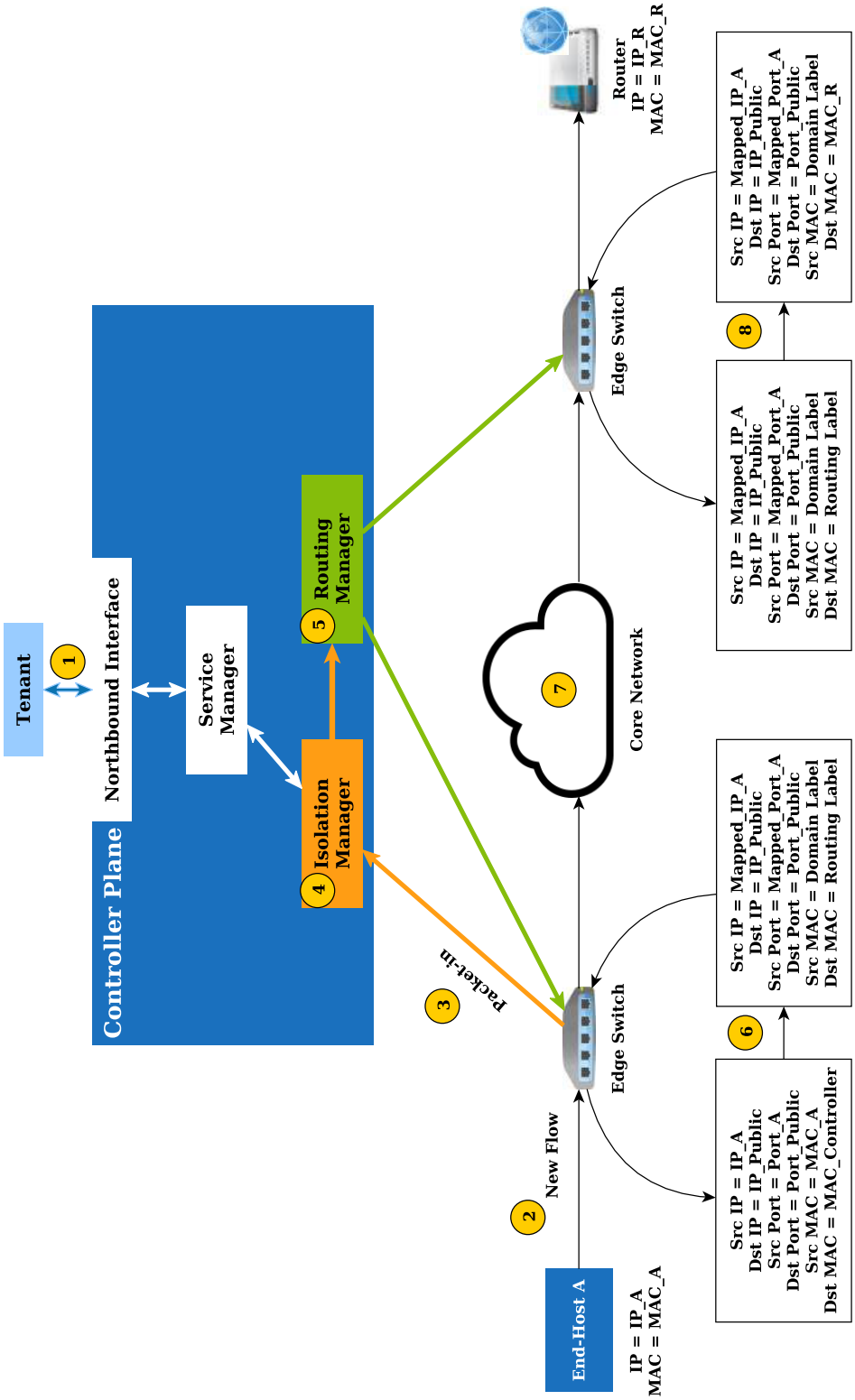

Figure 4.7: The operational flow for the communication to the external network

Connection from the Internet to the Internal Network

At the following, we give an example for the connections from the external network to the end-host inside of a shared network. Figure 4.8 depicts the operational flow for this type of communication. It should be noted that we assume this type of communication is a response to a request sent from the end-host inside of a shared network. For instance, it is a response after sending a ping request from end-host A to a public server.

1. Tenant A should create policy groups with the accessibility level equal to "external". The policy group should allow the traffic from the external network to the local end-host.

2. The edge switch receives the flow from the external network and checks the flow table for finding any matching entry for the new flow. If it cannot find any matching entry for the new flow, the first packet of the flow is forwarded to the controller.

3. At the controller, the packet is received by the isolation manager and it asks the service manager for finding the source and destination domains. If it finds the destination domain for the received packet but it does not find the source domain, it means that the flow is received from the external network and the destination of the flow is available locally. Consequently, the isolation manager checks the policy groups with the accessibility level set to "external". Based on the result from checking the policy groups, it changes the routing policy. Afterward, the packet with the corresponding routing policy is delivered to the routing manager.

4. If the routing policy allows this type of communication, the routing manager installs forwarding rules at the ingress and egress edge switches for forwarding the flow. Otherwise, the flow will be dropped at the ingress edge switch.

5. In case the flow is allowed for forwarding, at the ingress switch port, the source and destination MAC addresses should be changed before forwarding the flow to the core network. The source MAC address is changed to the domain label and the destination MAC address is changed to the routing label. Additionally, the routing manager changes the destination IP address and destination port number to the real IP address (private IP address) and port number for end-host B.

6. The flow is forwarded at the core network based on the routing label embedded in the destination MAC address.

7. The flow is received at the egress edge switch and it changes the destination MAC address to the MAC address of end-host A and forwards the flow to the destination end-host.

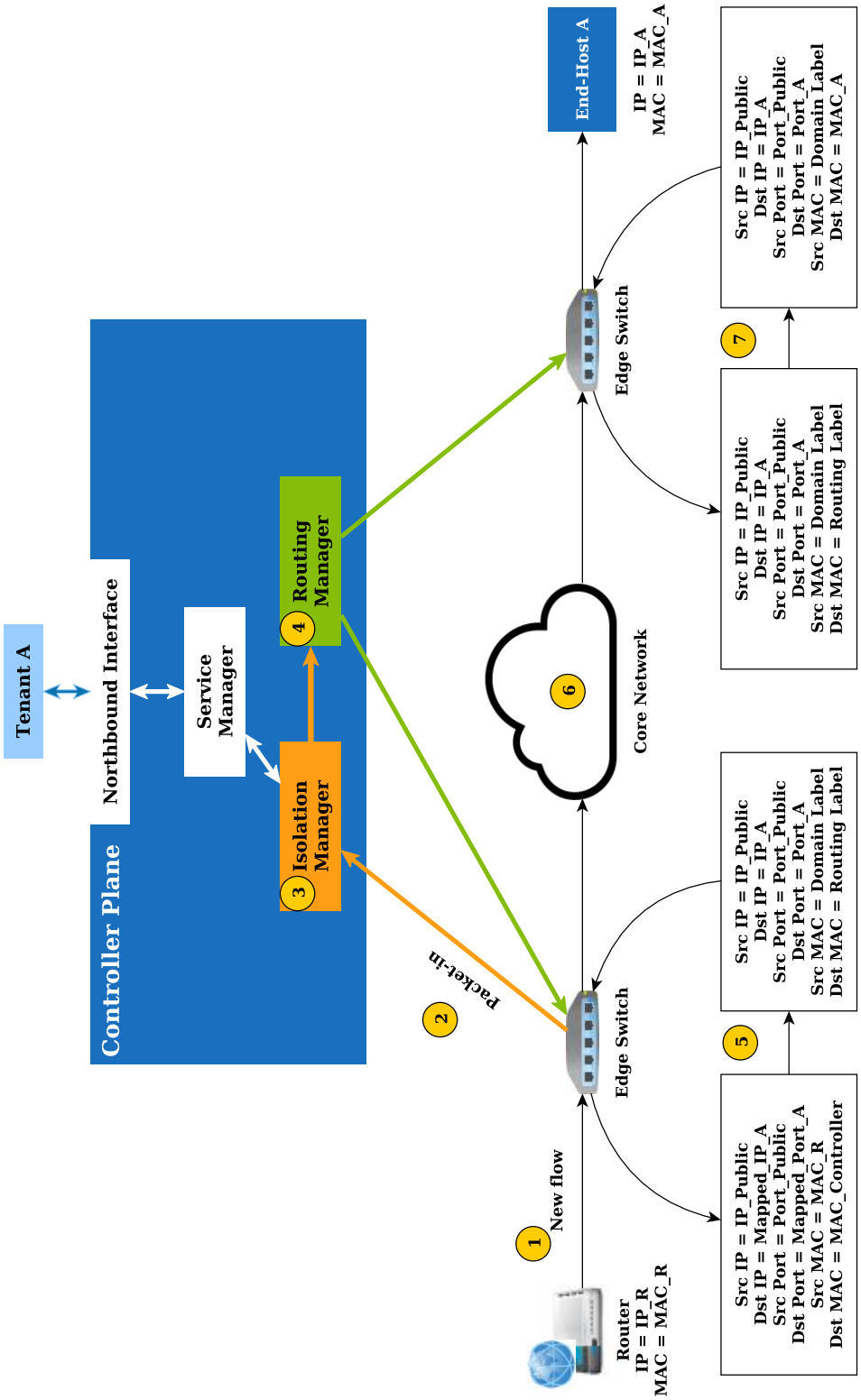

Figure 4.8: The operational flow for the communication from the external network

4.6 Chapter Summary

A scalable solution is proposed in this chapter to increase the isolation level in a multi-tenant SDN network. The new system architecture is based on the concept of domains and it assigns a domain with a unique domain ID to each tenant. In the new solution, the flows are verified at the edge of the network and the domain labels and the routing labels are embedded in the packet headers. The domain label is used for providing isolation between different tenants and the routing label is used for aggregating several flows and route them through the network. Additionally, the new architecture allows tenants to control their network by connecting to the SDN provider network and making their own configurations. While this solution provides isolation between tenants, it does not limit tenants to their own networks and allows them to interact with each other and with the external resources outside of a shared network. The main components in the proposed system architecture are the northbound interface, isolation manager, routing manager, service manager, monitoring, ARP handler and DHCP server.

Chapter 5

Implementation of the Prototype

In this chapter, we present a detailed description about the implementation of our architecture. At first, we explain about the application development with the OpenDaylight controller and then we discuss the implementation of different modules in our architecture.

5.1 Implementation Environment

Our prototype is implemented on the OpenDaylight controller. The version of OpenDaylight controller that we used for this work is *Hydrogen*. We gave a brief description of the architecture of the OpenDaylight controller in Chapter 2. In this section, we explain the implementation details for deploying new modules on the controller. The OpenDaylight controller is based on Java language and it uses strong development tools and frameworks like OSGi [28] and Maven [29]. The application development on the OpenDaylight controller is based on the OSGi framework which allows to define modular applications. Using OSGi, each module can be installed, uninstalled, stopped and started without stopping the whole SDN controller. These modular applications defined on the OSGi framework are named *bundles*. Each bundle is a jar file which includes an activator to register the bundle to the OSGi framework. All services in OpenDaylight are bundles. Bundles can offer services to each other by making Java interfaces and registering to the OSGi framework.

Figure 5.1 shows the operational states of a bundle. At first, each bundle should be installed. If the OSGi framework resolves the bundle, it will be placed in the resolved state. From this step, the bundle can be started and subsequently, it goes to the active state. If the bundle is stopped, it goes to the resolved state again. To remove the bundle from the controller, it should be uninstalled.

For our architecture, we made two bundles on the OpenDaylight controller. One bundle is used for northbound interface and the other bundle includes the rest of components in our architecture including the isolation manager, routing manager, ARP handler, DHCP server and service manager. The total number of code lines in our prototype is about 6000 lines of Java code.

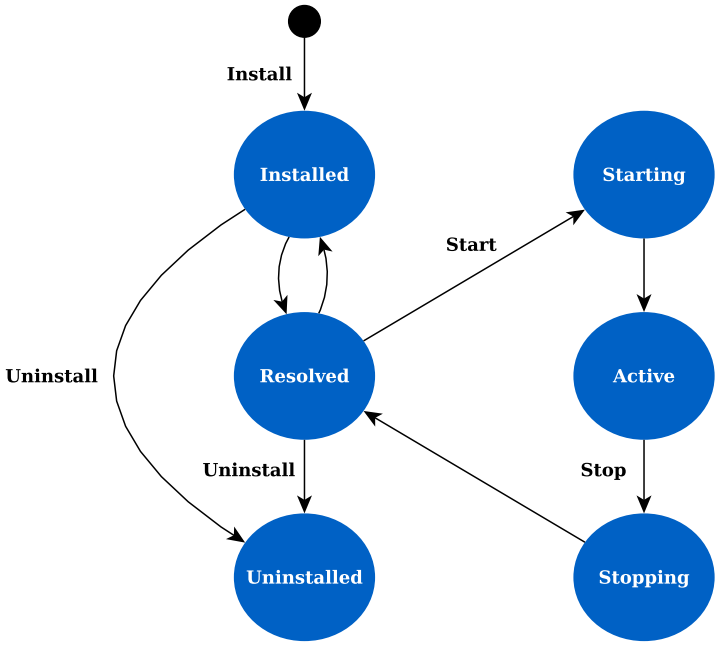

Figure 5.1: The operational states of a bundle in the OSGi framework

5.2 Implementation of the Northbound Interface

In our prototype, the REST API is used for the definition of the northbound interface. We gave a brief explanation about the REST API in Section 2.1.1. The reason for choosing REST API in our architecture is that it is widely supported by different controller platforms. Moreover, the nature of REST API leads to higher simplicity and functionality for interaction with the SDN controller. The REST API hides the complexity of the underlying network and allows tenants to simply make modifications in their network. In the OpenDaylight controller, JAX-RS is used to define the REST API. JAX-RS is implemented based on Jersey library which allows to define data in both of JSON and XML formats.

The northbound interface in our prototype can be secured by the usage of TLS. In this case, the administrator and tenants should make certificates for connecting to the controller. While the unsecure northbound interface is accessible at port 8080, the secure connection to the northbound interface should be established on port 8443.

At the following, we explain the definition of important configurations in our prototype. The list of all APIs is provided in Appendix A.

Tenant Registration:

Table 5.1: REST request for registering a tenant

```
{
  "Tenant_Name":"<Tenant name>",
  "Password":"<Tenant password>",
  "Roles":["<Tenant Role>"],
  "Inter-Tenant_Address_Size":"<Size of addresses>",
  "External_Address_Size":"<Size of addresses>",
  "Switch_Ports":["<List of edge switch ports>"],
  "Violation_Action":"<Action in case of violation>"
}
```

As we mentioned in this chapter earlier, the administrator in the SDN provider network should register tenants to the network. For registering a tenant to the network in our prototype, the SDN provider needs to add the information illustrated in Table 5.1. Based on the received request, our prototype creates a domain and the created domain includes all tenant's registration information. At the following, we explain the details of information for registering a tenant:

- **Tenant name, Password and Roles:** The SDN provider needs to add information about the new tenant to the network. Each tenant has a unique name. Moreover, a tenant needs to be assigned a password. This password is used for authentication in future contacts by a tenant. Additionally, each tenant has a role. The role helps to limit the scope of a tenant to special configurations so tenants are not able to affect the configurations made by the SDN provider. We choose the "Network-Operator" as the role of a tenant in the OpenDaylight controller. This role limits the scope of a tenant to special configurations.

- **Inter-Tenant and External Addresses:** The SDN provider should specify the number of inter-tenant and external addresses. The inter-tenant addresses are allocated from the list of free addresses in a range of 10.128.0.0/9 and 172.16.0.0/12[1]. In addition, the SDN provider needs to assign a number of external addresses to a tenant. The external addresses are allocated from the list of available public IP addresses.

- **Switch Ports:** It specifies the list of ports on edge switches belong to a tenant. As a result, tenants can connect their hosts to these switch ports. Since our approach is based on the usage of virtual switches at the edge of the network, these ports are considered as virtual ports connected to the end-hosts.

- **Violation Action:** The SDN provider should specify the violation action for a tenant. This violation action is used when the monitoring section finds an unusual traffic on the network. This option can be *Blocking* or *Warning*. In case of warning, the monitoring section warns the administrator by logging a warning message. In case of blocking, in addition to warning, the monitoring

[1]We discussed about the inter-tenant communications in Section 4.5.2

blocks the detected unusual traffic. We will explain the functionality of the monitoring component in our prototype in Section 5.8.

Creating a Subnet:

Table 5.2: REST request for setting a subnet

```
{
  "Subnet":"<Subnet/Mask>",
  "Static_IP":["<IP address , MAC address>"]
}
```

Tenants should specify a subnet for their end-hosts. As we explained in Section 4.5.1, tenants can choose any subnet in range of 10.0.0.0/9 for their end-hosts. When tenants set a subnet, a new DHCP pool is created for a tenant. This DHCP pool is mapped to the corresponding domain. Moreover, tenants can specify a list of static IPs. The static IPs are reserved for hosts based on the MAC address. The information for setting a subnet in our prototype is showed in Table 5.2.

Creating a Policy Group:

Table 5.3: REST request for creating a policy group

```
{
  "Name":"<Policy group name>",
  "Access_Levels":["<List of Access levels>"]
}
```

Table 5.3 illustrates the information for creating a new policy group. A policy group is created by a name and access levels. The name should be unique in each policy group. This name is used for further modifications by tenants. An access level limits the scope of a policy group to intra-tenant, inter-tenant or external communications.

Creating an Isolation Policy (Rules) For a Policy Group:

Table 5.4: REST request for adding rules to a policy group

```
{
  "Policy_Group":"<Policy group name>",
  "Rule_Name":"<Rule name>",
  "Source_IP_Address":"<IP address>",
  "Destination_IP_Address":"<IP address>",
  "Protocol" : "<Protocol name>",
  "Source_Port":"<TCP/UDP port / ICMP type>",
  "Destination_Port": "<TCP/UDP port /ICMP code>"
}
```

As we explained in Section 4.3.4, an isolation policy consists of a set of rules. These rules are used for allowing the traffic in the network. The configuration information for adding a new rule to a policy group is showed in Table 5.4. For each

rule, our prototype assigns a *rule ID* and all created rules by a tenant are mapped to the corresponding policy group.

Advertising a Service:

Table 5.5: REST request for service advertisements

```
{
  "Description":"<Description about the service>",
  "IP_Address":"<IP address of the service>",
  "Port":"<Transport port number of the service>",
  "Protocol":"<Protocol of the service>"
}
```

The service advertisement is used for advertising an inter-tenant service in the network. We explained about the service advertisement in Section 4.5.2. Tenants can advertise a service to other tenants and others can join the service. The information for advertising a new service is showed in Table 5.5.

Network Configuration:

Table 5.6: REST request for setting network configurations

```
{
  "Gateway_MAC_Address":"<MAC address>",
  "Gateway_Input_Port":"<Switch port>",
  "External_Address_List":["<List of IP addresses>"]
}
```

The SDN provider is responsible for making network configurations. Table 5.6 shows the information for setting network configurations. The configurations include the gateway IP address, MAC address and switch port number. The reachability information of the gateway is used for routing the traffic outside of the network. Moreover, the configurations include a list of all external IP addresses which will be used by tenants for external communications.

5.3 Implementation of the Service Manager

All configurations from the northbound interface are processed by the service manager. One of the main tasks of the service manager is to manage domains for different tenants. Figure 5.2 shows the data flow diagram for managing domains using the service manager. According to this figure, the administrator of the network creates domains. When the service manager receives the request from the administrator, it creates a new domain based on the tenant information. After this step, a tenant is able to make configurations in the domain.

When a tenant makes a new configuration, the service manager finds the domain of a tenant by its name and updates the corresponding domain based on the new configurations. Additionally, the service manager is responsible for finding the corresponding domain for the received flow requests (packet-in messages) from the

data plane. At the next section, we explain about the domain discovery mechanism in the service manager.

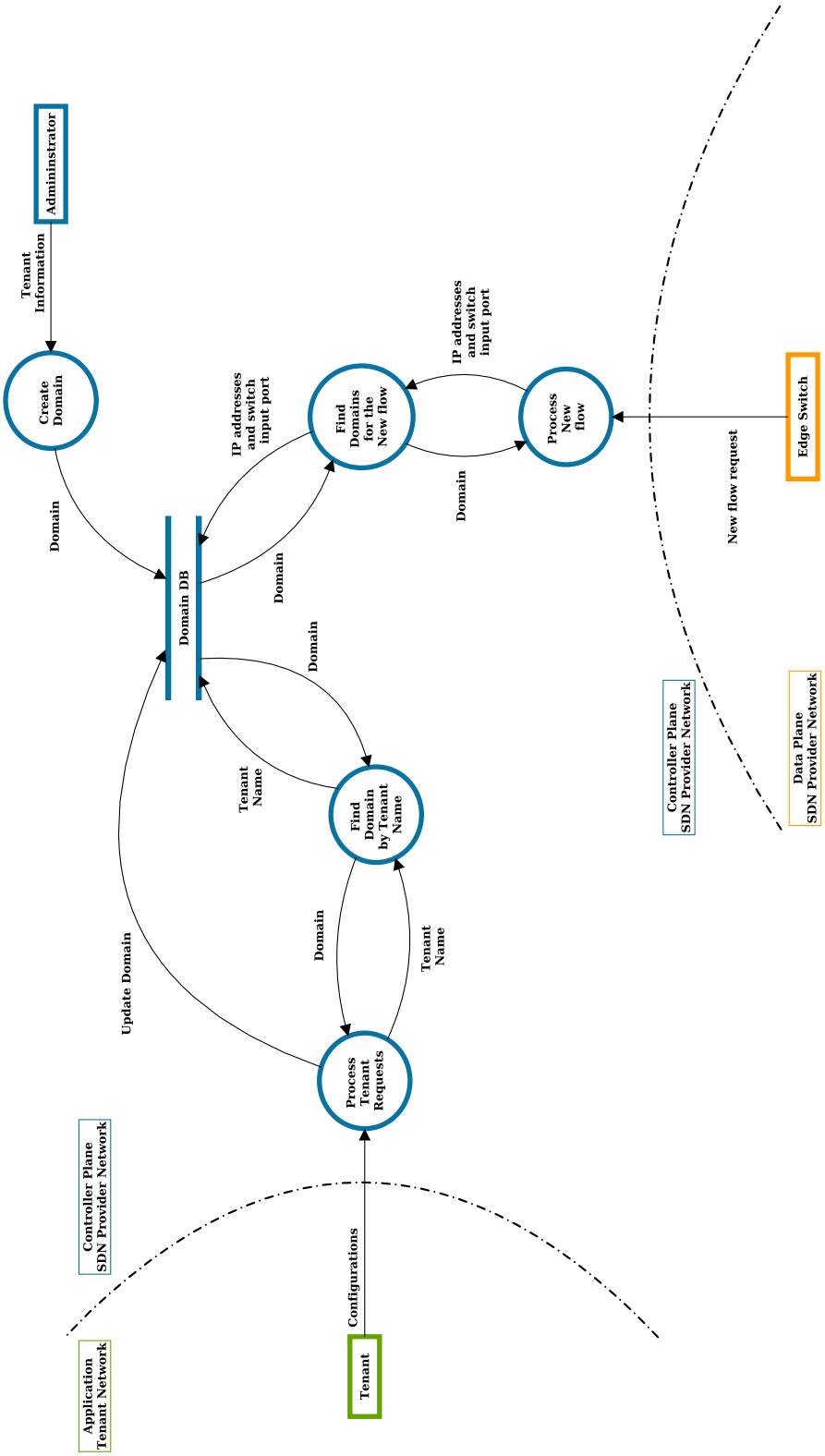

Figure 5.2: Domain management in the service manager

5.4 Domain Discovery Mechanism

Since all configurations for a specific tenant are attached to its domain, the domain discovery is the first functional step of our architecture. We defined the concept of domain in Section 4.3 and in this section, we explain in more detail the process of finding a domain using the service manager.

5.4.1 Domain Discovery for Northbound Requests from Tenants

The domain discovery for the received requests from the northbound interface depends on the tenant's name. As we stated in Section 5.2, this name should be unique. By using the tenant's name, we can easily find the corresponding domain.

5.4.2 Domain Discovery for Flow Requests from the Data Plane

Since our architecture supports the intra-tenant, inter-tenant and external communications, for each received flow request, we need to find the source and destination domains. At the following, we explain the procedures for discovering the source and destination domains.

Finding the Source Domain

As we explained in this chapter earlier, each domain includes a list of switch input ports. When we receive a packet-in message for a new flow request from the data plane, the packet-in message includes information about the switch input port. In our prototype, we use the switch port in packet-in messages for finding the source domain.

Finding the Destination Domain

For finding the destination domain, we cannot use the switch input port since the packet-in message only includes information about the input port of the ingress switch. As a consequence, for discovering the destination domain, we need to check the source and destination IP addresses. Since our solution supports the intra-tenant, inter-tenant and external communications, we need to check individually for assigned IP addresses in all three types of communications. For finding the destination domain, we have used the algorithm depicted in Figure 5.3. The first step is to check if the IP address is in a range of intra-tenant IP addresses (10.0.0.0/9). In this case, the source and destination domains are equal since this range of IP addresses are used for intra-tenant communications. If the connection is not intra-tenant, then we check for the inter-tenant connections. For checking the inter-tenant connections, we compare the destination IP address with allocated inter-tenant IP addresses. Finally, if we cannot find the destination domain from the inter-tenant IP addresses, we compare the destination IP address with the allocated external IP addresses.

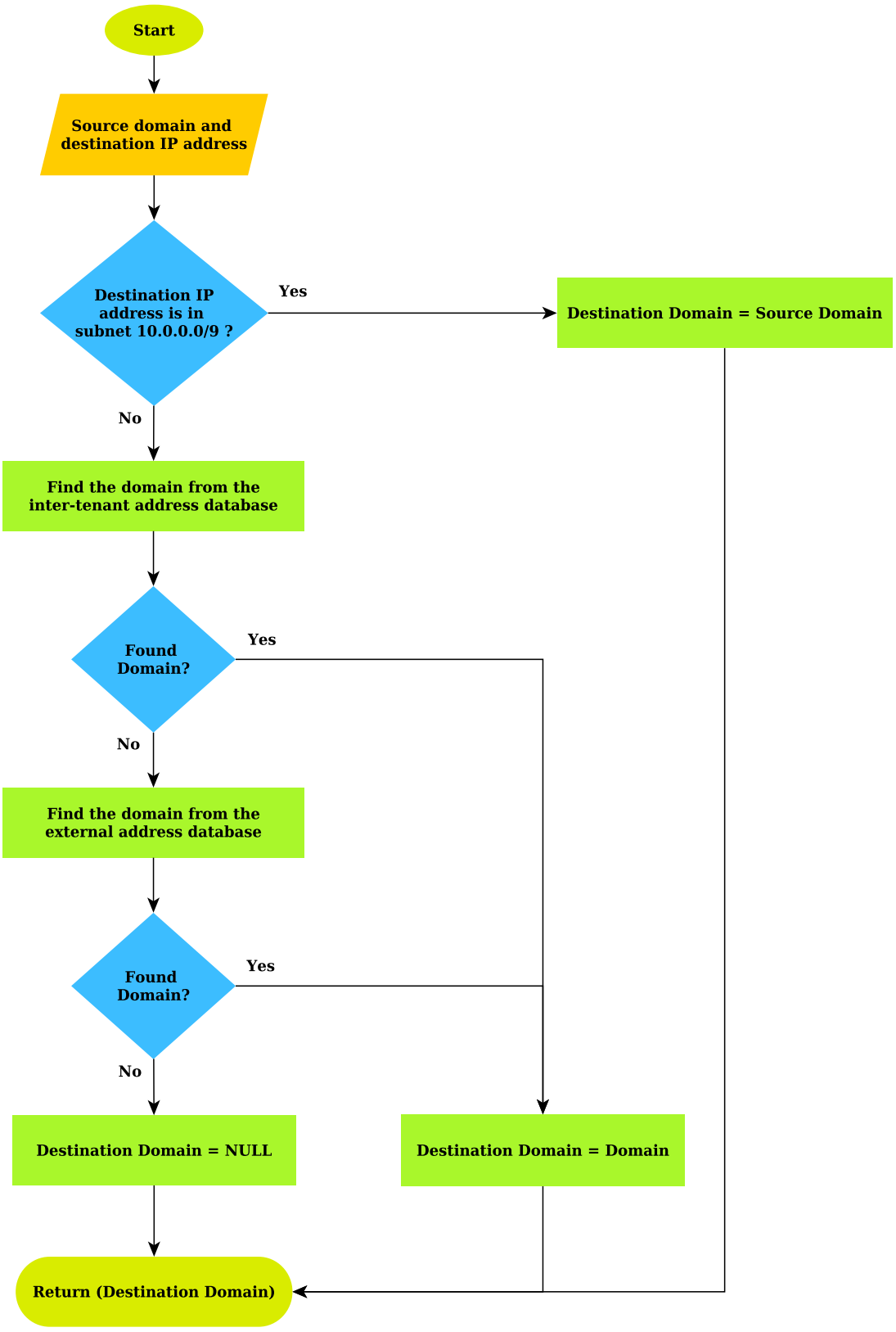

Figure 5.3: Implemented algorithm for finding the destination domain

5.5 Implementation of the DHCP Server

The OpenDaylight controller does not support the DHCP packet structure. For defining the DHCP packets on the OpenDaylight controller, we implement the serialize and deserialize functions on the UDP packets. In fact, we receive the UDP packets with the source port 67 and the destination ports 68 in the DHCP server and subsequently, we deserialize the UDP packets and extract the remaining bytes in the payload and then serialize it to make the DHCP packets.

The internal structure of the DHCP server is fairly according to the RFC 2131 [24]. The DHCP Server might receive five types of messages from the client: DHCP Discover, DHCP Request, DHCP Decline, DHCP Release and DHCP Inform. All these messages use the source UDP port 68 and the destination UDP port 67 and the option field in the received DHCP packet shows the type of message. The DHCP server sends three types of messages to the client: DHCP Offer, DHCP ACK and DHCP NAC. These messages are encapsulated in UDP packets with the source port 67 and the destination port 68.

In our DHCP server, we use the DHCP pool to store the list of free and allocated IP addresses. Because in our architecture, we can have overlapped IP addresses and each tenant should be isolated from the other tenants, we assign a separate DHCP pool to each tenant. The DHCP pool for each tenant is attached to the corresponding domain. In fact, each domain holds a separate DHCP pool. The size of the DHCP pool is allocated based on the subnet defined by a tenant. Besides, the DHCP pool is able to assign the static or dynamic IP address to the end-hosts. In case of static IP addresses, tenants should specify the static addresses with the northbound API.

5.5.1 Host Detection with the DHCP Server

As we explained in Section 5.4, the configurations related to a tenant including the host database is stored in its domain. The DHCP server adds a reachability information for the newly detected end-host to the host database after successfully allocating an IP address and sending the DHCP ACK message to the end-host. The reachability information of the new end-host includes the IP address, the MAC address and the switch port connected to the end-host. Additionally, the DHCP server informs about the newly detected end-hosts to the routing manager. The routing manager uses the reachability information of end-hosts to detect edge switches. We will explain in more detail on how the routing manager uses this information in subsequent sections. Figure 5.4 illustrates the operation of the DHCP server for finding a new end-host.

5.5.2 Expired Leased Addresses

A separate thread in the DHCP server checks for the expired leased addresses. It checks the DHCP pool in each domain and releases the expired addresses. The released address will be added to the DHCP pool as a free address for future address allocation.

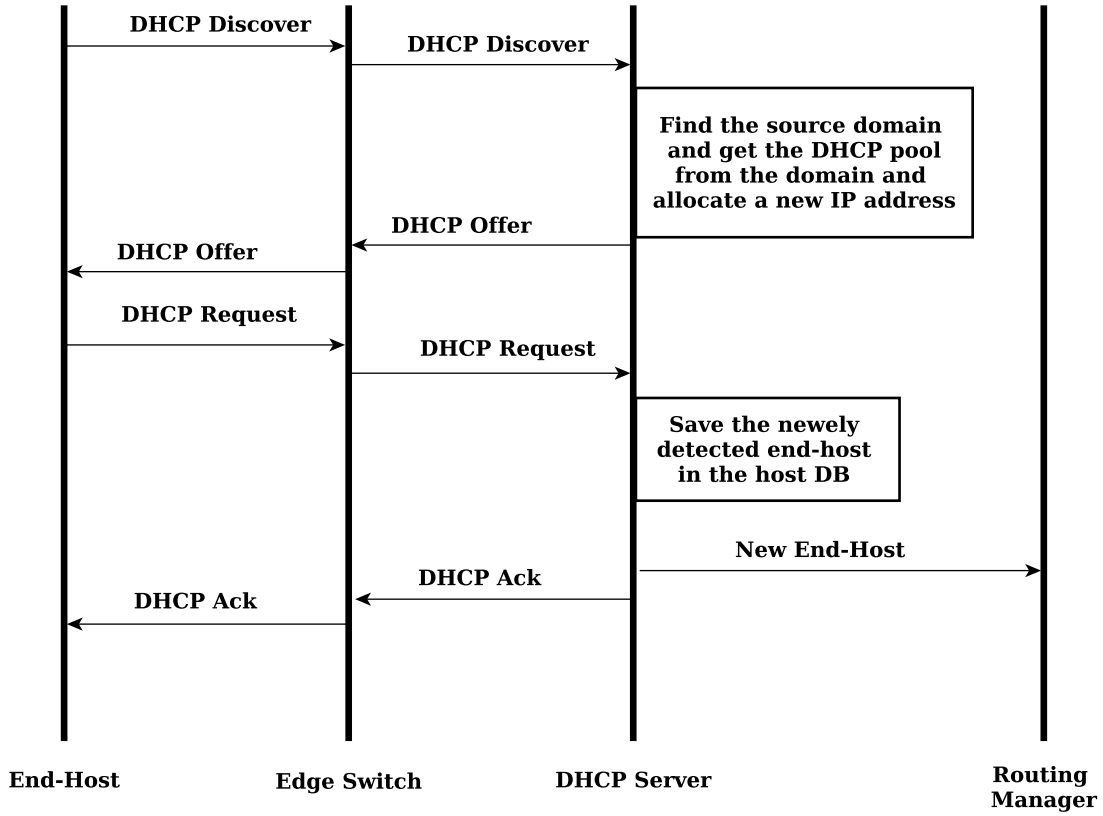

Figure 5.4: Host detection using the DHCP server

5.6 Isolation Mechanism in the Prototype

Except the DHCP and ARP packets, all other data packets in our prototype are checked by the isolation manager. As we described in Section 4.4.5, when a packet-in message is received, the isolation manager queries the service manager for finding the source and destination domains. After finding the source and destination domains, it finds the policy groups related to that type of communication. For each policy group, we compare the isolation policy (rules) with the new flow request. If the packet matches any of the rules in the policy group, the routing policy is changed for the flow to allow this type of communication and if the rule does not match, then the routing policy is changed to drop the flow. Finally, the packet and the corresponding routing policy are delivered to the routing manager.

At the following, we explain the implementation of the rule matching algorithm in our prototype.

5.6.1 Rule Matching

We explained about the definition of rules in Section 5.2. Our prototype considers a unique rule ID for each rule and then it processes and stores the rules in the corresponding policy group. The rules are processed based on the *divide and conquer* algorithm [55]. In our architecture, we divide a rule based on 5 attributes: *source IP address*, *destination IP address*, *protocol*, *source port* and *destination port* and

we place the rules with the same attribute in the same group. For classifying the rules, we benefit from *HashMap*, *IntervalTree* and *BitSet* data structures. At the following, we explain in more detail about how the information is classified.

Classify rules based on the protocol

We classify rules based on the protocol. For this purpose, rules with the same protocol number are placed in the same group. In fact, all rules with the same rule ID are placed in the same BitSet (Rule IDs) and the resulting BitSet is mapped to the protocol number using HashMap. If the value is "Any", then -1 is used as the protocol number.

Classify rules based on the transport port

The port numbers are defined in a "range". Therefore, we use the concept of Interval Trees [30] for storing and processing information about port range in a rule. In fact, for every unique range of port numbers, we add similar rules to the BitSet (Rule IDs) and the resulting BitSet is placed in our interval tree. If the value is "Any", then the range 0-65535 is used as a port range.

Classify rules based on the IP address

For storing the IP address information, all rules that have the same IP address are grouped together in the same BitSet (Rule IDs). It means, all rules with the same IP address are placed in the same BitSet and the resulting BitSet is mapped to the IP address using HashMap. If the value is "Any", then -1 is used as the IP address.

Matching a new flow with rules

For every new flow request, the source IP address, the destination IP address, the source port, the destination port and the protocol are checked. For each of them, we find the list of rules containing the information in common. For example, if the protocol of the new flow is TCP, we find the list of rules that include TCP as their protocol. This process continues for the source IP address, the destination IP address, the source port, the destination port and the protocol. At the end, the intersection of all rules is calculated. After the intersection step, we should find one rule that matches the new flow. If we find the rule, the flow is forwarded (allowed), otherwise, it should be dropped.

Assume that the attributes in a new flow request are the protocol, source IP address, destination IP address, source transport port and destination transport port. $result_{ruleID}$ is the result of rule matching function which is based on searching each of flow attributes in our data structures.

$$result_{ruleID} = \bigcap_{attribute \in flow} search(attribute)$$

5.7 Forwarding at the Data Plane

The routing manager is responsible for forwarding the flows at the data plane. As we described in Section 4.4.6, the functionality of the routing manager is different at the core and edge network. While the forwarding rules at the edge of the network are fine-grained rules including detailed information to specify a certain flow, at the core network, the forwarding rules are simpler and they are only based on the destination MAC address of each flow. In fact at the edge of the network, we isolate flows by installing fine-grained rules and at the core network, we aggregate several flows and route them through the network.

At the following, we explain about the structure of forwarding rules at the edge and the core network.

Forwarding rules at the core network

Forwarding rules at the core network are installed proactively. In fact, between two edge switches, a one-way routing path is installed at the core network. As we explained in Section 5.5, the new end-hosts are detected by the DHCP server. Upon the detection of a new end-host, the DHCP server informs the routing manager. The routing manager finds the edge switch connected to the newly detected end-host and finds the path between the new edge switch to all other existing edge switches. For a pair of edge switches, it chooses the shortest path and allocates a unique and random routing label and then it installs forwarding rules on all the switches on the path at the core network . Figure 5.5 shows the process for installing proactive rules at the core network. For finding the path between the edge switches, we used the *IRouting* service of the OpenDaylight controller. It finds the shortest path using the Dijkstra algorithm based on the topology of the network and the properties of the network elements in the data plane (i.e bandwidth for each switch interface).

The structure of the forwarding rules at the core network is depicted in Table 5.7. The match field is based on the destination MAC address (which is a routing label) and the action field for all flows is the output to a specific port.

Table 5.7: Forwarding rules at the core of the network

Switch	**Match Fields**	**Actions**[2]
Core Switch	Destination MAC address	Output

[2]The definition of these actions is explained in Chapter 2

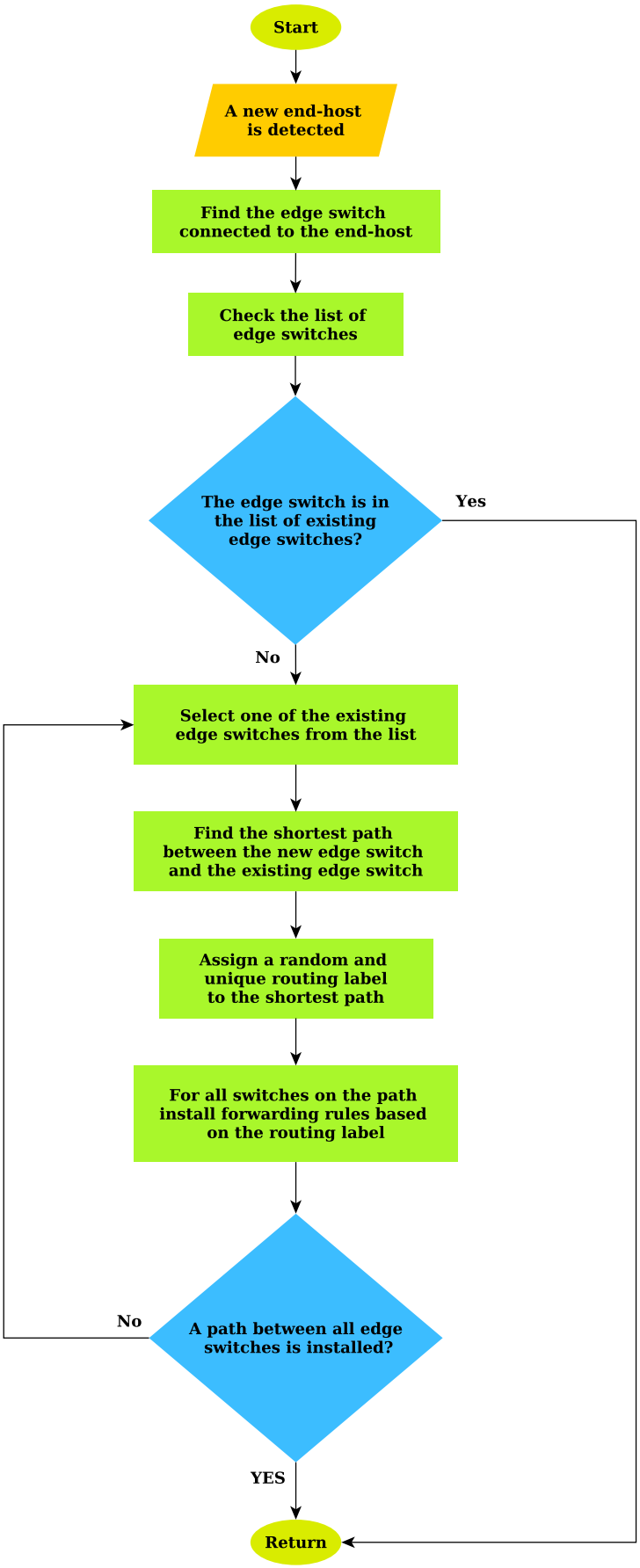

Figure 5.5: Proactive rule installation at the core network

Forwarding rules at the edge network

For providing isolation at the edge of the network, we install fine-grained forwarding rules. Figure 5.6 explains the functionality of the routing manager for installing forwarding rules on the edge switches. The routing manager receives the packet-in and the routing policy from the isolation manager. If the routing policy is to drop the flow, the routing manager installs a forwarding rule on the ingress edge switch to drop the flow. Otherwise, the routing manager finds the source and the destination end-hosts. If the source or the destination end-host is not available in the host DB, then we assume that the connection comes from the external network and we set the gateway as the source or the destination end-host. Subsequently, the routing manager checks the location of the source and the destination end-hosts in the data plane. If both of them are connected to the same switch, then it installs one forwarding rule on the ingress edge switch for forwarding the flow. Otherwise, it installs a forwarding rule on both of the ingress and egress edge switches for forwarding the flow at the core network. Moreover, after installing the forwarding rules, we send the received packet from the isolation manager to the destination (packet-out message)[3]

The fine-grained forwarding rules are created in a way to only match a specific flow on the edge switches. Each rule has a set of match fields and the match fields are different in the ingress and egress edge switches. The match fields for the ingress and egress edge switches are explained in Table 5.8. Moreover, each forwarding rule has a set of actions. These actions are performed on each flow that matches a forwarding rule on the edge switches. The list of actions for different types of communication is depicted in Table 5.9 and Table 5.10. For all flows at the edge of the network, the idle time is equal to 60 seconds and the hard timeout is 120 seconds.

Table 5.8: Match fields of forwarding rules at the edge of the network

Switch	**Match Fields of Forwarding Rules**
Ingress Edge Switch	Data link Type, Input port, Source MAC address, Source IP address, Destination IP address, Protocol, Source port number, Destination port number
Egress Edge Switch	Data link Type, Source MAC address, Destination MAC address, Source IP address, Destination IP address, Protocol, Source port number, Destination port number

[3]It should be considered that this step is not part of our architecture and we send the packet because this version of the OpenDaylight controller does not support the buffer IDs and we need to send the received packet to the destination by our prototype. Otherwise, we will lose the first packet of the flow.

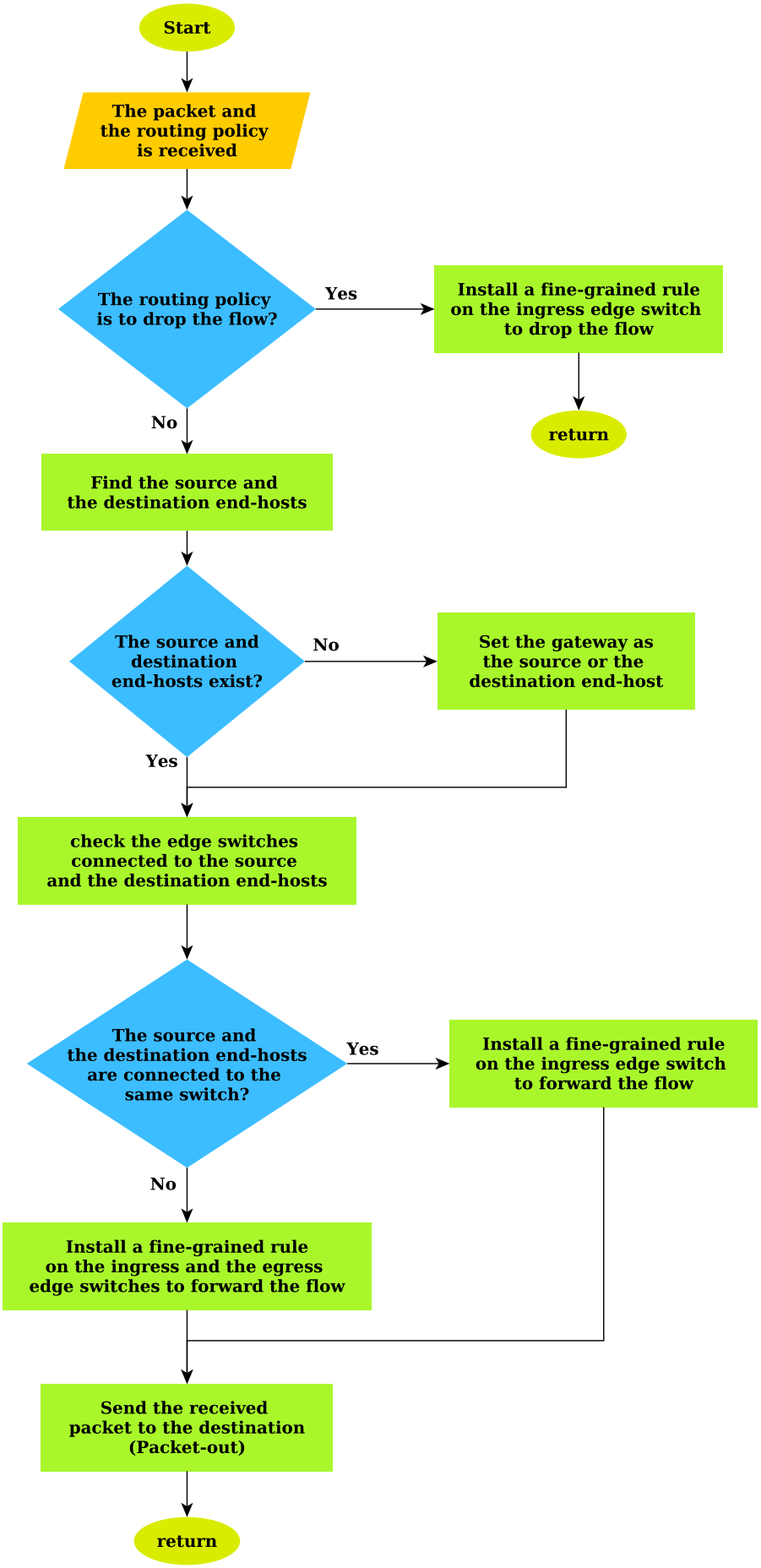

Figure 5.6: Reactive forwarding using the routing manager at the edge of the network

Table 5.9: Actions for forwarding rules at the edge of the network (the source and destination end-hosts are not connected to the same edge switch)

Switch	**Type of Communication**	**Actions**[4]
Ingress Edge Switch	Drop	No actions
Ingress Edge Switch	Intra-Tenant	SetDlSrc, SetDlDst, Output
Ingress Edge Switch	Inter-Tenant	SetDlSrc, SetDlDst, SetNwSrc, SetNwDst, SetTpSrc(TCP/UDP), SetTpDst (TCP/UDP), Output
Ingress Edge Switch	External	SetDlSrc, SetDlDst, SetNwSrc, SetTpSrc(TCP/UDP), Output
Egress Edge Switch	Intra-Tenant, Inter-Tenant and External	SetDlDst, Output

Table 5.10: Actions for forwarding rules at the edge of the network (the source and destination end-hosts are connected to the same switch)

Switch	**Type of Communication**	**Actions**
Ingress Edge Switch	Drop	No actions
Ingress Edge Switch	Intra-Tenant	SetDlDst, Output
Ingress Edge Switch	Inter-Tenant	SetDlDst, SetNwSrc, SetNwDst, SetTpSrc(TCP/UDP), SetTpDst (TCP/UDP), Output

[4]The definition of these actions is explained in Chapter 2

5.8 Implementation of Monitoring

The monitoring is responsible for detecting unusual traffic in the network. We have explained the functionality of the monitoring in Section 4.4.7. In this part, we discuss the implementation of the monitoring in our prototype. The implementation of monitoring in our prototype is based on the concept of performance aware SDN [5]. The monitoring consists of three major tasks: *Collection of statistics*, *Analysis of statistics* and *Reaction*. At first, we need to sample and collect the statistics from the switches at the data plane. The next task is to analyze the statistics and finally, in a case of detection, we should react to the unusual traffic. These statistics are collected from the edge switches at the data plane and are transferred to the analyzer using sFlow protocol [27]. The analyzer in our implementation is based on sFlow-RT [6]. sFlow-RT analyses the statistics in real-time and in case of violation in the traffic, it makes new events for the detected violation. The reaction tasks in our monitoring are implemented in the monitor manager. The monitor manager receives the information about the newly detected violations and reacts to the detected issues. For connecting sFlow-RT and monitor manager, we wrote an application with Node.js [7].

The process of the monitoring module for detecting the violations is demonstrated in Figure 5.7. According to the figure, our protection application configures sFlow-RT to detect violations. From this step, sFlow-RT analyzes all sampled data received from the edge switches. If sFlow-RT finds a violation, it creates a new event. Our protection application queries the new events by the REST API provided by sFlow-RT and in case of new events, it informs the monitor manager on the OpenDaylight controller on a UDP channel to react to the detected issues. At the following, we explain more details about the main components of our monitoring module.

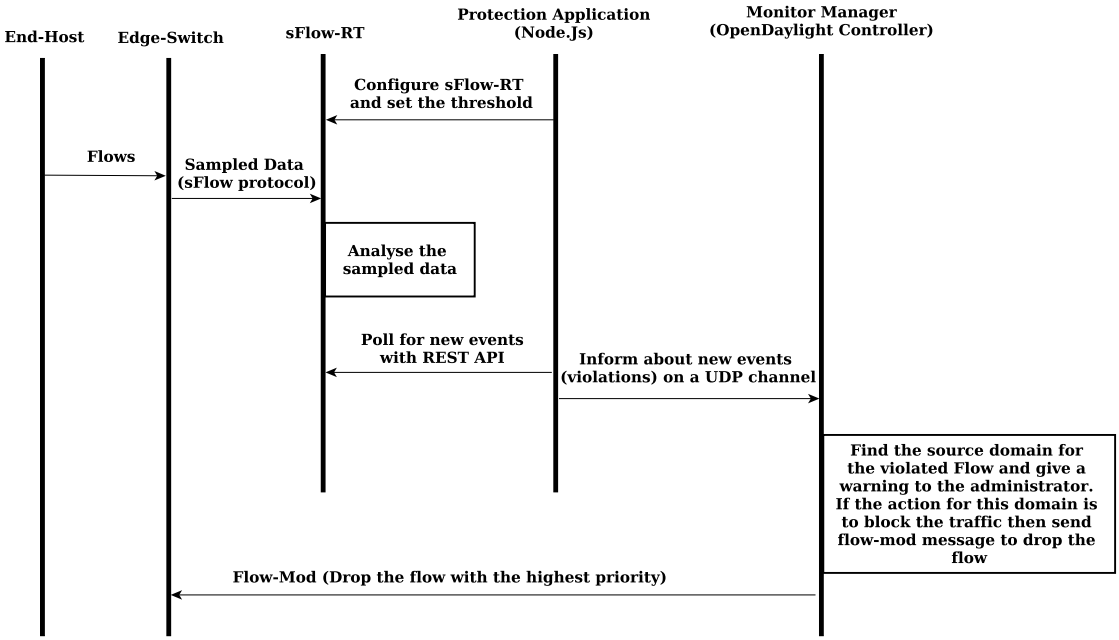

Figure 5.7: Message diagram for monitoring process in the prototype

5.8.1 sFlow-RT

sFlow-RT provides real-time monitoring and analysis of data traffic at the data plane. For monitoring the traffic, sFlow-RT supports the definition of groups, flows and thresholds. With groups and flows, we can specify specific attributes to be monitored. For instance, we can specify to monitor a specific subnet that carries special type of traffic (such as TCP). The threshold defines the level that should be exceeded for making notifications about the detected flows. In fact, sFlow-RT monitors the flows based on the flow definition and if the flows exceed the threshold, it produces a notification about the detect flows. Additionally, sFlow-RT provides a set of REST APIs for interaction with applications. The REST API allows the external applications to query new detected events.

5.8.2 Protection Application

Our protection application is written using Node.js. At first, this application should set flows, groups and a threshold on sFlow-RT. We set flows in a way to receive all information about flows routed at the data plane. This includes switch input port, source IP address, destination IP address, source port number and destination port number. After the initialization phase, this application sends queries every 60 seconds and if it finds any event from sFlow-RT, it sends the information about the detected violations to the monitor manager on a UDP channel.

5.8.3 Monitor Manager

The monitoring manager is implemented on the OpenDaylight controller. This unit is listening to the notifications from sFlow-RT and in case of new events, it makes reaction to the detected issues. As we discussed in Section 5.2, the administrator of the network should set the violation action for a tenant during the registration phase. In fact, the monitor manager uses this value to react to the detected events. When the monitor manager detects a new event, it finds the domain based on the received information about the violated flows from sFlow-RT and checks the violation action for the corresponding domain. If the action is *Warning*, then the monitor manager logs the error and makes a notification on the console for the administrator of the network. If the violation action is *Blocking*, the monitoring manager shows a warning to the administrator of the network and also drops the flow at the edge of the network. For dropping a flow, we install a Flow-Mod message based on the received information about the violated flow with the highest priority on the ingress edge switch.

5.9 Implementation Issues and Challenges

The first challenge was the usage of sFlow and OpenFlow together. The sFlow protocol uses *ifindex* to identify switch ports. However, the switch ports in OpenFlow are determined by *ifname*. As a result, there is not anyway to find the ifname of the switch ports from the information received by the sFlow protocol from the switches at the data plane. To overcome this problem, we made a script for translating ifindex

number to ifname in Mininet. In future, it is expected to integrate ifindex number of switch ports in the OpenFlow protocol.
The other issues in the deployment of this solution were mainly related to the OpenDaylight controller. Unfortunately, the Hydrogen version of the Opendaylight controller is not mature and we experienced some limitations during the implementation of our prototype. With the current version of the OpenDaylight controller, we couldn't use the buffer IDs in our prototype. As a consequence, we should send all the packet-in messages to the destination by our prototype. Moreover, the OpenDaylight controller does not support the DHCP packet structure. To solve this problem, we used the UDP packets and the serialize and deserialize functions to make the DHCP packets.

5.10 Chapter Summary

This chapter describes the implementation of the prototype using Java language on the OpenDaylight controller. This chapter also discusses the implementation of the northbound interface using the REST API. The REST API in the prototype allows the SDN provider to configure the network and register tenants. Moreover, it allows tenants to define the subnet, the policy groups and the rules for their domains. The next part of this chapter is about the implementation of the domain discovery mechanism for processing different requests from the northbound interface and from the network elements in the data plane. Additionally, the structure of the DHCP server is explained in this chapter. The DHCP server assigns a separate DHCP pool to each tenant and allocates IP addresses to the end-hosts.

The next part of this chapter focuses on the isolation mechanism in our prototype. In this prototype, for verifying the flow requests, a rule matching process is used which is based on the use of the divide and conquer algorithm. After verifying the flows, the routing manager installs fine-grained rules at the edge of the network to forward specific flows through the core network. At the core network, the routing manager installs forwarding rules based on the destination MAC address to aggregate several flows. Additionally, the monitoring module in this prototype inspects the traffic using sFlow-RT in real-time to detect violations from the tenants at the data plane.

Chapter 6

Evaluation and Experimental Results

In this chapter, we evaluate the implemented prototype in several aspects. At first, we explain about how effective is our approach in terms of isolation in a multi-tenant SDN and then we run several test scenarios to show the functionality of our prototype in practice. In the next section, we focus on the scalability, overhead and latency of our approach. We will finish this chapter by discussing the advantages and disadvantages of our solution.

6.1 Analysis of Isolation Enforcement

In this section, we analyze our solution in terms of isolation in a multi-tenant SDN. Based on our design goals in Section 4.1, we have defined isolation requirements for a multi-tenant network in four different aspects: *Traffic Isolation*, *Address Space Isolation*, *Control Isolation* and *Performance Isolation*. At the following, we discuss the effectiveness of our solution to provide the aforementioned isolation aspects.

6.1.1 Traffic Isolation

In our approach, all flows should be verified at the edge of the network based on the defined policy by the tenants. In fact, before routing any traffic through the network, we guarantee that only verified flows by tenants are able to be routed in the network. Moreover, after the verification process, we embed domain labels to all flows at the edge of the network. Since the core network is shared between all tenants, these unique domain labels are used at the destination to differentiate several flows of different tenants. Additionally, we have removed the broadcast ARP messages from our network which violates the isolation between tenants. The combination of all these features leads to strong traffic isolation for tenants in a shared network.

6.1.2 Address Space Isolation

As we explained earlier, our solution rewrites packet headers and embeds domain labels to all verified flows at the edge of the network. With these unique domain

labels, we can easily differentiate the similar flows of different tenants at the destination. As a consequence, tenants are allowed to have overlapped addresses in a shared network. However, as we explained in Section 4.5.1, while tenants can choose any MAC addresses, or transport port numbers, the overlapped IP addresses should be in a range of 10.0.0.0/9.

6.1.3 Control Isolation

In our solution, tenants have the complete control over their accounts and all configurations created by a tenant are mapped to the corresponding domain. All tenants in our solution are authenticated by the tenant name and the password to only allow registered tenants to connect to the SDN controller. Additionally, for every authenticated tenant, we have provided a network role which limits the scope of a tenant to special configurations. As a result, tenants are not able to affect the global administrative configurations of the network. Moreover, all successful and unsuccessful attempts for creating configurations from tenants are logged for later auditing purposes by the administrator of the SDN provider. On the other hand, the northbound API is secured by the TLS protocol which reduces the chance of Man-in-the-Middle attacks by other tenants or external entities. The combination of all these features leads to control isolation between tenants.

6.1.4 Performance Isolation

Our system provides performance isolation by monitoring the traffic of all tenants at the data plane. All originated traffic at the edge of the network is monitored with sFlow-RT and in case of violation, the monitor manager on the OpenDaylight controller reacts to the violation. One of the biggest advantages of sFlow-RT is the detection time which is almost near to real-time. The combination of sFlow-RT and SDN allows the SDN provider to detect and react to violations in real-time. The effectiveness of sFlow protocol and particularly the sFlow-RT is studied in [5, 6].

6.2 Functional Testing

In this section, we evaluate our solution based on the different test scenarios. In our test scenarios, we use Mininet [13, 60] for emulating the data plane. Mininet is an emulation tool which provides accurate results and we use it for emulating sample networks for our test scenarios. Mininet emulates a network by the use of Open vSwitches [8] and Linux containers [10]. Mininet is connected to the OpenDaylight controller where we have implemented our prototype. The whole system is running on a computer with 64 bit Ubuntu 14.04, Core i7 CPU M 640 @ 2.80 GHz x 4 and 8 GB RAM.

6.2.1 Test Scenario 1: Isolation in Intra-Tenant Communications

In this test scenario, we want to show the functionality our solution in providing isolation for the intra-tenant communications. This test scenario will focus on the traffic isolation and the address space isolation between tenants in the intra-tenant connections. The operational flow of our system architecture for the intra-tenant communications illustrated in Section 4.5.1 . For this test scenario, we have implemented a sample network depicted in Figure 6.1. In this network, we have 3 tenants named A, B and C and each tenant has two end-hosts. End-hosts A-1 and A-2 belong to tenant A, end-hosts B-1 and B-2 belong to tenant B and end-hosts C-1 and C-2 belong to tenant C. End-hosts A-1, B-1 and C-1 use the same IP address and MAC address and end-hosts A-2, B-2 and C-2 use the same IP address and MAC address. For testing the isolation, we use Iperf [11] to make simultaneous connections from end-hosts A-1, B-1 and C-1 to end-hosts A-2, B-2 and C-2. End-hosts A-1, B-1 and C-1 are clients trying to make a TCP connection with their servers A-2, B-2 and C-2, respectively. It should be noted that the end-hosts are registered to the system using our DHCP server and we have added the policy groups and the required rules to allow this type of communication for all tenants beforehand.

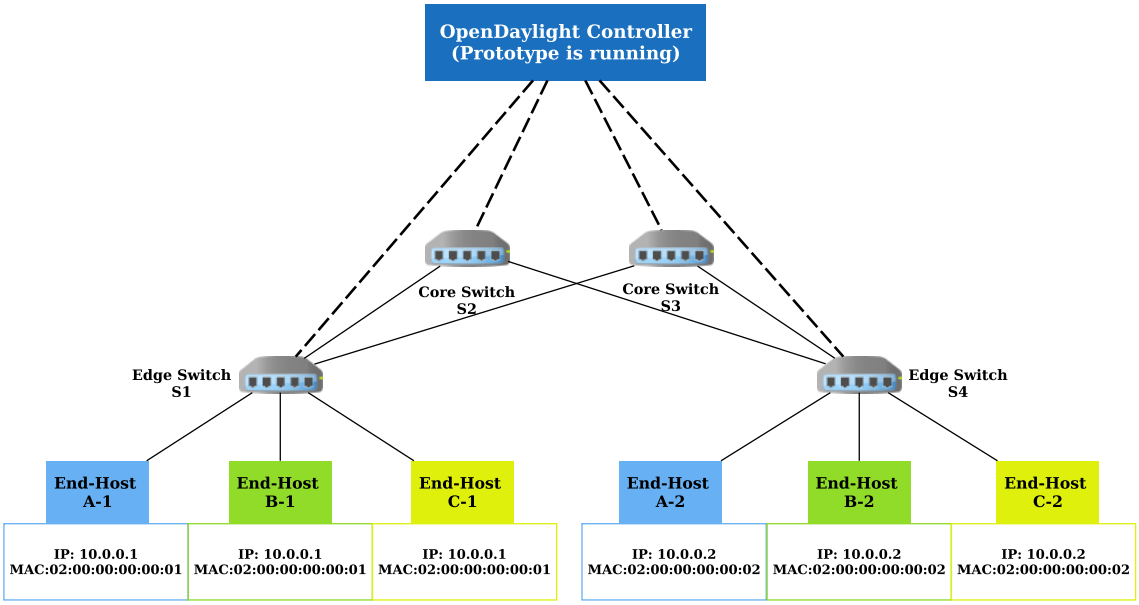

Figure 6.1: Test network for intra-tenant communications

Analysis of Results

As we expected, all end-hosts A-1, B-1 and C-1 were able to successfully connect to the end-hosts A-2, B-2 and C-2, respectively. As we explained in Section 4.5.1, for the intra-tenant communications, the packet headers in all flows are rewritten at the ingress edge switch. As a result, in this type of communication, the ingress edge switch (S1) rewrites the packets and embeds domain labels to the source MAC

address and the routing labels to the destination MAC address of all packets. At the core network, the flows are routed based on the routing label and at the egress edge switch (S4), the flows from different tenants are differentiated based on the information in the packet header. Figure 6.2, Figure 6.3 and Figure 6.4 depict the sample captured packets at the core network. According to the figures, since the selected path from end-hosts A-1, B-1 and C-1 to end-hosts A-2, B-2 and C-2 is the same, all packets use the same routing label (destination MAC address). However, the domain label is different between tenants (source MAC address).

```
Frame 16745: 1514 bytes on wire (12112 bits), 1514 bytes captured (12112 bits) on interface 0
Ethernet II, Src: 00:00:00_95:38:ad (00:00:00:95:38:ad), Dst: f8:bf:d2:62:18:c7 (f8:bf:d2:62:18:c7)
Internet Protocol Version 4, Src: 10.0.0.1 (10.0.0.1), Dst: 10.0.0.2 (10.0.0.2)
Transmission Control Protocol, Src Port: 47879 (47879), Dst Port: http (80), Seq: 42918489, Ack: 1,
```

Figure 6.2: A packet captured between end-hosts A-1 and A-2 at the core network

```
Frame 52775: 1514 bytes on wire (12112 bits), 1514 bytes captured (12112 bits) on interface 0
Ethernet II, Src: 00:00:00_7c:ec:5e (00:00:00:7c:ec:5e), Dst: f8:bf:d2:62:18:c7 (f8:bf:d2:62:18:c7)
Internet Protocol Version 4, Src: 10.0.0.1 (10.0.0.1), Dst: 10.0.0.2 (10.0.0.2)
Transmission Control Protocol, Src Port: 47880 (47880), Dst Port: http (80), Seq: 5308065, Ack: 1,
```

Figure 6.3: A packet captured between end-hosts B-1 and B-2 at the core network

```
Frame 110133: 1514 bytes on wire (12112 bits), 1514 bytes captured (12112 bits) on interface 0
Ethernet II, Src: 00:00:00_98:db:53 (00:00:00:98:db:53), Dst: f8:bf:d2:62:18:c7 (f8:bf:d2:62:18:c7)
Internet Protocol Version 4, Src: 10.0.0.1 (10.0.0.1), Dst: 10.0.0.2 (10.0.0.2)
Transmission Control Protocol, Src Port: 47881 (47881), Dst Port: http (80), Seq: 100819009, Ack: 1
```

Figure 6.4: A packet captured between end-hosts C-1 and C-2 at the core network

6.2.2 Test Scenario 2: Isolation in Inter-Tenant Communications

For this test, we focus on the traffic isolation and the address space isolation for the inter-tenant communications. Figure 6.5 shows our test network. In this network, we have three tenants: tenant A, tenant B and tenant C. Each tenant has one end-host and all end-hosts use the same IP address and MAC address. Tenant A is offering a web services on port 80. Other tenants (B, C) are trying to connect to the advertised service of tenant A. For implementing this test, we use Iperf server on end-host A-1 on port 80 and others can join this advertised service by using the mapped address for this service. The mapped address of this service is provided through the service advertisement process explained in Section 4.5.2. It should be noted that all end-hosts in this test received IP address from our DHCP server and we have added the policy groups and rules for each tenant for allowing the inter-tenant communication between the end-hosts.

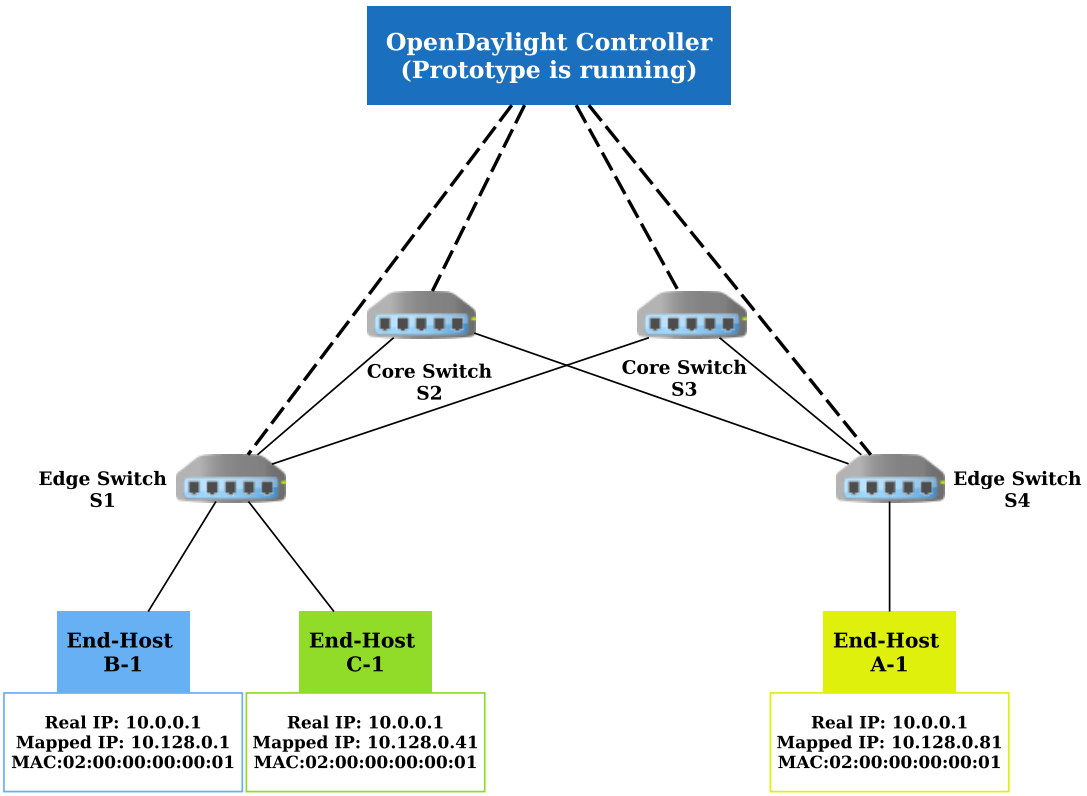

Figure 6.5: Test network for inter-tenant communications

Analysis of Results

We repeated the test for several times and we have observed that both of end-hosts B-1 and C-1 were able to connect with the advertised service offered by end-host A-1. In this test, similar to test scenario 1, the source and destination MAC addresses are rewritten at the ingress edge switch (S1) for providing isolation between different tenant's requests and routing the flows at the core network. Moreover, since it is the inter-tenant communication, we rewrote the source and destination IP addresses and TCP ports at the edge of the network.

Figure 6.6 shows a sample packet captured on the interface between end-host B-1 and edge switch S1. As we can see, at the beginning of the communication, the source MAC address is equal to the real MAC address of end-host B-1 and the destination MAC address of the packet is equal to the MAC address of the SDN controller. The destination IP address and TCP port are equal to the mapped address in end-host A-1 (advertised service).

```
Frame 12022: 1514 bytes on wire (12112 bits), 1514 bytes captured (12112 bits) on interface 3
Ethernet II, Src: 02:00:00:00:00:01 (02:00:00:00:00:01), Dst: 0a:00:27:00:00:00 (0a:00:27:00:00:00)
Internet Protocol Version 4, Src: 10.0.0.1 (10.0.0.1), Dst: 10.128.0.81 (10.128.0.81)
Transmission Control Protocol, Src Port: 48106 (48106), Dst Port: 1024 (1024), Seq: 3321457134, Ack
```

Figure 6.6: A captured packet on the link between end-host B-1 and edge switch S1

Figure 6.7 illustrates a captured packet on the link between the edge switch S1 and the core switch S3. As we can see, the source MAC address is changed to

the domain label for tenant B and the destination MAC address is changed to the routing label for the path between two edge switches (the path between S1 and S4). Moreover, the source IP address is changed to a free inter-tenant IP address and port number for end-host B-1 and the destination IP address and port numbers are changed to the real IP address and TCP port number for end-host A-1.

```
Frame 12019: 1514 bytes on wire (12112 bits), 1514 bytes captured (12112 bits) on interface 1
Ethernet II, Src: 00:00:00_44:de:35 (00:00:00:44:de:35), Dst: 93:85:75:50:c4:38 (93:85:75:50:c4:38)
Internet Protocol Version 4, Src: 10.128.0.41 (10.128.0.41), Dst: 10.0.0.1 (10.0.0.1)
Transmission Control Protocol, Src Port: 1024 (1024), Dst Port: http (80), Seq: 3321457134, Ack: 13
```

Figure 6.7: A captured packet on the link between edge switch S1 and core switch S3

Figure 6.8 depicts a captured packet on the link between edge switch S3 and end-host A-1. According to the figure, the destination MAC address is changed to the destination MAC address of end-host A-1 for accepting the packet.

```
Frame 12020: 1514 bytes on wire (12112 bits), 1514 bytes captured (12112 bits) on interface 4
Ethernet II, Src: 00:00:00_44:de:35 (00:00:00:44:de:35), Dst: 02:00:00:00:00:01 (02:00:00:00:00:01)
Internet Protocol Version 4, Src: 10.128.0.41 (10.128.0.41), Dst: 10.0.0.1 (10.0.0.1)
Transmission Control Protocol, Src Port: 1024 (1024), Dst Port: http (80), Seq: 3321457134, Ack: 13
```

Figure 6.8: A captured packet on the link between edge switch S3 and end-host A-1

Finally, if we check the output of Iperf server on host A-1 in Figure 6.9, we can see that both end-hosts B-1 and C-1 successfully made a TCP connection with the server.

```
root@mininet-vm:~# iperf -s -p 80
------------------------------------------------------------
Server listening on TCP port 80
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  4] local 10.0.0.1 port 80 connected with 10.128.0.41 port 1024
[  5] local 10.0.0.1 port 80 connected with 10.128.0.1 port 1024
```

Figure 6.9: The output of Iperf server on the end-host A-1

6.2.3 Test Scenario 3: Isolation in External Communications

The purpose of this scenario is to test the traffic and address space isolation for the external communications by tenants. For implementing this test, consider that we are working in the network depicted at Figure 6.10. One of the advantages of our approach in the external communications is that we do not need any NAT middlebox since we do mapping at the ingress edge switch. Hence, we should remove the NAT feature from our network for connecting to the Internet. Indeed, we use the public IP addresses for the external communications and a free public IP address is allocated to each tenant. For allowing this IP addresses to be routed through the Internet, we should simulate the functionality of the real gateways without the NAT feature. Since our tests are running in a virtual environment by using Mininet on Ubuntu, we have disabled the NAT feature on Ubuntu and we made two network

interfaces with public IP addresses[1] and we have used the OpenvSwitch bridges and the patch ports [14] for connecting Mininet to the real network interfaces.

In this network, we have tenant A and tenant B. Each tenant has one end-host with the same IP address and MAC address. The end-hosts try to make TCP connections with the public server 91.144.184.232 on port 5001 using Iperf. The public IP address of tenant A for the external communications is 84.249.205.146 and the public IP address of tenant B for the external communications is 80.220.225.185.

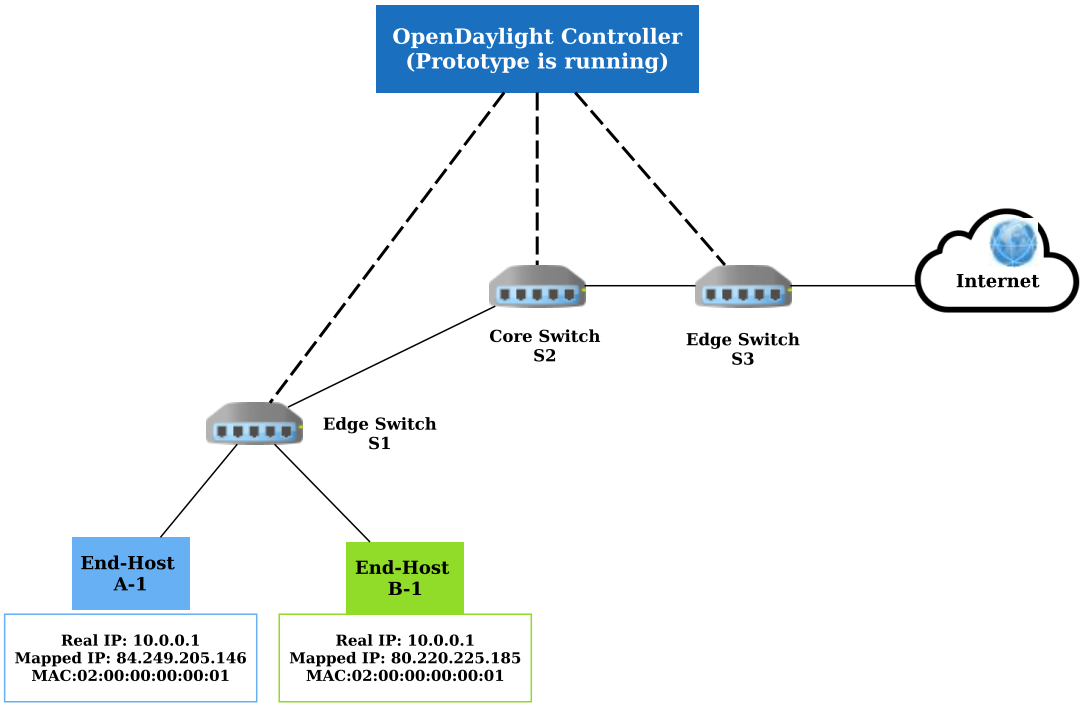

Figure 6.10: Test network for external communications

Analysis of Results

During this test, both end-hosts were able to connect to the public Iperf server. Similar to the intra-tenant and inter-tenant connections, we rewrite the packet headers at the edge of the network.

For sending the request from the end-hosts through the Internet, the source MAC address is changed to the domain label and the destination MAC address is changed to the routing label. Moreover, the source IP address and source port are also changed to the allocated public IP address and available port number. Figure 6.11 and Figure 6.12 show sample packets captured on the link between edge switch S1 and core switch S2. End-host A-1 and B-1 sent these packets to the public Iperf server. As we can see, the IP addresses are mapped to the public IP addresses and port numbers and the source MAC address is equal to the domain label and the destination MAC address is equal to the routing label.

[1]Our service provider is Sonera and it offers public IP addresses for end-users

```
Frame 36796: 1514 bytes on wire (12112 bits), 1514 bytes captured (12112 bits) on interface 2
Ethernet II, Src: 00:00:00_bc:73:8a (00:00:00:bc:73:8a), Dst: f5:4f:f6:ca:17:1d (f5:4f:f6:ca:17:1d)
Internet Protocol Version 4, Src: 84.249.205.146 (84.249.205.146), Dst: 91.144.184.232 (91.144.184.232)
Transmission Control Protocol, Src Port: netinfo-local (1033), Dst Port: 5001 (5001), Seq: 4154958639,
```

Figure 6.11: A packet captured on the link between switch S1 and S2. The source of the packet is end-host A-1

```
Frame 36825: 1514 bytes on wire (12112 bits), 1514 bytes captured (12112 bits) on interface 2
Ethernet II, Src: 00:00:00_d3:ad:ae (00:00:00:d3:ad:ae), Dst: f5:4f:f6:ca:17:1d (f5:4f:f6:ca:17:1d)
Internet Protocol Version 4, Src: 80.220.225.185 (80.220.225.185), Dst: 91.144.184.232 (91.144.184.232)
Transmission Control Protocol, Src Port: 1028 (1028), Dst Port: 5001 (5001), Seq: 2638465028, Ack: 3042
```

Figure 6.12: A packet captured on the link between switch S1 and S2. The source of the packet is end-host B-1

Figure 6.13 and Figure 6.14 show the responses to the requests sent from the end-hosts. These packets are captured on the link between core switch S2 and edge switch S1. As we can see the source IP address is correctly changed to the real IP address of the end-hosts (10.0.0.1).

```
Frame 36800: 66 bytes on wire (528 bits), 66 bytes captured (528 bits) on interface 1
Ethernet II, Src: 00:00:00_bc:73:8a (00:00:00:bc:73:8a), Dst: 32:5d:28:d0:a8:b7 (32:5d:28:d0:a8:b7)
Internet Protocol Version 4, Src: 91.144.184.232 (91.144.184.232), Dst: 10.0.0.1 (10.0.0.1)
Transmission Control Protocol, Src Port: 5001 (5001), Dst Port: 35491 (35491), Seq: 2046044649, Ack:
```

Figure 6.13: A packet captured on the link between the core switch S2 and the edge switch S1. The destination of the packet is end-host A-1

```
Frame 36810: 66 bytes on wire (528 bits), 66 bytes captured (528 bits) on interface 1
Ethernet II, Src: 00:00:00_d3:ad:ae (00:00:00:d3:ad:ae), Dst: 32:5d:28:d0:a8:b7 (32:5d:28:d0:a8:b7)
Internet Protocol Version 4, Src: 91.144.184.232 (91.144.184.232), Dst: 10.0.0.1 (10.0.0.1)
Transmission Control Protocol, Src Port: 5001 (5001), Dst Port: 35490 (35490), Seq: 3042488671, Ack:
```

Figure 6.14: A packet captured on the link between the core switch S2 and the edge switch S1. The destination of the packet is end-host B-1

6.2.4 Test Scenario 4: Performance Isolation

The performance isolation in our architecture is implemented based on the monitoring by sFlow-RT. The use of monitoring in our architecture gives a possibility to track the traffic of different tenants at the data plane. The monitoring module is based on the definition of a threshold for the traffic originated at the edge of the network. If each tenant goes beyond the predefined threshold, our monitoring module reacts to the detected violation. We run a test in the test network depicted in Figure 6.1. In this network, we assume that we have three tenants A, B and C. Each tenant has two end-hosts. End-hosts A-1 and A-2 are for tenant A, end-hosts B-1 and B-2 are for tenant B and end-hosts C-1 and C-2 are for tenant C. We use Iperf to make a TCP connection from end-hosts A-1, B-1 and C-1 to A-2, B-2 and C-2, respectively. At first, we run this test without any monitoring over tenant's traffic and then we run the same test but by activating our monitoring module. The monitoring module is configured to detect flows that are transmitted at a rate over

2 Mbps. As we explained in Section 5.8, the reaction in our monitoring module can be *warning* or *blocking*. For this test we have configured our monitoring module to block all detect violations.

Analysis of Results

After running the first test, all tenants were competing to send higher ranges of traffic through the network. Figure 6.15 shows the range of traffic transmitted by different tenants. We can see that without any monitoring, tenants were able to increase the transmission range to more than 50 Mbps.

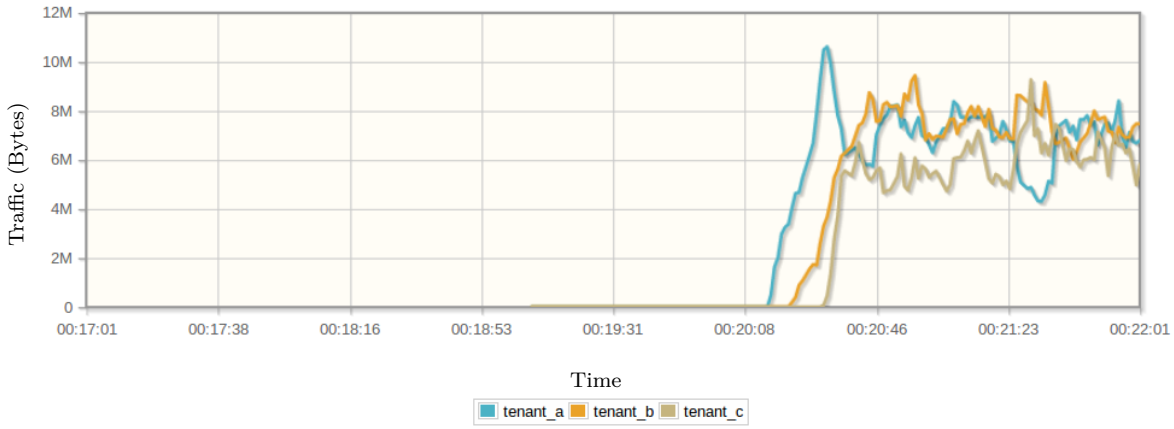

Figure 6.15: Testing the prototype without the monitoring module

After implementing the next test, our monitoring module could successfully detect large flows in the network. Figure 6.16 shows the output of sFlow-RT after the detection of unusual traffic for different tenants. The unusual traffic from tenants was detected by sFlow-RT and the monitor manager on the OpenDaylight controller blocked all detected violations.

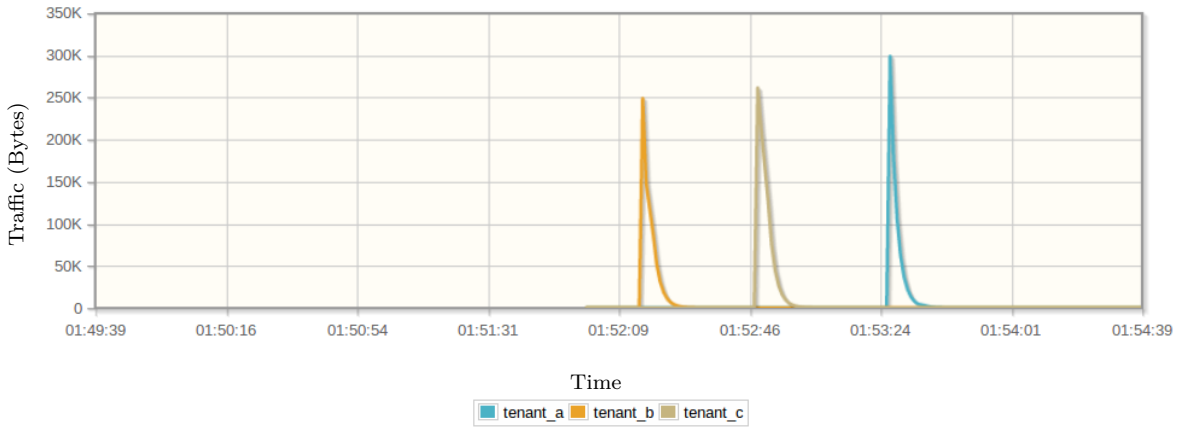

Figure 6.16: Testing the prototype by enabling the monitoring module

6.3 Scalability

In this section, we evaluate the scalability of our solution for implementation at the core network and for assigning IP addresses. Since current hardware switches at the core network are not able to hold the large number of forwarding rules, our solution should provide a satisfactory level of scalability at the core network. Moreover, we need to evaluate the scalability of our solution in terms of IP address assignment between tenants in a shared network.

6.3.1 Scalability at the Core Network

As we explained in Section 4.1, while the software switches are able to hold millions of flow entries, today's hardware switches cannot hold the large number of forwarding rules. Because of this reason, we designed our system architecture based on the use of software switches at the edge of the network and the hardware switches at the core of the network. For increasing the scalability on hardware switches, we used the routing labels to aggregate several flows and route them in the same path.

In this section, we want to evaluate our solution in terms of the scalability on the number of forwarding rules at the core network. We evaluate the scalability of our solution based on a sample network depicted in Figure 6.17. In this network, all end-hosts are trying to send a flow request to the server. This server might be available locally or through the external network[2]. Consider that we have 100 ingress edge switches connected to the end-hosts in the network of Figure 6.17 and each ingress edge switch has 500 end-hosts.

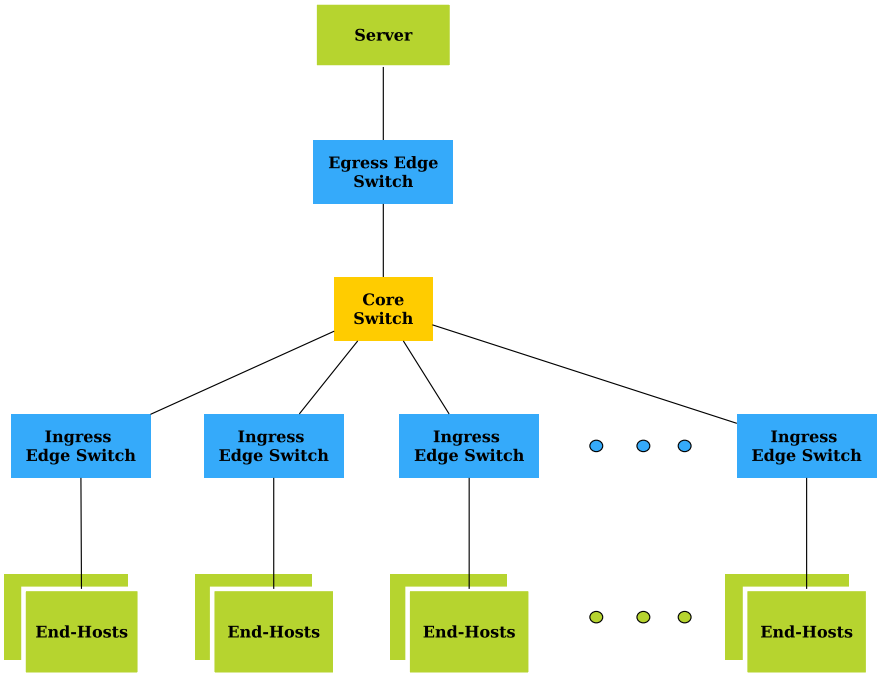

Figure 6.17: A sample network for testing the scalability

[2]For making it clearer for our analysis, we chose one core switch. However, it is extendable to more number of switches at the core network

Figure 6.18 shows the number of forwarding rules at the ingress edge switch and the core switch based on the number of flows initiated from each end-host. At the edge of the network, the number of forwarding rules is increasing based on the number of flows from end-hosts. However, at the core network, since we aggregate several flows, the number of forwarding rules is constant and it is based on the number of paths between the edge switches. Indeed, while the number of entries at the edge switches increases based on the number of flows, at the core network, we increases the scalability by aggregation.

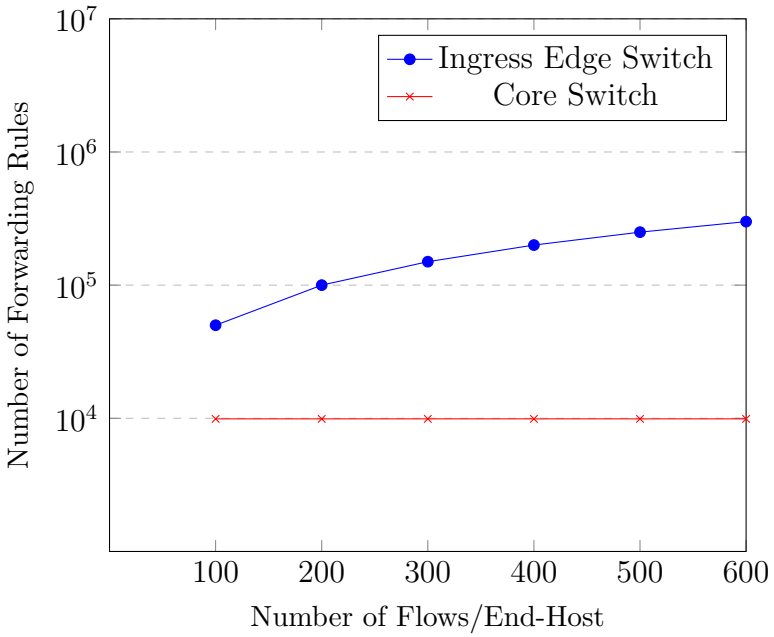

Figure 6.18: The number of flow entries at the ingress edge switch and the core switch

6.3.2 Scalability in IP Address Assignment

In this section, we explain about the scalability of the IP address assignment in our solution. Considering the private address space of 10.0.0.0/8 and 172.16.0.0/12, we have allocated these ranges of IP address for the intra-tenant and inter-tenant communications between tenants. For the intra-tenant communications, we have considered the subnet 10.0.0.0/9. Each tenant in the network can use this range of IP address in any range of transport ports for their intra-tenant communications. In fact, the overlapped IP addresses for the intra-tenant communications are allowed in our architecture. For the inter-tenant communications, since the IP addresses between two different tenants should be unique, we choose an IP address from the range of 10.128.0.0/9 and 172.16.0.0/12. Moreover, we have considered the transport port range of 1024-65535 for each inter-tenant IP address. For the external communications, we relied on the available external IP addresses. In this version of our architecture, we have not considered any special mechanism for reducing the number of public IP addresses for the external communications and we assumed that there is at least one public IP address available for each tenant. Similar to the inter-tenant addressing, for the external communications, the transport ports

should be in a range of 1024-65535. Table 6.1 shows the number of assigned IP addresses and transport port numbers for tenants based on the type of communication.

Table 6.1: The number of available IP addresses and port numbers for different types of communications

Type of Communication	**Allocated Subnet(s)**	**Number of IP Addresses**	**Transport Ports Range**
Intra-Tenant	10.0.0.0/9	2^{23} / per tenant	0-65535
Inter-Tenant	10.128.0.0/9 & 172.16.0.0/12	2^{23} / total number of tenants	1024-65535
External	Available public IP addresses	≥ 1 / per tenant	1024-65535

6.4 Control Traffic Overhead

In this section, we aim to evaluate the control traffic overhead that our approach puts on the network for processing the new flows. At first, we calculate the size of different messages transferred between the controller and the switches and then we calculate the overall overhead in our prototype[3].

6.4.1 Size of OpenFlow Messages

- **Header Field:**
  All communications between the controller and the switch is transferred using TCP/IP protocol and through Ethernet frames. Table 6.2 shows the size of header fields for messages transmitted between the controller and the switches. For our analysis, we assumed the size of Ethernet header is 18 bytes, IP header is about 20 bytes, TCP header is about 20 bytes and the OpenFlow header is 8 bytes.

[3]It should be noted that in this section we have calculated the control traffic for communications between the source and the destination end-hosts that are connected to different edge switches since it gives us a better estimation on the overall amount of the control traffic in the network.

Table 6.2: Size of header fields for all transmissions between the controller and the switches

Protocol	Header (B)
Ethernet Header	18
IP header	20
TCP header	20
OpenFlow header	8
Total header overhead	66

- **Packet-In:**
  The packet-in message is initiated upon receiving a new flow request. Each packet-in message includes a Buffer ID (4 bytes), Frame Total Length (2 bytes), Frame Receive Port (2 bytes), Reason (1 byte), Pad (1 byte) and Data Field. The size of Data field depends on if the edge switch supports buffering or not. For our analysis, we assumed that the edge switch does not support buffering and the received packet from the end-host is a HTTP GET message and approximately, the size of a simple HTTP GET message is about 200 bytes. Table 6.3 shows the size of packet-in message for a simple HTTP GET request.

Table 6.3: Size of Packet-in message

Message	Header (B)	Payload (B)	Frame (B)
Packet-in	66	10 + 200	276

- **Packet-Out:**
  The packet-out message is used for sending a message from the controller to the switch at the data plane. For measuring the traffic overhead, we assume that the switches at the data plane do not support buffering. In this case, the original packet that has been sent to the controller will be returned to the switch. The packet-out includes a Buffer ID (4 bytes), input port (2 bytes), actions length (2 bytes), Output port action (8 bytes), Data. The size of Data field is equal to the size of data field in the packet-in messages so we choose 200 bytes. Table 6.4 shows the size of packet-out message for a simple HTTP Get request.

Table 6.4: Size of Packet-out message

Message	Header (B)	Payload (B)	Frame (B)
Packet-out	66	16 + 200	282

- **Flow-Mod:**
  This message includes: Match field (40 bytes), Cookie (8 bytes), Command (2 bytes), Idle timeout (2 bytes) , Hard timeout (2 bytes), Priority (2 bytes), Buffer ID (4 bytes), Out port (2 bytes), Flags (2 bytes) and list of Actions. The size of the action field depends on the type of action that has been defined by the SDN controller. If the list of actions is empty, then it means the flow should be dropped. However, if we want to forward the flow depending on the type of communication (intra-tenant, inter-tenant and external), we should add a separate set of actions.

  For the intra-tenant communications, we have actions to change the source and the destination MAC addresses and output the packet from the switch port. The size of these actions are: SetDlSrc (16 bytes), SetDlDst (16 bytes), Output (8 bytes).

  For the inter-tenant communications, we change the source and the destination MAC addresses, IP addresses and ports. Moreover, we add an output action to send the flow out of a specific port. The size of these actions is: SetDlSrc (16 bytes), SetDlDst (16 bytes), SetNwSrc (8 bytes), SetNwDst (8 bytes), SetTpSrc (8 bytes), SetTpDst (8 bytes), Output (8 bytes).

  For the external communications, we change the source IP address, source port and the source and the destination MAC addresses. Finally, we send it out of a specific port. The total size is based on the following actions: SetNwSrc (8 bytes), SetTpSrc (8 bytes), SetDlSrc (16 bytes), SetDlDst (16 bytes), Output (8 bytes).

  Additionally, for all of the intra-tenant, inter-tenant or external communication, we install a flow-mod message on the egress edge switch to change the destination MAC address. The size of the action to change the destination MAC address is: SetDlDst (16 bytes).

  Considering the Ethernet, TCP, IP and OpenFlow headers, the total size of Flow-Mod messages based on the type of communication is depicted in Table 6.5.

Table 6.5: Size of Flow-Mod message for different types of flows

Messages	Header (B)	Payload (B)	Frame (B)
Flow-Mod (Drop)	66	64	130
Flow-Mod (Intra-Tenant) Ingress Switch	66	64 + 40	170
Flow-Mod (Inter-Tenant) Ingress Switch	66	64 + 72	202
Flow-Mod (External) Ingress Switch	66	64 + 56	186
Flow-Mod Egress Switch	66	64 + 16	146

- **Barrier**
  Barrier messages are transferred after sending the Flow-mod messages. These messages do not have any payload so the size of barrier messages (request or reply) are equal to the size of the header fields. Table 6.6 shows the size of barrier request and reply messages.

Table 6.6: Size of Barrier messages

Messages	Header (B)	Payload (B)	Frame (B)
Barrier Request	66	0	66
Barrier Reply	66	0	66

- **Flow-Removed**
  The Flow-Removed message is forwarded to the controller if an installed forwarding rule on the switch is expired (timed-out). With this message, the switch informs the controller about the removed flows. This message includes: Match field (40 bytes), Cookie (8 bytes), Priority (2 bytes), Buffer ID (4 bytes), Reason (1 byte), Flow duration seconds (2 bytes), Flow duration nano seconds (4 bytes), Idle time before discarding (2 bytes), Packet count (8 bytes), Byte count (8 bytes), Pad (1 byte). Considering the Ethernet, TCP and IP headers, the size of Flow-Removed messages is about 80 bytes. Table 6.7 shows the size of Flow-Removed message.

Table 6.7: Size of Flow-Removed message

Message	Header (B)	Payload (B)	Frame (B)
Flow-Removed	66	80	146

6.4.2 Control Overhead in the Prototype

As we discussed in the last section, for each flow request, a packet-in message is sent to the controller. Subsequently, depending on the isolation process, we may drop the packet or forward it. In case of forwarding, depending on the type of connection (intra-tenant, inter-tenant or external), two flow-mod messages are installed on the ingress edge switch and the egress edge switch. Moreover, we send back the received packet to the switch (packet-out). Figure 6.19 shows the type of messages transferred between the controller and the edge switches.

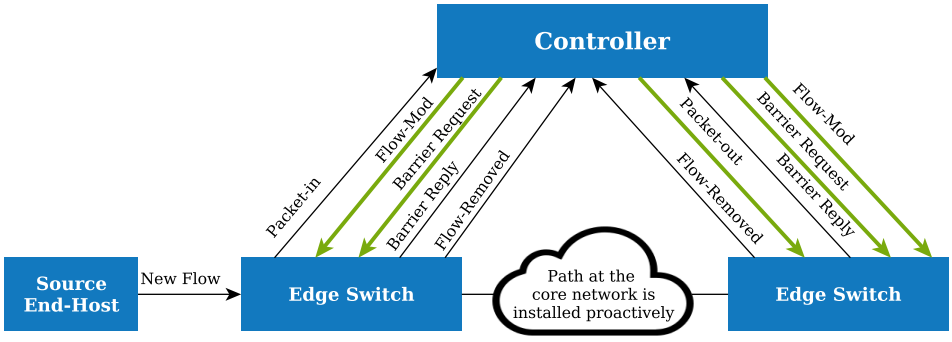

Figure 6.19: Control messages (OpenFlow) for processing a new flow request in our prototype

According to Figure 6.19, we can calculate the control overhead for processing a new flow. The estimated size of different messages in our architecture, based on the type of flow request, is illustrated in Table 6.8.

Table 6.8: Total control overhead for handling a new flow

Messages	Drop	Intra-Tenant	Inter-Tenant	External
Packet-in (B)	276	276	276	276
Packet-out (B)	0	282	282	282
Flow-Mod (B) Ingress Switch	130	170	202	186
Flow-Mod (B) Egress Switch	0	146	146	146
Barrier Request (B)	66	66 + 66	66 + 66	66 + 66
Barrier Reply (B)	66	66 + 66	66 + 66	66 + 66
Flow-Removed (B)	146	146 +146	146 +146	146 +146
Total Overhead (B) /Flow	**684**	**1430**	**1462**	**1446**

Based on the results in Table 6.8, we are able to show the relation between the amount of control overhead and the number of flows in the network. Figure 6.20 shows the amount of control traffic generated based on the number new flow requests.

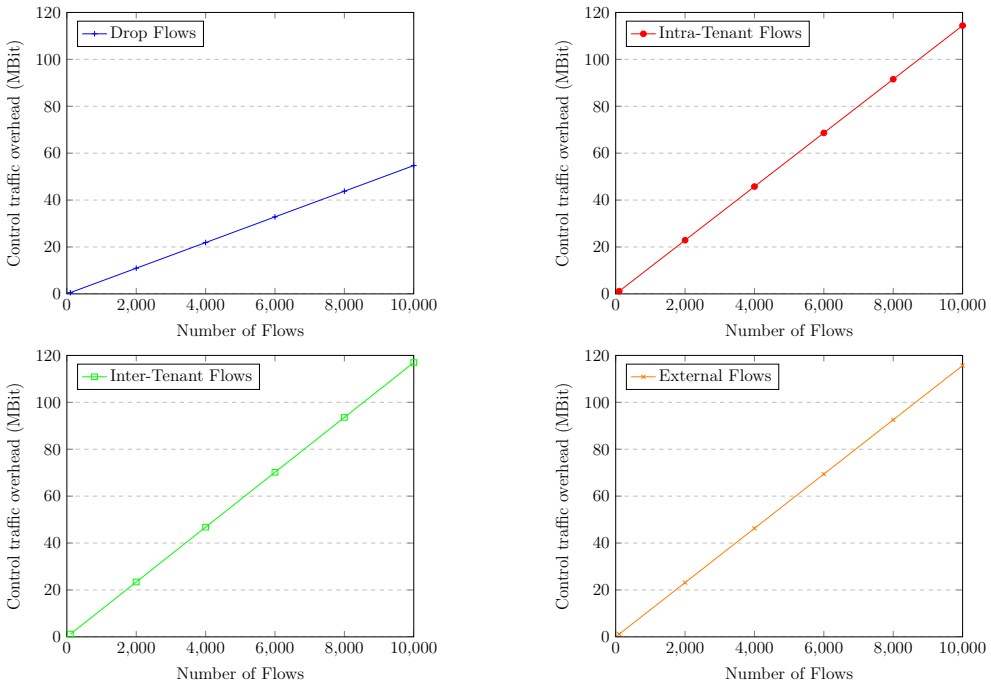

Figure 6.20: The amount of control traffic based on the number of new flow requests (a) dropped, (b) intra-tenant, (c) inter-tenant and (d) external communications

Based on the results from Figure 6.20, we can see that the highest amount of traffic is generated for the inter-tenant communications. For the inter-tenant communications, the amount of the control traffic for processing 10000 flows is about 116 MBit. Today's Ethernet links can handle this amount of traffic in the network. However, it is possible to decrease this amount of overhead by removing the extra messages such as barrier messages and flow-removed messages. These messages are not used in our system architecture and they add extra overhead to our network. Moreover, the usage of switch buffering removes the packet-out messages and decreases the size of packet-in messages (only header is transmitted to the controller).

6.5 Latency of Rule Matching Process

In our solution, tenants are able to define rules for their traffic. These rules are created by tenants to allow special types of traffic in the network. In this section, we want to measure the latency of our solution based on the number of rules. For this purpose, we run the controller on a separate machine with 64 bit Ubuntu 14.04, Core i7 CPU M 640 @ 2.80 GHz x 4 and 8 GB RAM. For testing the latency, we send the packet-in requests to the controller and measure the time difference between sending a packet-in message to the controller and receiving back a flow-mod message. The

isolation rules are placed in one policy group and we have assigned random values to each rule. During this test, we ensured that only one rule will match the new flow. We repeated this test for 20 times. Figure 6.21 shows our results. This diagram shows the mean values and the standard deviation values for the latency based on the number of rules.

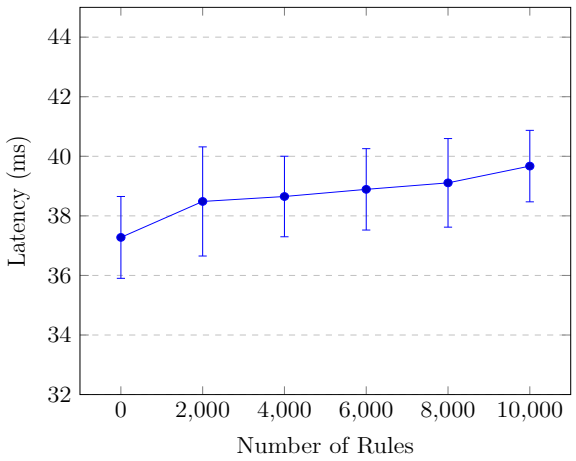

Figure 6.21: The latency of rule matching technique for installing forwarding rules

The result of our test shows that the latency in our prototype is quite stable for the large number of rules. The mean value for 10000 rules is less than 39 ms and the variation in latency for the different number of rules is quite small. The reason why the latency does not change considerably is because of our rule matching algorithm discussed in Section 5.6.1. Our rule matching algorithm is based on the *divide and conquer* algorithm. In fact, for every new flow request, we break the flow into five attributes (source IP address, destination IP address, source port, destination port and protocol) and we search for each of these attributes separately. The searching operation for finding the correct rule is based on the use of HashMap and Interval Tree. The time complexity for searching in HashMap is $O(1)$ [9] and the time complexity for searching in Interval Tree is $O(logn + L)$ (L is a constant value) [55]. As a result, the time complexity of our rule checking algorithm is quite small and the latency does not change considerably for the increasing number of rules. We believe that the amount of latency can be decreased by the code optimization in our prototype. However, the code optimization was not part of the work in this thesis and in future, we will optimize the code for achieving a lower latency.

6.6 Discussion

In this thesis, after an extensive research about the isolation requirements and techniques, we proposed a new system architecture which improves the isolation in a multi-tenant SDN. For providing isolation, our approach is based on the packet rewriting at the edge of the network. This approach has several advantages in comparison with the other techniques like encapsulation. The first advantage of this approach is to provide isolation between tenant's traffic and to allow tenants to define overlapped addresses. Moreover, by packet rewriting we are able to embed

the routing labels in the packet headers and route the flows based on the embedded routing labels. Our results proved that the usage of routing labels leads to a satisfactory level of scalability at the core network. The other advantage of the packet rewriting is that it makes it possible for tenants to make the intra-tenant, inter-tenant and external communications. In fact, while the packet rewriting approach provides isolation in our solution, it increases the interoperability between tenants and the external resources.

Furthermore, our solution provides a possibility for tenants to make their own configurations. Indeed, in our system, tenants are able to decide about their networks by setting subnets and implementing the policy groups and rules. This feature leads to higher functionality and simplicity compared with the other available solutions, where the administrator of the network should decide about the configurations for each tenant [40].

Additionally, in our solution, we have considered the concept of performance isolation between tenants. While it is possible to use the statistics received by the OpenFlow protocol for monitoring, the recent investigation [61] proves that the monitoring based on the OpenFlow protocol is not a scalable solution. For this reason, the performance isolation in our solution is implemented using the sFlow protocol which is based on the sampling technique and it is highly accepted as a low overhead and a scalable monitoring approach. Additionally, the use of sFlow-RT as the analyzer in our architecture leads to the real-time detection of violations between tenants.

For making a connection with the Internet using our solution, the edge switches are used which are connected to the gateway. In fact, we have used the edge switches to make our approach more uniform at the edge of the network. However, in the real world deployments, the gateway and the edge switch might be integrated with each other. Current OpenFlow-enabled gateways [4] are based on software solutions to support millions of flows and they can do routing tasks. This kind of gateways are the suitable choice for connecting our solution in a multi-tenant SDN network to the Internet.

While our approach brings several advantages, there are other aspects that need to be investigated in more detail. Our solution is based on rewriting the packet headers at the edge of the network. The packet rewriting is a part of today's communications. For instance, for communications using NAT, the IP address and port number are rewritten. However, we need an accurate measurement on the effect of header rewriting on the throughput of our communications.

6.7 Chapter Summary

This chapter presents the results of the evaluation of the prototype. In this regard, we have evaluated the prototype in terms of the effectiveness of the prototype in providing isolation in a multi-tenant SDN and then several test scenarios were implemented to test the functionality of the prototype in real deployments. At the next part of this chapter, the prototype is evaluated based on the scalability at the core network and the scalability in IP address assignment. Additionally, the control traffic overhead for the prototype and the latency in the rule matching process are

evaluated in this chapter. Finally, different aspects of our solution are discussed in this chapter.

Chapter 7

Conclusion

We started this thesis to explore the possibility of providing isolation in a multi-tenant SDN. At first, we started to explain about the SDN architecture and particularly OpenFlow which is used as a southbound protocol in the SDN architecture. We discussed on how the SDN controller is able to use the OpenFlow protocol to proactively or reactively install the forwarding rules on the network elements in the data plane.

After giving an introduction to SDN, we explained about the concept of multi-tenancy in SDN and how it can be used for increasing the efficiency in management and decreasing the operational costs in today's networks. The multi-tenancy in SDN enables tenants to interact with the SDN provider using their own applications or controllers. Since in a multi-tenant SDN the resources are shared between tenants, we continued our research to explore the possibility of providing isolation in a multi-tenant SDN. Then, we presented the existing isolation techniques in a multi-tenant SDN network. The existing solutions provide isolation by slicing, encapsulation and packet rewriting methods. The slicing technique provides isolation by slicing the resources and assigning each slice to a tenant while the encapsulation technique adds new labels to the flows for isolating tenants. Additionally, the packet rewriting approach is used for providing isolation which allows to change and embed new information in the packet headers.

We continued our work by proposing a solution which improves the isolation between tenants in a shared network. In this regard, we proposed a new solution that separates the edge and core network. While at the edge of the network we provide isolation, at the core network, we concentrate on the routing and forwarding tasks. Our solution focuses on four main isolation requirements including the traffic isolation, the address space isolation, the control isolation and the performance isolation. In our solution, the traffic isolation guarantees that all flows are verified at the edge of the network and there will not be any information leakage between tenants. The address space isolation provides a possibility for tenants to use the overlapped addresses for their communications. By the control isolation, we have provided a possibility for each tenant to control their own network without affecting the configurations made by other tenants and finally, the performance isolation makes it possible to find violations from tenants by monitoring the traffic at the data plane. In the proposed system, tenants are able to make the intra-tenant, inter-tenant and external communications. With the intra-tenant communications,

tenants are able to limit the scope of their communication to their own network while in the inter-tenant communications, tenants are able to join the special service offered by the other tenants or advertise a specific service to the others in a shared network. Moreover, with the external communications, tenants are able to make interaction with the resources outside of the shared network. As a part of our work, we have implemented our approach using Java bundles on the OpenDaylight controller

At the last part of this work, we evaluated our solution in terms of isolation and we tested our prototype in different scenarios for testing the isolation in the intra-tenant, the inter-tenant and the external communications. Furthermore, we evaluated our solution in terms of scalability, overhead and latency. The results proved that our solution is scalable enough to be implemented in large-scale networks. While the proposed system in this thesis needs more improvements in terms of latency and control overhead, we believe that we have fulfilled our design goals.

At the following, we introduce the possible future works for our solution:

- **Integration with Cloud Computing:**
  The concept of multi-tenancy in cloud networks has been widely studied [62, 63]. The combination of SDN with the cloud computing leads to a solution for deployments in the multi-tenant data centers. OpenStack [22] is a cloud computing software for creating private and public clouds and the OpenDaylight controller has a driver for connecting to OpenStack. We have a plan to integrate our approach which is implemented on the OpenDaylight controller with OpenStack for providing an isolation system for the deployments in multi-tenant data centers.

- **Extending the work to a Multi-Location Solution:**
  Our solution in this work is limited to one network which is shared by several tenants. However, today's networks and data centers are geographically distributed [57]. In fact, tenants may have several end-hosts on different networks which are located in different places. Our future work is to extend our solution to be implemented in distributed networks.

  As we discussed during this work, we have considered a domain for each tenant. We can extend this feature to *distributed domains* across several networks. Figure 7.1 demonstrates an example of this approach. According to this figure, end-hosts in the same or different domains might be distributed across two different networks. We have a plan to make it possible that end-hosts from the same or different domains which are placed in different networks communicate with each other.

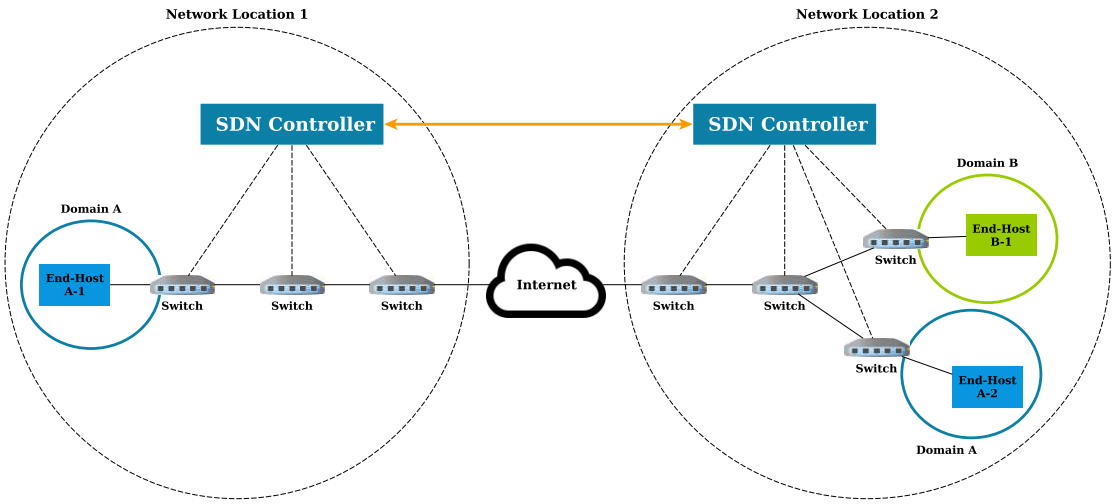

Figure 7.1: Multi-location approach for future work

- **Supporting Clustering Approach:**
  Currently, our solution has been implemented on one controller. However, for increasing the responsiveness, reliability and scalability, it is desirable to implement this solution to work in the clustering mode on distributed systems [64]. The OpenDaylight controller supports this approach and we plan to extend our solution to be implemented in the clustering mode [17].

Bibliography

[1] Software-defined Networking: The New Norm for Networks, Open Networking Foundation, 2012. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf, Last access [1.11.2014]

[2] OpenFlow Switch Specification, Version 1.0.0, Open Networking Foundation, 2009. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.0.0.pdf, Last access [1.11.2014]

[3] SDN Architecture, Issue 1, Open Networking Foundation, 2014. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/TR_SDN_ARCH_1.0_06062014.pdf, Last access [30.10.2014]

[4] SDN Gateway Reference Design, Available: https://netronome.com/wp-content/uploads/2014/10/SDN-Gateway-Solution-Brief-10-14.pdf, Last access [27.2.2015]

[5] sFlow Blog, Available: http://blog.sflow.com/2013/01/performance-aware-software-defined.html, Last access [27.2.2015]

[6] sFlow-RT, Available: http://www.inmon.com/products/sFlow-RT.php, Last access [27.2.2015]

[7] Node.js, Available: https://nodejs.org, Last access [27.2.2015]

[8] Open vSwitch, Available: http://openvswitch.org, Last access [22.2.2015]

[9] Big-O Cheat Sheet, Available: http://bigocheatsheet.com, Last access [22.2.2015]

[10] Linux Containers. Available: https://linuxcontainers.org, Last access [28.2.2015]

[11] IPerf, Available: https://iperf.fr, Last access [10.2.2015]

[12] MPLS FAQ For Beginners, Available: http://www.cisco.com/c/en/us/support/docs/multiprotocol-label-switching-mpls/mpls/4649-mpls-faq-4649.html#qa2, Last access [2.2.2015]

[13] An Instant Virtual Network on your Laptop, Available: http://mininet.org/, Last access [22.2.2015]

[14] Connecting OVS bridges with Patch Ports, Available: http://blog.scottlowe.org/2012/11/27/connecting-ovs-bridges-with-patch-ports, Last access [1.3.2015]

[15] OpenDaylight Jenkins Repository, Available: https://jenkins.opendaylight.org, Last access [22.11.2014]

[16] OpenDaylight - An Open Source Community And Meritocracy For Software-Defined Networking, A Linux Foundation, 2013. Available: http://www.opendaylight.org/resources/publications, Last access [20.10.2014]

[17] OpenDaylight Wiki, Available: https://wiki.opendaylight.org/view/Main_Page, Last access [28.2.2015]

[18] VTN Documentation, Available: https://wiki.opendaylight.org/view/OpenDaylight_Virtual_Tenant_Network_(VTN):Main, Last access [25.11.2014]

[19] OpenVirtex Documentation, Available: http://ovx.onlab.us/documentation/, Last Access [3.11.2014]

[20] Roy Thomas Fielding, "Architectural Styles and the Design of Network-based Software Architectures," Dissertation, University of California, Irvine, 2000. Available: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm, Last access [28.2.2015]

[21] An Introduction To VLAN Trunking, available at: http://www.formortals.com/an-introduction-to-vlan-trunking, Last accessed: [3.11.2014]

[22] OpenStack Wiki page, https://wiki.openstack.org/wiki/Main_Page, Last accessed: [10.2.2015]

[23] Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks, RFC 7348, Available: http://tools.ietf.org/html/rfc7348, Last access [2.11.2014]

[24] Dynamic Host Configuration Protocol, RFC 2131, Available: https://www.ietf.org/rfc/rfc2131.txt, Last access [23.12.2014]

[25] The Transport Layer Security (TLS) Protocol, Version 1.2, RFC 5246, Available: https://tools.ietf.org/html/rfc5246, Last access [11.10.2014]

[26] Encapsulation Methods for Transport of Layer 2 Frames over MPLS Networks, RFC 4950, Available: https://tools.ietf.org/html/rfc4905, Last access [25.10.2014]

[27] InMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks, RFC 3176, Available: https://www.ietf.org/rfc/rfc3176.txt, Last access [26.10.2014]

[28] OSGi Architecture, Available: `http://www.osgi.org/Technology/WhatIsOSGi`, Last access [25.2.2015]

[29] Apache Maven, Available: `http://maven.apache.org`, Last access [20.2.2015]

[30] M. Berg, M. Kreveld, M. Overmars, and O. Schwarzkopf, "Computational Geometry: Algorithms and Applications", Springer-Verlag, Second Revised Edition, 2000.

[31] P. Goransson, and C. Black, "Software Defined Networks: A Comprehensive Approach", Elsevier, 2014.

[32] K. Benton , L. J. Camp, and C. Small, "OpenFlow vulnerability assessment," In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, August 16-16, 2013.

[33] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, and S. Shenker, "Extending Networking Into the Virtualization Layer", *8th ACM Workshop on Hot Topics in Networks (HotNets-VIII)*, 2009.

[34] M. Kobayashi, S. Seetharaman, G. Parulkar, G. Appenzeller, J. Little, J. Reijendam, P. Weissmann, and N. McKeown, "Maturing of OpenFlow and Software Defined Networking through Deployments", *Computer Networks, Elsevier Journal*, v. 62, pp. 151-175, 2014.

[35] C. P. Bezemer , A. Zaidman , B. Platzbeecker, and T. Hurkmans , A. Hart, "Enabling multi-tenancy: An industrial experience report," In *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, 2010, pp.1-8.

[36] S. Walraven , T. Monheim , E. Truyen, and W. Joosen, "Towards performance isolation in multi-tenant SaaS applications," In *Proceedings of the 7th Workshop on Middleware for Next Generation Internet Computing*, 2012, pp.1-6.

[37] D. Crisan, R. Birke, K. Barabash, R. Cohen, and M. Gusat, "Datacenter Applications in Virtualized Networks: A Cross-Layer Performance Study," *presented at IEEE Journal on Selected Areas in Communications*, 2014, pp.77-87.

[38] D. Drutskoy , E. Keller and J. Rexford, "Scalable Network Virtualization in Software-Defined Networks," *IEEE Internet Computing*, v.17, n.2, p.20-27, 2013.

[39] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, "FlowVisor: A Network Virtualization Layer," *OpenFlow Switch Consortium, Tech. Rep*, 2009.

[40] R. Sherwood , G. Gibb , K.-K. Yap , G. Appenzeller , M. Casado , N. McKeown, and G. Parulkar, "Can the Production Network be the Testbed?", In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, 2010, p.1-6.

[41] E. Salvadori, R. Corin, M. Gerola, A. Broglio, and F. De Pellegrini, "Demonstrating Generalized Virtual Topologies in an OpenFlow Network," In *Proceedings of the ACM SIGCOMM 2011 conference on SIGCOMM ACM*, 2011, pp. 458-459.

[42] R. D. Corin, M. Gerola, R. Riggio, F. D. Pellegrini, and E. Salvadori, "VeRTIGO: network virtualization and beyond," *in 1st European Workshop on Software-Defined Networking (EWSDN)*, 2012.

[43] Z. Bozakov, and P. Papadimitriou, "Towards a Scalable Software-Defined Network Virtualization Platform," In *Proceedings of the 2014 IEEE Network Operations and Management Symposium*, 2014.

[44] S. Gutz , A. Story , C. Schlesinger, and N. Foster, "Splendid isolation: a Slice Abstraction for Software-Defined Networks," In *Proceedings of the first workshop on Hot topics in software defined networks*, 2012.

[45] C. Monsanto , N. Foster , R. Harrison, and D. Walker, "A Compiler and Run-Time System for Network Programming Languages," In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2012, pp. 25-27.

[46] P. Skoldstrom, and K. Yedavalli, "Network Virtualization and Resource Allocation in OpenFlow-Based Wide Area Networks," In *Proceedings of SDN'12: Workshop on Software Defined Networks. IEEE ICC*, 2012.

[47] P. Skoldstrom, and W. John, "Implementation and Evaluation of a Carrier-Grade OpenFlow Virtualization Scheme," In *Second European Workshop on Software Defined Networks (EWSDN)*, 2013, pp. 75-80.

[48] Z. Bozakov, and P. Papadimitriou, "Autoslice: Automated and Scalable Slicing for Software-Defined Networks," In *Proceedings of CoNEXT Student*, 2012, pp. 3-4.

[49] A. Al-Shabibi, M. De Leenheer, M. Gerola, A. Koshibe, G. Parulkar, E. Salvadori, and B. Snow, "OpenVirteX: Make Your Virtual SDNs Programmable," In *Proceedings of the third workshop on Hot topics in software defined networking (HotSDN '14)*, 2014, pp. 25-30.

[50] T. Koorevaar, "Dynamic Enforcement of Security Policies in Multi-Tenant Cloud Networks," Master's thesis, Ecole Polytechnique de Montreal, 2012.

[51] K. Agarwal, C. Dixon, E. Rozner, and J. Carter, "Shadow MACs: Scalable label-switching for commodity ethernet," In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking, ACM*, 2014, pp. 157-162.

[52] A. Schwabe, and H. Karl, "Using MAC addresses as efficient routing labels in data centers," In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking, ACM*, 2014, pp. 115-120.

[53] M. Casado , T. Koponen , S. Shenker, and A. Tootoonchian, "Fabric: a retrospective on evolving SDN," In *Proceedings of the first workshop on Hot Topics in Software Defined Networking, ACM*, Helsinki, Finland, 2012.

[54] M. Sloman, and K. Twidle, "Domains: A Framework for Structuring Management Policy," In *Network and Distributed Systems Management*, 1994, pp. 433-453.

[55] S. Pozo, A.J. Varela-Vaca, R.M. Gasca, and R. Ceballos, "Efficient Algorithms and Abstract Data Types for Local Inconsistency Isolation in Firewall ACLs," *4th International Conference on Security and Cryptography (SECRYPT). IEEE Computer Society Press*, 2009.

[56] N. Katta, J. Rexford, and D. Walker, "Infinite Cacheflow in Software-Defined Networks," *Technical Report TR-966-13, Department of Computer Science, Princeton University*, 2013.

[57] K. Phemius, M. Bouet, and J. Leguay, "DISCO: Distributed Multi-domain SDN Controllers," In *IEEE/IFIP Network Operations and Management Symposium (NOMS)*, 2014.

[58] R. Kloti, "OpenFlow: A security analysis," Master's thesis, Swiss Federal Institute of Technology Zurich (ETH), Zurich, Swiss, 2013.

[59] M. Factor, D. Hadas, A. Hamama, N. Har'el, E. K. Kolodner, A. Kurmus, E. Rom, A. Shulman-Peleg, and A. Sorniotti, "Secure Logical Isolation for Multi-Tenancy in Cloud Storage," In *Proceedings of the 29th IEEE MSST*, 2013, pp. 1-5.

[60] 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*, 2010, pp.1-6, .

[61] K. Giotis, C. Argyropoulos, G. Androulidakis, D. Kalogeras, and V. Maglaris, "Combining Openflow and sFlow for an Effective and Scalable Anomaly Detection and Mitigation Mechanism on SDN Environments," *Computer Networks*, v. 62, pp. 122-136, 2014.

[62] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes. "Cloudscale: Elastic Resource Scaling for Multi-Tenant Cloud Systems" In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC '11)*, 2011.

[63] J. Espadas, A. Molina, G. Jimenez, M. Molina, R. RamÃrez, and D. Concha "A Tenant-based Resource Allocation Model for Scaling Software-as-a-Service Applications over Cloud Computing Infrastructures," *Future Generation Computer Systems*, vol.29, no.1, pp. 273-286, 2013.

[64] D. Levin, A. Wundsam, B. Heller, N. Handigol, and A. Feldmann, "Logically Centralized?: State Distribution Trade-offs in Software Defined Networks," In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks (HotSDN),* 2012.

[65] D. Levin, A. Wundsam, A. Feldmann, S. Seethamaran, M. Kobayashi, and G. Parulkar, "A first look at OpenFlow Control Plane Behavior from a Test Deployment," *Technical Report, Technische Universität Berlin, Fakultät Elektrotechnik und Informatik*, ISSN. 1436-9915, 2011

Appendix A

List of REST APIs

Set network configurations:
Method: PUT
Request URI: */setNetworkConfig
Request Body:

```
{
  "Gateway_MAC_Address":"<MAC address>",
  "Gateway_Input_Port":"<Switch port>",
  "External_Address_List":["<List of IP addresses>"]
}
```

Response:
Code: 200, Success
Code: 400, Bad Request
Code: 401, Not Authorized

Set a tenant:
Method: PUT
Request URI: */setTenant
Request Body:

```
{
  "Tenant_Name":"<Tenant name>",
  "Password":"<Tenant password>",
  "Roles":["<Tenant Role>"],
  "Inter-Tenant_Address_Size":"<Size of addresses>",
  "External_Address_Size":"<Size of addresses>",
  "Switch_Ports":["<List of edge switch ports>"],
  "Violation_Action":"<Action in case of violation>"
}
```

Response:
Code: 200, Success
Code: 400, Bad Request
Code: 401, Not Authorized

Set a subnet:
Method: PUT
Request URI: */<tenant name>/setSubnet
Request Body:

```
{
  "Subnet":"<Subnet/Mask>",
  "Static_IP":["<IP address , MAC address>"]
}
```

Response:
Code: 200, Success
Code: 400, Bad Request
Code: 401, Not Authorized

Set a policy group:
Method: PUT
Request URI: */<tenant name>/setPolicyGroup
Request body:

```
{
  "Name":"<Policy group name>",
  "Access_Levels":["<Access levels for a policy group
    >"]
}
```

Response:
Code: 200, Success
Code: 400, Bad Request
Code: 401, Not Authorized

Set a rule (Isolation Policy):
Method: PUT
Request URI: */<tenant name>/<policy group>/setRule
Request body:

```
{
  "Policy_Group":"<Policy group name>",
  "Rule_Name":"<Rule name>",
  "Source_IP_Address":"<IP address>",
  "Destination_IP_Address":"<IP address>",
  "Protocol" : "<Protocol name>",
  "Source_Port":"<Transport port number/ICMP type>",
  "Destination_Port": "<Transport port number/ICMP
    code>"
}
```

Response:
Code: 200, Success
Code: 400, Bad Request

Code: 401, Not Authorized

Advertise a service:
Method: PUT
Request URI: */<tenant name>/setInterTenantService
Request body:

```
{
   "Description":"<Description about the service>",
   "IP_Address":"<IP address of the service>",
   "Port":"<Transport port number of the service>",
   "Protocol":"<Protocol of the service>"
}
```

Response:
Code: 200, Success
Code: 400, Bad Request
Code: 401, Not Authorized

Get network configurations:
Method: GET
Request URI: */getNetworkConfig
Response Body:

```
{
   "Gateway_MAC_Address":"<MAC address>",
   "Gateway_Input_Port":"<Switch port>",
   "External_Address_List":["<List of IP addresses>"]
}
```

Response:
Code: 200, Success
Code: 400, Bad Request
Code: 401, Not Authorized

Get the list of assigned inter-tenant addresses:
Method: GET
Request URI: */<tenant name>/getInterTenantAddress
Response Body:

```
{
   ["<List of IP addresses>"]
}
```

Response:
Code: 200, Success
Code: 400, Bad Request
Code: 401, Not Authorized

Get the list of assigned external addresses:
Method: GET
Request URI: */<tenant name>/getExternalAddress
Response Body:

```
{
   ["<List of IP addresses>"]
}
```

Response:
Code: 200, Success
Code: 400, Bad Request
Code: 401, Not Authorized

Get a policy group:
Method: PUT
Request URI: */<tenant name>/getPolicyGroup/<policy group>
Request body:

```
{
   "Name":"<Policy group name>",
   "Access_Levels":["<Access levels for a policy group
      >"],
   [
    {
      "Policy_Group":"<Policy group name>",
      "Rule_Name":"<Rule name>",
      "Source_IP_Address":"<IP address>",
      "Destination_IP_Address":"<IP address>",
      "Protocol" : "<Protocol name>",
      "Source_Port":"<Port number/ICMP type>",
      "Destination_Port": "<Port number/ICMP code>"
    }
   ]
}
```

Response:
Code: 200, Success
Code: 400, Bad Request
Code: 401, Not Authorized

Get the list of offered inter-tenant services (offered by other tenants):
Method: GET
Request URI: */<tenant name>/getInterTenantServices
Response Body:

```
{
   [
    {
     "Description":"<Description about the service>",
```

```
    "IP_Address":"<Mapped IP address of the service>",
    "Port":"<Mapped port number of the service>",
    "Protocol":"<Protocol of the service>"
   }
  ]
}
```

Response:
Code: 200, Success
Code: 400, Bad Request
Code: 401, Not Authorized

Delete a tenant:
Method: DELETE
Request URI: */deleteTenant/<tenant name>/
Response:
Code: 200, Success
Code: 400, Bad Request
Code: 401, Not Authorized

Delete a policy group:
Method: DELETE
Request URI: */<tenant name>/deletePolicyGroup/<policy group>
Response:
Code: 200, Success
Code: 400, Bad Request
Code: 401, Not Authorized

Delete a rule:
Method: DELETE
Request URI: */<tenant name>/<policy group>/deleteRule/<rule name>
Response:
Code: 200, Success
Code: 400, Bad Request
Code: 401, Not Authorized