Distributed Inter-Domain Routing and Load Balancing in Software Defined Networking (SDN)

By: Taixue Su

Master of Engineering in Computer Application Technology
at Jilin University, 2006

A Thesis

presented to Ryerson University

In partial fulfillment of the

requirements for the degree of

Master of Applied Science

In the program of Computer Networks

Toronto, Ontario, Canada, 2019

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I authorize Ryerson University to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize Ryerson University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

I understand that my thesis may be made electronically available to the public.

Abstract

Distributed Inter-Domain Routing and Load Balancing in Software Defined Networking

Master of Applied Science

Computer Networks

Faculty of Engineering and Architectural Science

Ryerson University, Toronto, 2019

Software Defined Networking (SDN) is an emerging network technology where network control plane, which is programmable, is decoupled from the data forwarding plane and moved to the SDN controller. Originally, an SDN network is controlled by a single controller. As the SDN architecture becomes more complex with multiple SDN domains and corresponding domain controllers, inter-domain routing becomes an important design issue.

There are a number of approaches for SDN inter-domain routing. The most popular approach is the hierarchical approach where a central controller coordinates with lower level domain controllers to control the data flow. The main drawback of the hierarchical approach is that it is not scalable, and the central controller represents a single point of failure. In this thesis, we proposed and designed a distributed inter-domain routing mechanism. The distributed approach enables domain controllers to exchange information with each other directly instead of communicating through the central controller. We also proposed and designed a new load balancing scheme which makes use of the network traffic information to choose a less-congested path among equal-cost multiple paths. We successfully implement the proposed inter-domain routing mechanism and load balancing scheme using Python/Java in the OpenDayLight SDN controller, which is a popular open-source SDN platform. In addition, the test result of the proposed load-balancing scheme shows that it performs better than the scheme based on the round-robin mechanism.

Acknowledgements

Foremost, I would like to express my sincere acknowledgement and gratitude to my supervisor, Dr. Ngok-Wah Ma for his consistent support, patient guidance and continuous encouragement throughout my graduate study. It was a great experience for me to work under his supervision.

I would also like to thank Dr. Yu Xu and Hassan Ahmed for their support during my study, especially their help in my thesis lab environment.

Special thanks to the Computer Networks Program and the Yeates School of Graduate Studies at Ryerson University, for providing this great opportunity and the financial support to my graduate study.

Last, but not least, a special thanks to my family, who constantly supported and encouraged me over the years.

Contents

List of Tables

List of Figures

1. Introduction

1.1 Overview of SDN

Traditional networks consist of many types of devices, such as switches, firewalls, load balancers, routers and so forth as shown in Figure 1. The traditional network's control and data planes are coupled, and the routing mechanism is hop-by-hop with vertically integrated hardware, which make it more complex to configure and reconfigure for rapid response to faults and significant load changes [1].

Software-Defined Networking (SDN) is an emerging technology to relieve the problems of traditional networking. The SDN architecture can abstract underlying infrastructure for all kinds of network applications and services and decouple the network control plane from the data plane. The control plane is moved to the centralized SDN controller, which is dynamic and programmable. The use of the centralized controller simplifies the tasks of network management and network service deployment. [2]

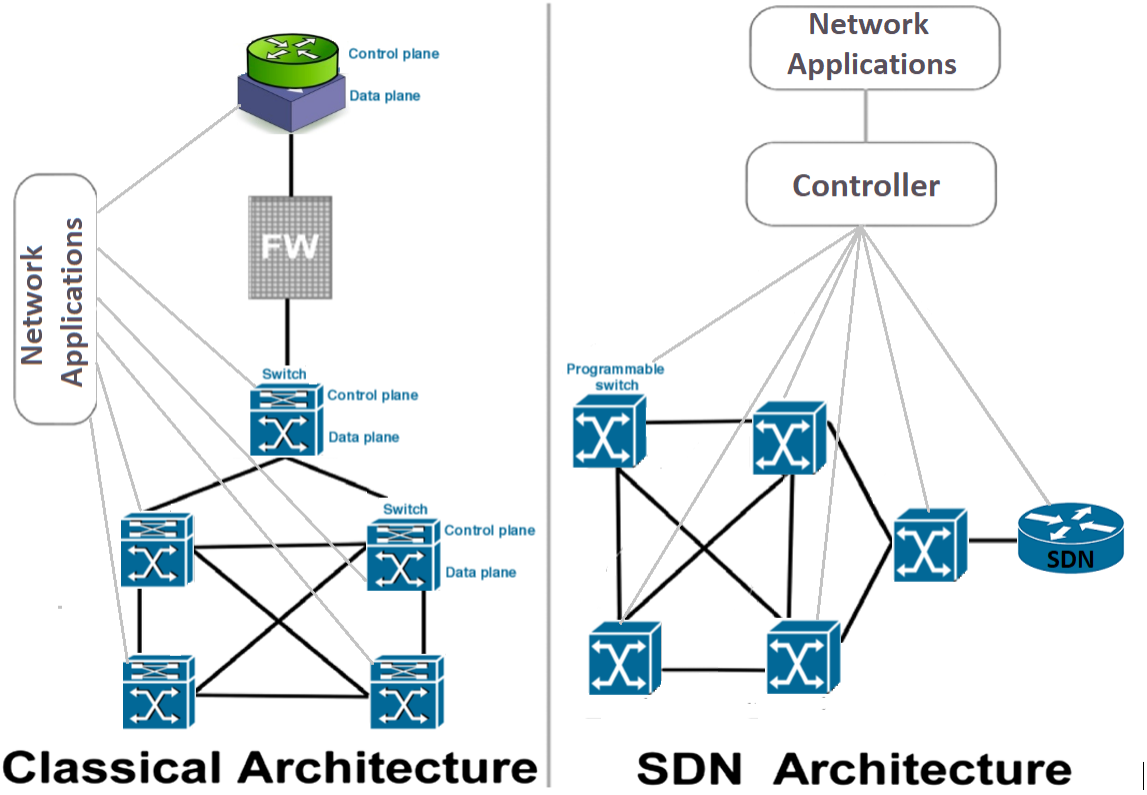

Figure 1 Traditional Network Architecture Vs SDN Architecture

1.2 Problem Statements

Initial design on SDN only deployed a centralized single controller to control the data flows based on the global view of the whole network. However, the current development of the internet, clouding computing and other applications increase the size and complexity of the network. Deploying only a single controller will cause long flow setup latency, lack of scalability and lower availability.

Furthermore, future Internet/Intranet can be interconnected multiple SDN-based domains and each domain is administered by a different controller or controller cluster. Domain controllers are required to communicate with each other and exchange the topology and QoS information to set up the effective flow rules across these domains.

To overcome the limitation of a single controller and support future SDN expansion, some researchers proposed a hierarchical or vertical solution which is comprised of multiple SDN domains with two levels of controllers - one root controller and multiple local domain controllers. These two types of controllers have different responsibilities - local domain controllers deal with the local event in their respective domains, and the root controller handles the global view of the network and communicates with the local domain controllers. Even though the hierarchical approach has higher scalability, it still cannot completely solve the scalability problem as the number of domains increases. Furthermore, the root controller remains a single point of failure.

With a large inter-domain topology, there will usually be equal-cost multiple paths between domains. Thus, an effective dynamic load-balancing scheme is also paramount to extend SDN technology to an inter-domain network.

1.3 Our Approaches

In order to deal with the two issues mentioned in the previous section, we proposed a distributed inter-domains routing mechanism. The distributed approach enables domain controllers to exchange information with each other directly instead of communication through the central controller. We also designed a new load balancing scheme based on the distributed inter-domain routing that is suitable for adaptive SDN data centres.

1.4 Contributions of the thesis

1) A new distributed routing mechanism for SDN inter-domains with the different subnets based on BGP protocol.

2) Based on the distributed routing mechanism, we proposed a new load balancing scheme to support equal-cost multiple path routing.

2. Background and Literature Survey

2.1 Background

Software Defined Networking (SDN) is an emerging network architecture where network control is decoupled from forwarding and is directly programmable [3]. The Open Networking Foundation (ONF) defined the 3 classic SDN-layer as shown in Figure 2, which has won the industry recognition.

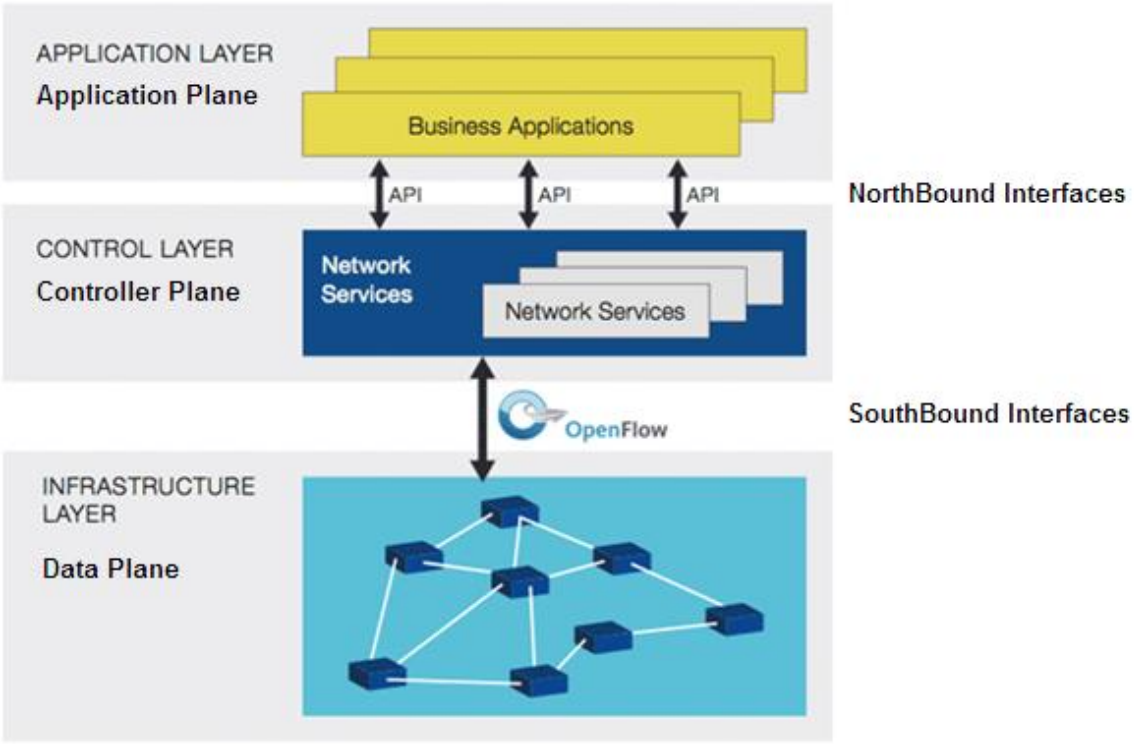

Figure 2 ONF SDN Architecture

1) The Infrastructure Layer/Data Plane consists of the network elements/devices that provide packet switching and forwarding, which expose their capabilities toward the control layer (controller plane) via southbound interfaces from the controller. In our work, we used Open vSwitch (OVS) as the network element in our testbed data centres.
2) The Control Layer/Controller Plane provides the consolidated control functionality that supervises the network forwarding behavior through an open interface. The SDN controller translates the applications' requirements and exerts low-level control over the network elements, while providing relevant information up to the SDN applications;
3) The Application Layer consisted of the SDN application that uses the northbound interfaces (often called NBIs) offered by the Control Layer to obtain topology information and request service across the network.

The SDN controller is the core application of software defined network that facilitates automated network management and controls the communication between network devices and applications.

OpenFlow [5] is an open standard protocol which helps researchers to run their experimental protocols in the campus networks. The current most SDN controllers adopt OpenFlow as the southbound protocol to communicate with network devices.

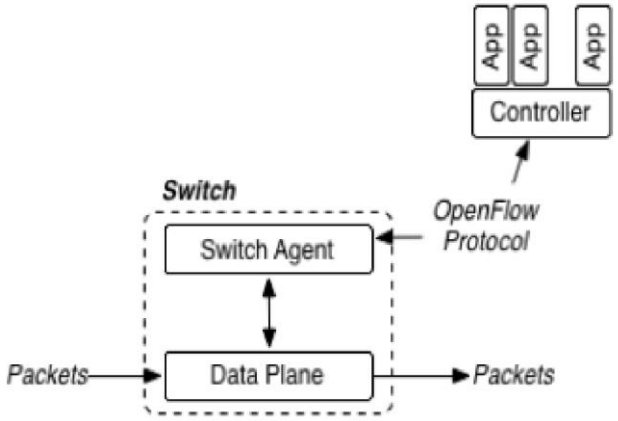

Figure 3 OpenFlow Switch Anatomy

2.2 Literature Survey Related to Software-Defined Inter-Domain Communication and load balancing

Some researchers have proposed some improvement solutions to overcome the SDN initial design's drawbacks. Amin Tootoonchian, et al. [10] proposed the first distributed control plane, called - HyperFlow, which is a logically centralized but physically distributed event-based control approach. The approach deploys a HyerFlow controller application in each SDN controller and HyperFlow uses public/subscribe Openflow message events to propagate the controller's information to others. Thus, all the controllers have a consistent network-wide view and localize all decisions on each controller to minimize the control plane response time. This approach will have scalability limitation for it may cause the generation of a large number of messages (a few thousand events per second) [10].

Teemu Koponen, et al. [11] proposed a distributed SDN control platform (named ONIX) for large-scale networks, which provides more general APIs for control plane and thus turns obscure networking problems into resolvable distributed systems problem for developers.

In " Kandoo "[12], to solve the scalability limitation of the prior distributed approaches, Soheil Hassas Yeganeh, and Yashar Ganjali proposed a novel framework for preserving scalability with minimal developer intervention. The approach separates the controllers into two different layers: one is root controller, which is a logically centralized controller that maintains the network-wide states, and another layer is local controller group, which knows only their owned domain's states. The root controller only processes rare events related global view, while the local controllers handle frequent events such as network-wide statistics collection. The main concern of the

approach is that the root controller remains a single point risk of failure.

Ashvanth Kumar Selvakumaran [13] proposal a hierarchical SDN inter-domain communication approach, which has a central controller on top of all the SDN domain controllers. In the solution, each domain has a local controller and an agent that is used to exchange information between the local controller and the central controller. The approach only considers the inter-domain network as a single layer 2 network. In addition, the approach might still have the scalability issue as the single controller approach when there are a large number of data centre domains.

Seyhan Civanlar, Erhan Lokman, et al.[1] represented a completely distributed scalable Inter-domain SDN controller design without resorting to a top-level controller. In the inter-domain SDN, each SDN controller exchanges its 'summarized' topology of service-enabled paths periodically with other peers, which enables all controller in all domains to make autonomous End-to-end dynamic path/flow management decisions with minimal processing overhead. The inter-domain SDN architecture exchanges the resource reservation messages between controllers, which is different from earlier protocols that use RSVP messages to traverse all routers on the selected route and recent hierarchical SDN inter-domain architecture depending on a top-level authority.

In order to solve the congestion problem in the different SDN architectures, many researchers have proposed some different load balancing approaches for SDN-based data centre. Recently many researchers are focusing on the study of load balancing in inter-domain SDN-based data centre. By migrating the throughput and network latency problems of the single path routing like Spanning Tree Protocol (STP), the proposed scheme in [14] achieved higher throughput and lower latency by a multipath routing mechanism with load balancing and admission control based SDN.

Tim Huang [15] proposed and implemented a load balancing algorithm based on a new path computation & path selection algorithm in SDN Networks. The path computation algorithm uses the basic information of the data center network to calculate all the shortest paths of the network. The path selection algorithm utilizes the output of the path computation and network congestion status to derive a set of paths with equal minimum congestion level and congestion weight and then select an optimal path.

In [16], a dynamic load-balanced path optimization (DLPO) algorithm was proposed to change paths of flows during flow transmissions in different SDN-based data centre network topologies. The DLPO algorithm is composed of two stages: the path initialization stage, which will try to find a temporary path according to available bottleneck links' bandwidth, and dynamic path optimization stage which retrieves load statistics from switches and detects the load-balance status by the Openflow protocol, and subsequently, it will trigger to balance link loads if link loads are imbalanced. The authors also proposed a priority-based flow table updating strategy based on the priority field in the flow to redistrict flows from the heavy-loaded path to light-loaded path without interrupting flow transmission problem.

In [17], an approach for Traffic Engineering (TE) in SDNs without optimization was proposed. It uses a logically hierarchical controller architecture, where TE is performed in two levels: the core switches controllers (CSC) maintains a very small number of core pre-defined simple shortest path rules in core switches; and access switches controllers (ASCs), which use a central measure to make traffic distribution decision in the access switches in its own domain. However, the approach uses a hierarchical controller architecture, thus, is susceptible to a single point of failure.

2.3 Proposed Scheme

We proposed a fully distributed approach to implement inter-domain routing and load balancing. This approach is comprised of two levels of forwarding: inter-domain forwarding and intra-domain forwarding. Inter-domain forwarding is implemented by setting the IP forwarding rules between domains based on the global inter-domain level topology of the entire network. In the intra-domain forwarding, each domain controller controls the layer-2 forwarding inside each domain. The mechanism is flexible, scalable and distributed to support inter-domain routing.

In order to improve our scheme' availability and efficiency, we also introduced a load balancing function. The load-balancing scheme uses inter-domain level topology to get the global topology of the entire networks and then find out all the equal-cost inter-domain shortest paths (ECIPs).

2.4 Comparison with the Approach in Literature

1) Our approach does not rely on an extra root/central controller like most hierarchical approaches. It is a fully distributed approach for SDN inter-domains' communication.
2) Compared with the other distributed approaches, our approach resorts to Westbound/Eastbound protocol to share topology and QoS information of entire networks rather than only neighbour domain's information or summarized topology.
3) Our approach also implemented a new load balancing schemes in the distributed SDN inter-domain routing architecture based on global topology and QoS information.

3. Design of Software-Defined Inter-Domain Communication and New Load Balancing

The objective of this thesis is to design a completely distributed SDN architecture to implement data forwarding and load balancing among multiple domains. In this chapter, we first introduced an overview of our proposed approach. We then described the topology discovery, inter-domain path selection, and load balancing algorithms in detail.

3.1 Overview of SDN Inter-Domain Routing and Load Balancing

The overview workflow of our proposed approach is shown in Figure 4. The proposed approach includes 4 modules: topology discovery, path computation, flow installation, and load balance modules. These modules are briefly described as follows.

1) Topology discovery: the approach proposes 2-level topology discovery. The first level is the intra-domain level that discovers the internal topology of a domain; the second one is the inter-domain level that discovers the global network topology among the domains. Note that each domain represents a subnet with its own unique IP address space.
2) Path computation: the module calculates the shortest path(s) between source and destination hosts in two levels. The result will be used in the flow installation and load balance modules.
3) Flow installation: the module pushes flows in the related border switches along the domain-level shortest path between source/destination host pairs, and it pushes flows in the relevant switches along the shortest path within a domain. If only one shortest path exists, the module will use the path computation's result as input, otherwise, it will use the result of the load balance module as input.
4) Load balance: If there are several equal-cost shortest paths between source and destination host pair from path computation, the module will detect the network congestion status to select a suitable path.

The parts of the above modules associated with the local-domain level have been comprehensively covered and implemented in the literature and are out of the scope of this thesis. In the following sections, we will provide more detail of the parts of the modules associated with the inter-domain level.

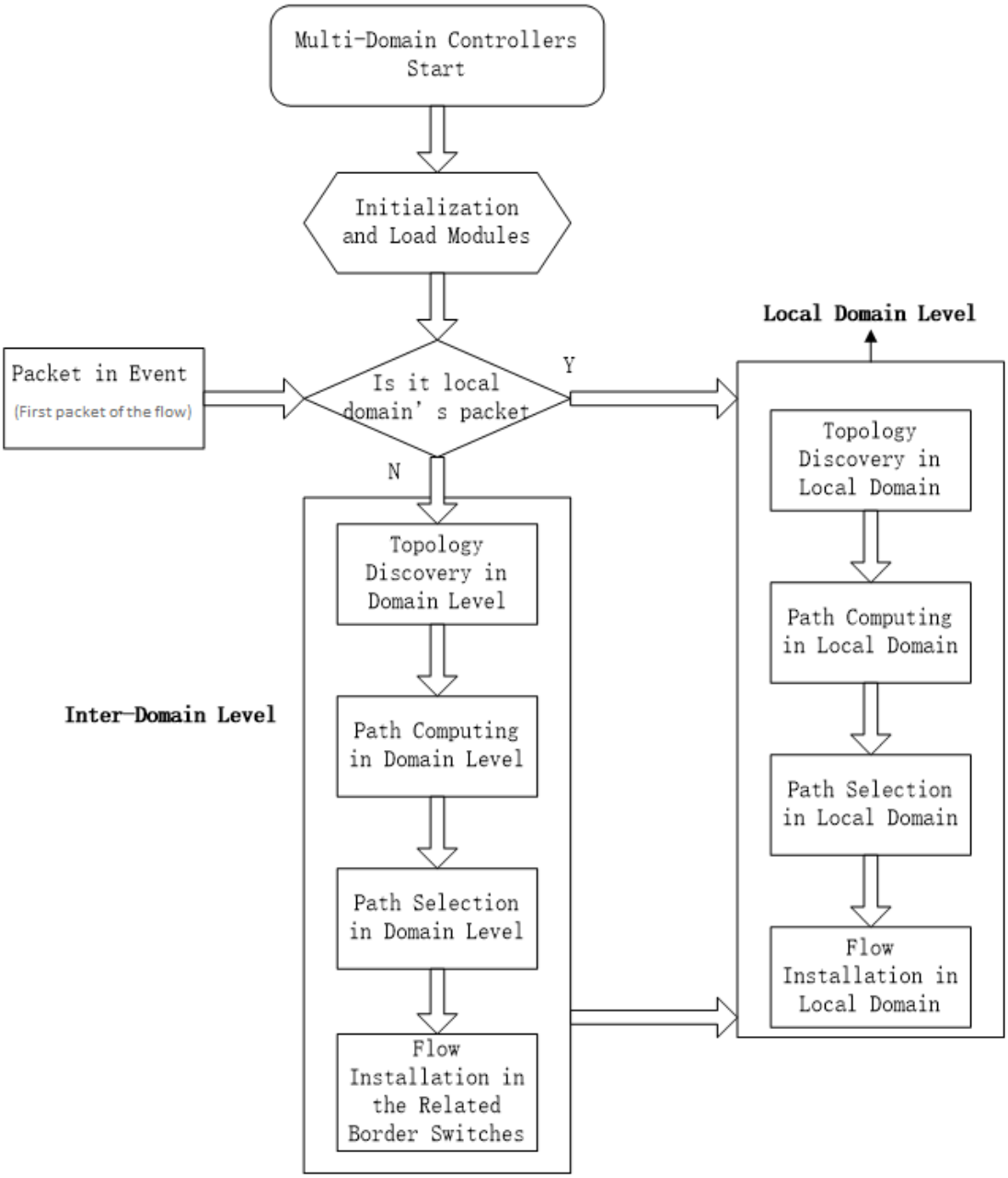

Figure 4 Overview Work Flow of SDN Inter-Domain Routing and Load Balancing

3.2Topology Discovery in Global Inter-Domain Level

3.2.1 The Information Essential to be Exchanged among Domain Controllers

Each domain controller can obtain the information of the intra-domain topology. For example, the controllers of domains 1, 2 and 3 in Figure 5 can acquire their intra-domain topologies through Link Layer Discovery Protocol (LLDP) [18]. However, the controllers could not recognize the

external links among domains using LLDP alone. Consequently, controllers need to exchange the intra-domain topology among themselves. Let us first define the symbols of some essential information.

Let denote D_i as Domain *i,* where *i* is the id of the domain. The global network thus consists of a *set of Domains = {*D_1*,* D_2*, ...* D_i*, ...,* D_n*}*

$S_{i,n}$ as *switch n* in Domain D_i. We define S_i as the set of all $S_{i,n}$ in D_i

$H_{i,n}$ as *host n* in Domain D_i ,

P_i as the port *i* of the switch/host,

define L_i as the set of all link *of* Domain D_i *, and the Link from port p of* $S_{i,n}$ *to port q of* $S_{j,m}$ *is denoted as [*$S_{i,n}$*:*P_p *,* $S_{j,m}$*:*P_q*].* When *i = j,* the link is the internal link of Domain D_i, otherwise, it is the external link between Domain D_i and D_j. Note that the link is unidirectional. The switch associated with the first part of the link is the source, while the switch associated with the second part is the destination.

Similarly, the link between port p of switch $S_{i,n}$ *and port q of host* $H_{i,m}$ *in Domain* D_i *, can be donated as [*$S_{i,n}$*:*P_p *,* $H_{i,m}$*:*P_q*]*

If Domain *i* has a link *[*$S_{i,n}$*:*P_p *,* $S_{j,m}$*:*P_q*]* and Domain j also has the corresponding link *[* $S_{j,m}$*:*P_q*,* $S_{i,n}$*:*P_p*]*, we can deduce that Domain i and Domain j are connected by the bi-directional external link consisting of *[*$S_{i,n}$*:*P_p *,* $S_{j,m}$*:*P_q*]* and *[* $S_{j,m}$*:*P_q*,* $S_{i,n}$*:*P_p*].*

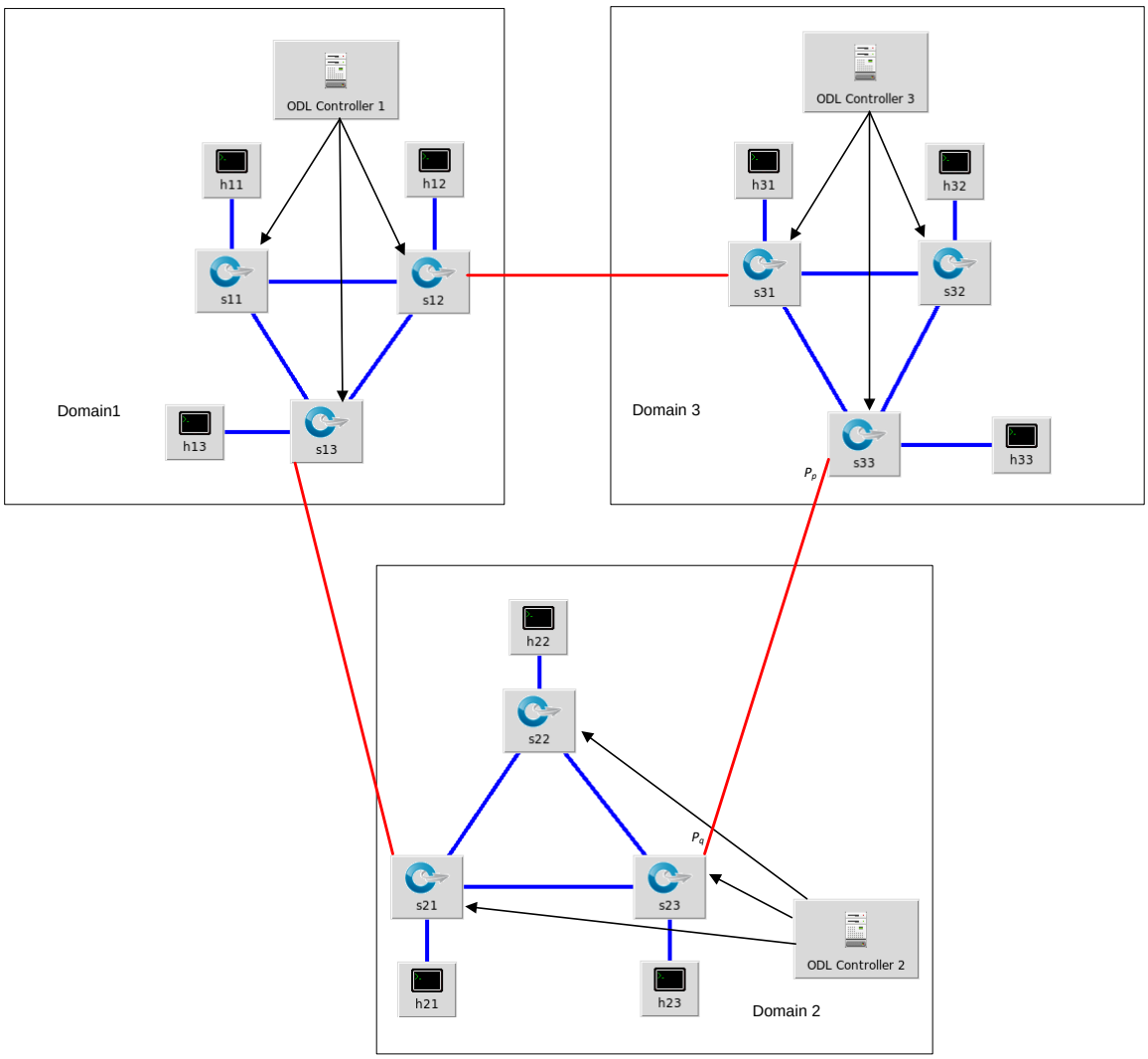

Figure 5 Basic Inter-Domain Topology

In our design, the external links among domains will be learned via BGP protocol. Once the external links are learned, the overall topology at the domain level can be derived by every domain controller without requiring a central controller. Note that each domain is treated as a single routing entity at the domain level. For example, given the inter-domain topology in Figure 6a, we can determine the corresponding domain level topology in Figure 6b.

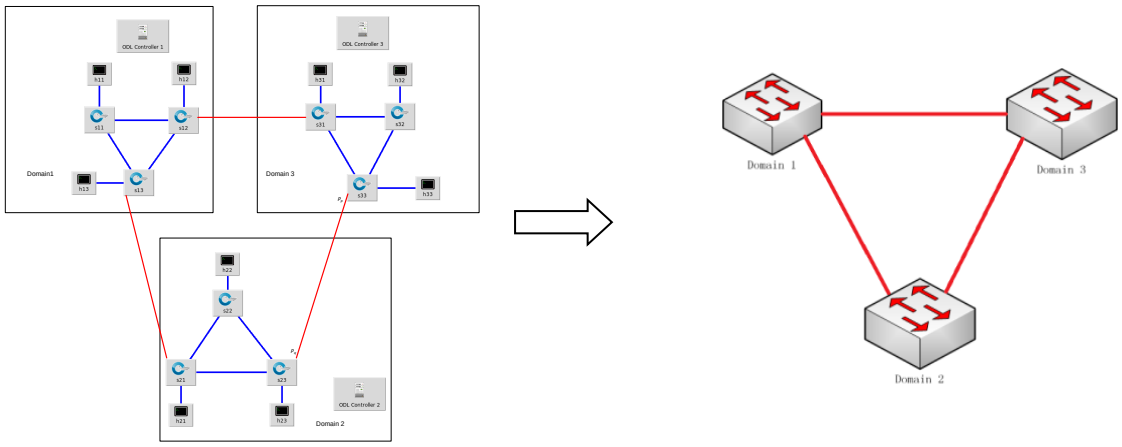

Figure 6a. 3-Physical Topology **Figure6b. Corresponding Domain Level Topology**

3.2.2 Algorithm of Discovery the Domain Level Topology

The pseudocode below describes the topology discovery algorithm which is run in each domain's controller to derive the domain level topology information.

Algorithm 3.1 - Pseudocode of Topology Discovery

1: Call the northbound API of current local domain D_k to get the local topology

2: Add all local switches into the local switches set S_k, and add all nodes and links L_k into local domain topology graph *G*.

3: **for** D_i in Domains **do**

4: **if** D_i is not D_k **then**

5: Acquire the topology information (S_i *and link set* L_i) of the domain D_i by west/east protocol

6: Add D_i into domain level topology graph *domainGraph* as a node

7: **if** $S_{i,n}$ **not in** S_k **then**

8: add $S_{i,n}$ into the peer switches list – *peerSwList*

9: add the links connected to $S_{i,n}$ into external links - *externalLinkList*

10: **end if**

11: **end if**

12: **end for**

13: **for** $S_{i,n}$ **in** *peerSwList* **do**

14: **for** D_k in Domains **do**

15: **if** $\exists[S_{k,m}, S_{i,n}] \in$ *externalLinkList* **and** *peer Domain* D_i*'s Link* $[S_{i,n}, S_{k,m}] \in Li$ **then**

16: identify $S_{i,n}$, $S_{k,m}$ and as border Switches and *link* **[**$S_{i,n}$, $S_{k,m}$**]** as an external link;

17: record the port number of the two border switches;

18: add a link[D_i, D_k] into *domainGraph*

19: **end if**

20: **end for**

21: **end for**

22: **for** D_i in Domains **do**

23: **for** D_j in Domains, $j>=i+1$ **do**

24: **if** D_i is not D_k and D_j is not D_k **then**

25: **if** at ∃ **[**$S_{i,n}$, $S_{j,m}$**]** ∈ L_i **and** ∃ **[**$S_{j,m}$, $S_{i,n}$**]** ∈ L_j **then**

26: add link[D_i, D_j] into *domainGraph*

27: **end if**

28: **end if**

29: **end for**

30: **end for**

Let us use an example based on Figure 7a to explain how to build the domain-level topology. In Figure 7a, each controller has the following local topology information:

Domain D_1:

Switch Set S1: { s11, s12, s13}

Host Set H1: { h11, h12, h13, h14}

Link Set L1: {[s11:1, s13:2], [s11:2, s12:2], [s11:3, h14:1], [s11:4, h11:1], [s12:1, s13:1], [s12:3, h11:1],[s13:3, h13:1], [s13:3, h13:2], [s13:4, s41:4], [s13:5, s21:4]}

Domain D_2:

Switch Set S2: { s21, s22, s23}

Host Set H2: { h21, h22, h23}

Link Set L2: {[s21:1, h21:1], [s21:2, s23:2], [s21:3, s22:2], [s21:4, s13:5], [s22:1, h22:1], [s22:2, s21:3], [s22:3, s23:3],[s22:4, s32:4], [s23:1, h23:1], [s23:2, s21:2] , [s23:3, s22:3]}

Domain D_3:

Switch Set S3: { s31, s32, s33}

Host Set H3: { h31, h32, h33, h34}

Link Set L3: {[s31:1, h31:1], [s31:2, h34:1], [s31:3, s32:2], [s31:4, s33:1], [s31:5, s43:4], [s32:1, h32:1],[s32:3, s33:2], [s32:4, s22:4], [s33:3, h33:1] }

Domain D_4:

Switch Set S4: { s41, s42, s43}

Host Set H4: { h41, h42, h43}

Link Set L4: {[s41:1, s43:1], [s41:2, s42:1], [s41:3, h41:1], [s41:4, s13:4], [s42:2, s43:2], [s42:3, h42:1],[s43:3, h43:1], [s43:4, s31:5]}

Let's focus on domain D_1. By scanning the link set L1 and switch set S1, the controller of D_1 finds the following links whose destination switches do not belong to S1:

[s13:4, s41:4] and [s13:5, s21:4]

Consequently, these are external links. The controller of domain 1 then advertises these external links to the other controllers. At the same time, it will receive the external link advertisements from the controllers of domains 2, 3 and 4.

From domain 2: [s21:4, s13:5], [s22:4, s32:4]

From domain 3: [s31:5, s43:4], [s32:4, s22:4]

From domain 4: [s41:4, s13:4], [s43:4, s31:5]

By examining all the external link, each controller can derive the domain-level topology as shown in Figure 7b.

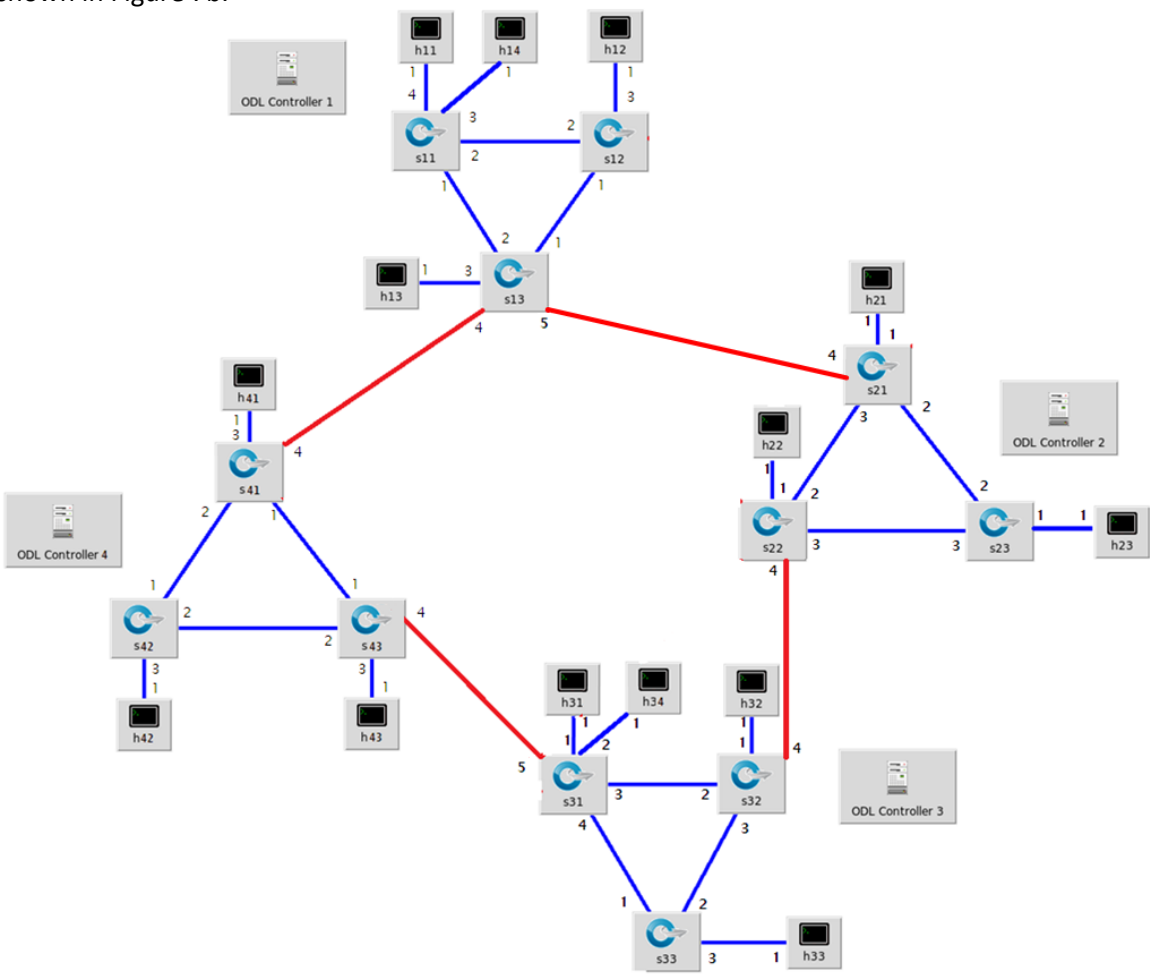

Figure 7a 4-Domain Full Topology

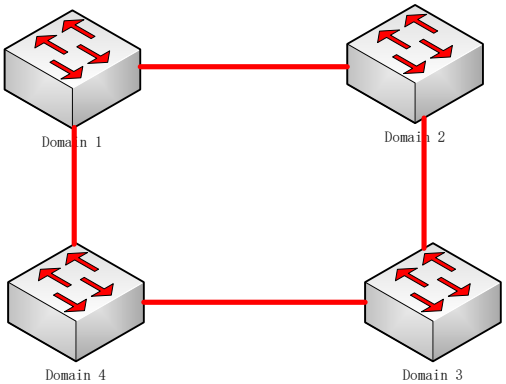

Figure 7b Domain-level Topology

3.3 Inter-Domain Path Computing

Algorithm 3.2 derives the domain level topology including domains and inter-domain links. Based on the domain-level topology, each controller can compute the shortest inter-domain path(s) between source and destination hosts. In this thesis, we assume all the external links have the same bandwidth. Consequently, we will first use the domain hops as the metrics to compute the shortest path. Note that our algorithm can easily be extended to accommodate cases where there are different bandwidths for different external links. The pseudocode of the path computing algorithm is given in Algorithm 3.2 below.

Algorithm 3.2 - Pseudocode of Path Computing

```
graph G ← Local Topology Information
domainGraph ← Global Topology Discovery
srcDomain ← the Domain of the source host
dstDomain ← the Domain of the destination host
shortestPaths ← get all shortest paths（domainGraph, srcDomain, dstDomain）
for shortestpath in shortestPaths do
    if current Domain is the end node of the shortestpath then
        return shortestpath
    else
        identify the border switch and port along the shortestpath
        weight metric of the shortestpath ← call the path computing method of the peer domain along the shortest path
    end if
end for
shortestpath ← the path which has the least weight metric of all the shortestpath
return shortestpath
```

Let us use Figure7b's topology as an example to illustrate the algorithm. For the domain level topology, Domain Nodes Set is $\{D_1, D_2, D_3, D_4\}$, Links set is $\{[D_1, D_2], [D_1, D_4], [D_2, D_3], [D_3, D_4]\}$. If computing the domain level path from h11 to h31 using domain hops as path metrics, the source host h11's domain is D_1, destination host h31's domain is D_3, so we find two domain level shortest paths $\{D_1, D_2, D_3\}$ and $\{D_1, D_4, D_3\}$.

Let us use another topology - Figure 8 to illustrate the shortest path computation at the domain-lever. The domain graph's nodes set is

$\{D_1, D_2, D_3, D_4, D_5, D_6\}$, links set is $\{[D_1, D_2], [D_1, D_3], [D_3, D_4], [D_4, D_5], [D_5, D_6], [D_2, D_6]\}$. There are 2 paths between D_1 and D_6: one path is $\{D_1, D_3, D_4, D_5, D_6\}$ and another is$\{ D_1, D_2, D_6\}$, so the shortest path is $\{D_1, D_2, D_6\}$.

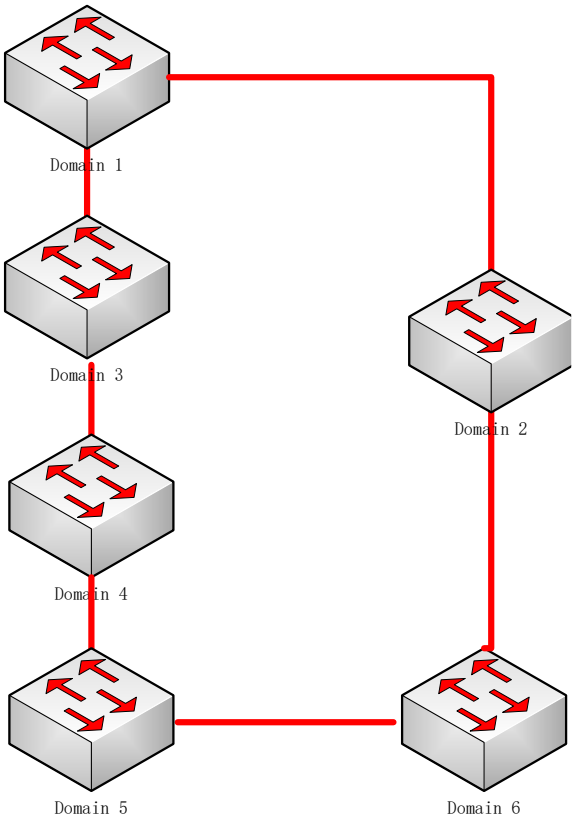

Figure 8 Domain-level Topology (6 Domains)

3.4 Inter-Domain Flow Installation

From algorithms 3.1 and 3.2, we get the domain-level topology and then compute the shortest path in the domain level. When the controller of the source domain receives the first packet sent by a source host to a destination host in the other domain (packetin event), it needs to install inter-domain flows in the border switches according to the inter-domain shortest path.

Normally, the controller will make a forwarding decision based on the destination MAC address. For the inter-domain traffic, however, the forwarding decision is based on the destination IP address. Thus, the controller needs to distinguish intra-domain traffic from the inter-domain traffic to process these two types of packets properly. In our approach, we introduce a virtual gateway MAC address and the corresponding IP address. The virtual gateway IP address will be configured in each host as the IP address of the default gateway. The mapping of the virtual gateway MAC and IP addresses is also configured in the ARP table of the host. With the above

configurations, when a host wants to send a packet to a host located in the other domain, it will insert the virtual gateway MAC address at the layer 2 header of the packet. Subsequently, when a controller sees the virtual gateway MAC address in a packet, it knows that it is processing an inter-domain packet and examines the IP header of the packet to determine the shortest inter-domain path.

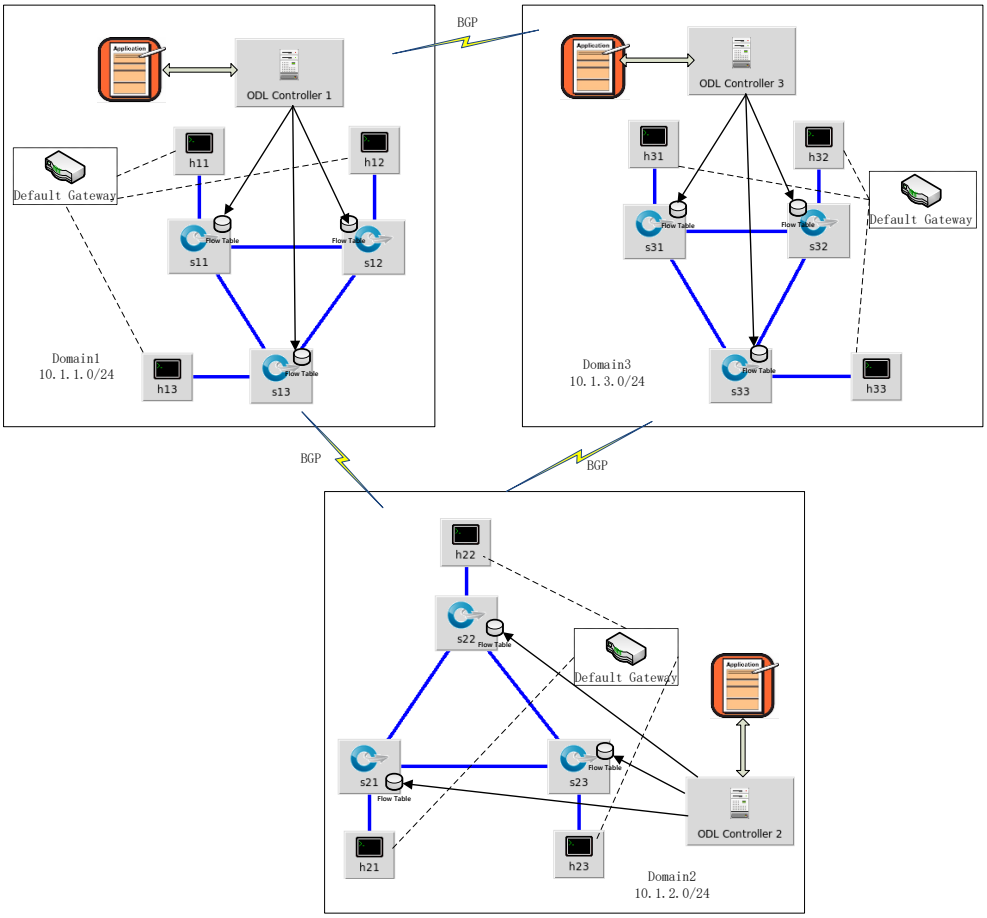

Figure 9 Design of Distributed Inter-SDN Domain Communication

Unlike some approaches such as [14], this approach can identify the border switches dynamically rather than relying on the fixed name of the switch/host. According to the output of the prior modules, we can get the shortest paths in domain level and the border switches of the paths in each domain. Thus, the packet-in event can trigger the flow installation module to install forward and reverse flow into border switches at the source domain, the destination domain and the transit domains along the domain-level shortest path. The source domain controller determines the shortest path between the source switch and the chosen border switch. Similarly, the transition domain controller determines the shortest path between the two border switches and the destination domain controller determines the shortest path between the border switch and the destination switch. Then these domain controllers install flows which match the MAC address and IP address along the intra-domain shortest paths.

Algorithm 3.3 - Pseudocode of Flow Installation

```
1:    Input: domain shortest path, source host, destination host
2:    for domain in domain shortest path do
3:      if domain is source domain then
4:        identify the border switch and port to downstream peer domain in shortest path
5:        push the forward and reverse flows to the border switch
6:      determines the intra-domain shortest path between the source switch and the chosen
border switch
7:        push flows along the intra-domain shortest path
8:      else if domain is destination domain then
9:        identify the border switch and port to upstream peer domain in shortest path
10:       push the forward and reverse flows the border switch
11:     determines the intra-domain shortest path between the chosen border switch and the
destination switch
12:       push flows along the intra-domain shortest path
13:     else
14:       identify the border switches and ports' number which are linked downstream and
upstream domain in the transit domain
15:       push the forward and reverse flows to downstream domain
16:       push the forward and reverse flows to upstream domain
17:       determines the intra-domain shortest path between the two chosen border switches
18:       push flows along the intra-domain shortest path
19:   end for
```

Let us study an example based on Figure 7's topology. Supposed that source and destination hosts are h11 and h31, respectively, and the algorithm adopts the shortest path {D_1, D_2, D_3}. Based on the shortest path, the source controller can determine that the border switch is s13 and its port 5 is connected to the downstream Domain D_2. So the Flow-Installation module pushes the following flow rules into flow table of s13 to lead the traffic to the downstream domain, D_2, and the reversed traffic from D_2 to D_1:

Table 1 Flow Rules of Border Switch s13 of Domain D_1

Match Fields	Action	Domain:Switch
IP Src(h11)-> IP Dst(h31) Inport(2)	Output at port 5	D_1: s13
IP Src(h31)-> IP Dst(h11) Inport(5)	Output at port 2	D_1: s13

Note that the installed flow rules make the border switch behave like a router.

As the destination of the packet is not inside the Domain1, so when the controller receives the packet-in message, it will find its designation MAC address is virtual gateway MAC address ('00:00:00:00:00:64'). The controller will then find the intra-domain shortest path between the source switch s11 and the chosen border switch s13 - [s11, s13], and push the flows along the intra-domain shortest path as shown in the following table 2.

Table 2 Flow Rules inside Domain D_1

Match Fields	Action	Domain:Switch
IP Src(h11)-> IP Dst(h31) Mac Dst(00:00:00:00:00:64)	Output at port 1	D_1: s11

When the packet reaches a transit Domain D_2, via the incoming border gateway s21, the controller of D2 identifies the packet as an inter-domain packet since it arrives at the external link. The controller of D2 thus processes the IP header and determines outgoing border gateway, s22. Consequently, the Flow-Installation module in the controller of D_2 will push a forward flow rule into the flow table of the border switch s22 to forward packet at port 4 to D_3 and push a reversed flow rule to receive the reserved packet from D_3, and also install the similar bidirectional flows into another border gateway s21.

Table 3 Flow Rules of Border Switches of Domain D_2

Match Fields	Action	Domain:Switch
IP Src(h11)-> IP Dst(h31), inport(4)	Output at port 3	D_2: s21
IP Src(h31)-> IP Dst(h11) inport(3)	Output at port 4	D_2: s21
IP Src(h11)-> IP Dst(h31) inport(2)	output at port 4	D_2: s22
IP Src(h31)-> IP Dst(h11) Inport(4)	output at port 2	D_2: s22

When the packet reaches the destination domain D_3 through the border switch s32, the Flow-Installation module in the controller of D_3 pushes a forward and reserved flow rules into flow table of the border switch s32 to lead the traffic to h31 and h11, respectively.

Table 4 summarizes the flow rules installed at the border switches of domains 1, 2 and 3.

Table 4 Flow Entry of Border Switches

Match Fields	Action	Domain:Switch
IP Src(h11)-> IP Dst(h31) inport(2)	Output at port 5	D_1: s13
IP Src(h31)-> IP Dst(h11) Inport(5)	Output at port 2	D_1: s13
IP Src(h11)-> IP Dst(h31), inport(4)	Output at port 3	D_2: s21
IP Src(h31)-> IP Dst(h11) inport(3)	Output at port 4	D_2: s21
IP Src(h11)-> IP Dst(h31) inport(2)	output at port 4	D_2: s22
IP Src(h31)-> IP Dst(h11) Inport(4)	output at port 2	D_2: s22
IP Src(h11)-> IP Dst(h31), inport(4)	Output at port 2	D_3: s32
IP Src(h31)-> IP Dst(h11) inport(2)	Output at port 4	D_3: s32

3.5 Load Balancing

There are some classic load balancing algorithms available for SDN traffic engineering such as Random, Global First Fit, and Round Robin. Round Robin is widely used in industry and extremely simple to implement.

Round Robin (RR) load-balancing algorithm pre-computes all the available equal-cost shortest paths in domain-level for all source and destination domains pair and save these paths. The pre-computed equal-cost paths will be selected for different traffic flows in round-robin fashion.

The RR algorithm does not provide a true load balancing for different traffic flows may have different traffic characteristics: some require more bandwidth while others have a longer life time. In this section, we introduce a mechanism to improve the load balancing performance.

3.5.1 New Metric of Proposed Load Balancing Scheme

In our load-balancing scheme, if the source domain finds that there are two or more equal-cost shortest paths, it will obtain the load metrics of all these paths and choose the one with the lowest load metric. The load metric of a path is the sum of the domain load metrics of the domains that form the path. The domain load metric, in turn, is the load metric of an intra-domain path of that

domain. Individual domain controller computes the load metric of the domain. For the transition domain, the intra-domain path is the path connecting the incoming and outgoing border switches of that domain. For the source domain, the intra-domain path is the path connecting the source and the outgoing border switch. Finally, for the destination domain, the intra-domain path connects the incoming border switch and the destination.

For a given intra-domain path with a set of links, the load metric is the function of the number of links, the bandwidths and the traffic loads of those links. Let

$$L_i = \{l_{i1}, l_{i2}, \dots, l_{i,n}\}$$

where L_i represents all the links associated with an intra-domain path i and l_{ij} represents one of those links. The total number of links of the path is n. The bandwidth of link l_{ij} is denoted as B_{ij} and the traffic load of the link is b_{ij}. Based on this definition, the load metric, LM_i, of an intra-domain path i is given by:

$$LM_i = \sum_{j=1}^{n} \frac{b_{ij}}{B_{ij}} \qquad (1)$$

Equation 1 tries to capture the load of all the links of the intra-domain path and the amount of resource (link bandwidth) required for using the path. The traffic load of b_{ij} is obtained by the controller from the switches associated with the link l_{ij} in every 5-sec window. The choice of the length of the window of 5 sec is the balance between bandwidth usage and the currency of the load information.

Similarly, the load metric of an inter-domain link LR_{ij} between two domain D_i and D_j is also defined as the ratio of the bandwidth of the link, Br_{ij}, and the traffic load of the link, br_{ij}:

$$LR_{ij} = \frac{br_{ij}}{Br_{ij}} \qquad (2)$$

The load-balancing algorithm will use the following mechanism to gather domain load metrics. First, the source domain controller makes load-metric inquiries to its adjacent downstream domains associated with all the equal-cost paths. These inquiries trigger the downstream domain controllers to make inquiries to their own downstream domain controller(s). This procedure repeats at each transition domain until the inquiries from different equal-cost paths reach the destination domain controller. The destination domain controller responds with the domain load metric(s) of its domain. The transition domain controller adds the load metric from the response with the load metric of its domain and passes the sum to the upstream domain controller. Thus, when the source domain controller collects the responses, it can compute the overall load metrics of all equal-cost paths. The source domain controller then will choose the path with the lowest

overall load metric. The following Pseudocodes describe the algorithms used by the source domain, transition and destination domain controllers.

Source Domain controller

1. Given the source and the destination, determines all the equal-cost paths, $\{P_1, \dots, P_n\}$, and the corresponding downstream domains. Let $\{D_1, \dots, D_n\}$ be the set of these downstream domains and $\{S_1, \dots, S_n\}$ be the corresponding border switches that connecting to these domains, respectively.
2. Determine the intra-domain paths from the source to each of these border switches in $\{S_1, \dots, S_n\}$. Calculate the corresponding load metric, $\{LM_1, \dots, LM_n\}$
3. Calculate the load metric of the inter-domain link to to $\{D_1, \dots, D_n\}$, let the load metric as $\{LR_1, \dots, LR_n\}$
4. Sends inquires to domain controllers whose domains belong to $\{D_1, \dots, D_n\}$. The inquiry contains the source and destination information.
5. The response received from an adjacent domain *j* contains the load metric of the path, DM_j, measured from domain *j* to the destination domain.
6. For every path in $\{P_1, \dots, P_n\}$, determine the overall load metric of P_j, PM_j:

   $$PM_j = DM_j + LM_j + LR_j$$

   If no response is received from domain D_j within the waiting window of 2 sec, P_j, is considered ineligible for the selection.

7. Choose P_j if PM_j = min ($\{PM_1, \dots, PM_n\}$).

Transit Domain controller

1. Upon receiving a load-metric inquiry from the upstream domain, D_u, at an incoming border switch, $S_{I,}$ the controller determines all the equal-cost paths, $\{P_1, \dots, P_n\}$, the corresponding downstream domains $\{D_1, \dots, D_n\}$ and the border switches that connecting to these domains $\{S_1, \dots, S_n\}$.
2. Determine the intra-domain paths from S_I to each of the border switches in $\{S_1, \dots, S_n\}$. Calculate the corresponding load metric, $\{LM_1, \dots, LM_n\}$
3. Calculate the load metric of the inter-domain link to $\{D_1, \dots, D_n\}$, let the load metric as $\{LR_1, \dots, LR_n\}$
4. Sends inquires to domain controllers whose domains belong to $\{D_1, \dots, D_n\}$.
5. The response received from an adjacent domain *j* contains the load metric of the path, DM_j, measured from domain *j* to the destination domain.
6. For every path in $\{P_1, \dots, P_n\}$, determine the load metric, PM_j:

   $$PM_j = DM_j + LM_j + LR_j$$

   If no response is received from domain D_j within the waiting window of 2 sec, P_j, is considered ineligible for the selection.

7. Send PM_j back to D_u, where PM_j = min ($PM_1, \dots, PM_n$}.

Destination Domain controller

1. Upon receiving a load-metric inquiry from the upstream domain, D_u, at an incoming border switch, S_I, the controller determines the intra-domain path from S_I to the destination. It then calculates the corresponding load metric, LM_d.
2. Send LM_d back to D_u.

3.5.2 Example

Let us use the topology in Figure 10 as an example to illustrate the proposed load balancing approach. When h11 sends packets to h33, there are two equal shortest paths according to algorithm 3.2 in section 3.3:

P1: {D_1, D_2, D_3}

P2: {D_1, D_4, D_3}

The figure also shows the traffic loads and bandwidths of the relevant links. The traffic load is measured by the number of bytes sent on the link in a 5-sec window.

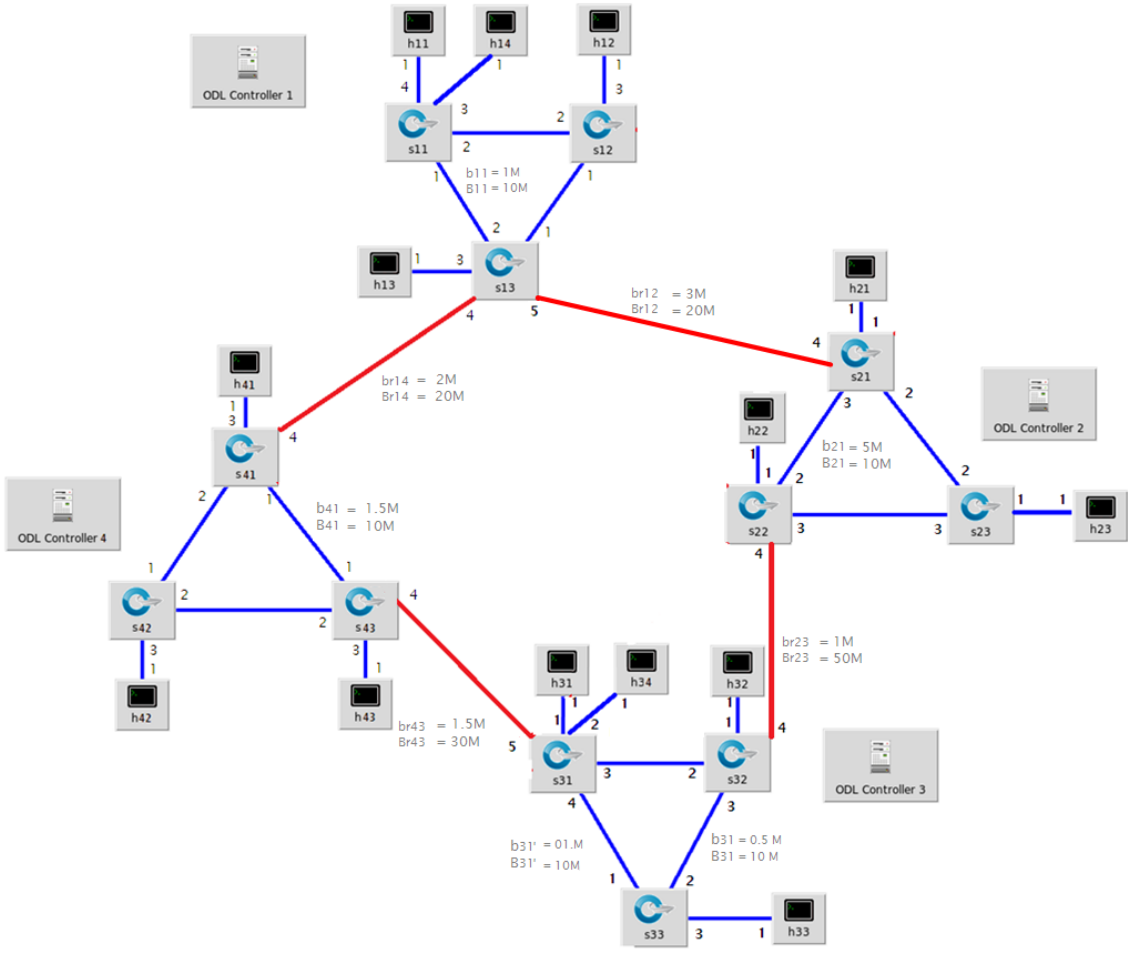

Figure 10 Topology Indicated QoS info along Equal Shortest Paths

In Source Domain 1: paths P_1 and P_2 have the same border switch s13, and the shortest path from the source switch s11 to the border switch s13, $L_1 = \{l_{11}\}$ = {[s11:1, s13:2]}, so the load metric of P_1 and P_2 is same , and $LM_1 = LM_2 = b_{11}/B_{11} = 1/10 = 0.1$. The controller of Domain1 also calculates the load metric of inter-domain link Lr_{12} ([s13:5, s21:4]) to Domain2 in P_1 and Lr_{14} ([s13:4, s41:4]) to Domain4 in P_2 : $LR_1 = br_{12}/Br_{12} = 3/20 = 0.15$ and $LR_2 = br_{14}/Br_{14} = 2/20 = 0.1$. Then the controller sends inquires to the controllers of Domain 2 (associated with P_1) and Domain 4 (associated with P_2).

In Transit Domain 2(P1): there is only one path to destination Domain 3 and the incoming switch is s21 and the outgoing border switch is s22, so the link set along the shortest path between the two border switches is $L_2 = \{l_{21}\}$ = { [s21:3, s23:2]}, and $LM' = b_{21}/B_{21} = 5/10 = 0.5$. The load metric of inter-domain link Lr_{23} ([s22:4, s32:4]) to destination Domain 3: $LR' = br_{23}/Br_{23} = 1/50 = 0.02$. The domain controller sends inquire to its downstream domain - Domain 3's controller.

In Transit Domain 4(P2): similar with Domain 2, $LM'' = b_{41}/B_{41} = 1.5/10 = 0.15$, and the load metric of inter-domain link Lr_{43} ([s41:4, s31:5]) to destination Domain 3: $LR'' = br_{43}/Br_{43} = 1.5/30 = 0.05$. The domain controller also sends an inquiry to the destination Domain 3.

In Destination Domain 3: the controller receives 2 inquiries from Domain 2 and 4.

For Domain2's inquiry: the link set along shortest path from s32 to the destination switch s33, $L_3 = \{l_{32}, l_{33}\}$ = {[s32:3, s33:2]}. It returns $LM_d = b_{31}/B_{31} = 0.5/10 = 0.05$ to Domain 2.

Thus, Domain 2 gets the $DM' = 0.05$ and then the Domain2's controller return $PM_1, = DM' + LM' + LR' = 0.05 + 0.5 + 0.02 = 0.57$ to Domain 1, so Domain 1s gets the $PM_1 = DM_1 + LM_1 + LR_1 = 0.57 + 0.1 + 0.15 = 0.82$

Similarly, Domain3 returns $LM_d, = 0.01$ to Domain 4, and then Domain 4 gets the $DM'' = 0.01$ and then the Domain4's controller return $PM' = DM'' + LM'' + LR'' = 0.21$. to Domain 1, so Domain 1 gets P2's metric $PM_2 = DM_2 + LM_2 + LR_2 = 0.21 + 0.1 + 0.1 = 0.41$.

Since $PM_2 (0.41) < PM_1 (0.82)$, the new load balancing algorithm will choose P2 $\{D_1, D_4, D_3\}$ as the shortest path for traffic between h11 and h33 .

4. Implementation and Results

4.1 Background Technologies

The OpenDaylight controller is JVM software for SDN and can be run on any OS. It supports OpenFlow protocol with some useful tools, such as Karaf, Maven, OSGi, JAVA interfaces, REST APIs. ([4]) We select the OpenDayLight (ODL) platform in its Beryllium-SR4 as the base SDN controller of our scheme. The version has integrated SDNi which is used to implement the controllers' information exchange.

ODL-SDNi (Software Defined Networking Interface) is an application that is used to connect multiple Opendaylight-federated controllers in a network and sharing the Topology and QoS information among them. ([7][8])
Python is an interpreted and object-oriented programming language which has a large standard library to support string processing, various kinds of Internet protocols and operating system interfaces. Many SDN researchers use Python to develop their own SDN applications. ([6])

Mininet is a network emulator which can run virtual hosts, switches and controllers and links on a PC, VM workstation or cloud system and so on. Mininet supports arbitrary custom topologies, Python API, developing and testing Openflow applications and SDN systems. Mininet is very useful for development, teaching, and research on network field. ([9])

4.2 Environment

The complete network environment runs on the vCloud with the following resource: 32HZ Virtual CPU, 64GB Memory, 1TB Storage.

There are 8 VM machines, 192.168.118.129/133/135/140 are Mininet servers, other 4 VM machines (192.168.30.136/137/138/142) are OpenDayLight controllers. The relationship of these Mininet servers and ODL controllers is as follows:

1)Mininet (192.168.118.129) is connected to ODL (192.168.30.136)

2)Mininet (192.168.118.133) is connected to ODL (192.168.30.137)

3)Mininet (192.168.118.135) is connected to ODL (192.168.30.138)

4)Mininet (192.168.118.140) is connected to ODL (192.168.30.142)

Their detailed configuration is shown as the beloved table:

Table 5 System Configuration of SDNi Environment

VM Machine	IP Address	Software	Hardware
Mininet 1	192.168.118.129	OS: Ubuntu 14.04 Mininet 2.2.1	1 CPU, 1GB Memory, 8GB Storage
Mininet 2	192.168.118.130	OS: Ubuntu 14.04 Mininet 2.2.1	1 CPU, 1GB Memory, 8GB Storage
Mininet 3	192.168.118.135	OS: Ubuntu 14.04 Mininet 2.2.1	1 CPU, 1GB Memory, 8GB Storage
Mininet 4	192.168.118.140	OS: Ubuntu 14.04 Mininet 2.2.1	1 CPU, 1GB Memory, 8GB Storage
ODL Controller 1	192.168.30.136	OS: Ubuntu 16.04 64bits ODL Beryllium SR4	2 CPU, 8GB Memory, 32GB Storage
ODL Controller 2	192.168.30.137	OS: Ubuntu 16.04 64bits ODL Beryllium SR4	2 CPU, 8GB Memory, 32GB Storage
ODL Controller 3	192.168.30.138	OS: Ubuntu 16.04 64bits ODL Beryllium SR4	2 CPU, 8GB Memory, 32GB Storage
ODL Controller 4	192.168.30.142	OS: Ubuntu 16.04 64bits ODL Beryllium SR4	2 CPU, 8GB Memory, 32GB Storage

These servers' deploying is shown in Figure 11:

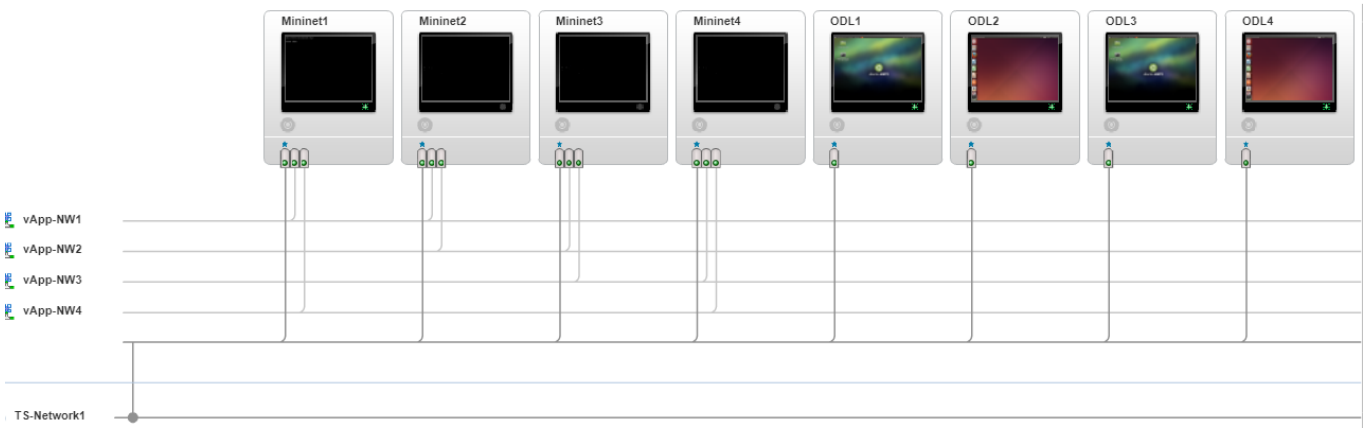

Figure 11 vCloud Deploying of the SDN Inter-domains

4.3 System Implementation

Following are the steps to implement the SDN inter-domain routing.

1) Capture and process inter-domain packets in L2-Switch module of the controller
2) Develop an application that provides an interface to the controller. The functions of the application include topology discovery based on SDNi, Inter-Domain shortest path computation and inter-domain flows installation along the shortest path;
3) Implement load-balancing function based on the distribution SDN inter-domain routing architecture.

4.3.1 Topology

We use the following python script to create the test network in each domain.

```
    "Create Domain1 Network"
    net = Mininet( controller=lambda a: RemoteController(a, ip='192.168.118.136' ))
    info( '*** Adding Controller\n' )
    net.addController( 'c1' )
    # Add hosts and switches
    info( '*** Creating Nodes\n' )
    Host11 = net.addHost( 'h11', mac='00:00:00:00:00:01', ip='10.1.1.1/24'  )
    Host12 = net.addHost( 'h12', mac='00:00:00:00:00:02', ip='10.1.1.2/24'  )
    Host13 = net.addHost( 'h13', mac='00:00:00:00:00:03', ip='10.1.1.3/24'  )
    Host14 =  net.addHost( 'h14', mac='00:00:00:00:00:04', ip='10.1.1.4/24'  )

    info( '*** Creating Switches\n' )
    Switch11 = net.addSwitch( 's11' )
    Switch12 = net.addSwitch( 's12' )
    Switch13 = net.addSwitch( 's13' )

    # Add links
    info( '*** Creating links\n' )
    net.addLink( Host11, Switch11 )
    net.addLink( Host14, Switch11 )
    net.addLink( Switch11, Switch12 )
    net.addLink( Switch12, Host12 )
```

```
    net.addLink( Switch12, Switch13 )
    net.addLink( Switch11, Switch13 )
    net.addLink( Switch13, Host13 )

    info( '*** Building network\n' )
    net.build()
```

Figure 12 is our test-bed topology for inter-domain communication and load balancing test. The red lines in the figures refer to the inter-domain links connecting the border switches among the domains; the blue lines refer to the local links connecting the local switches/hosts in each domain.

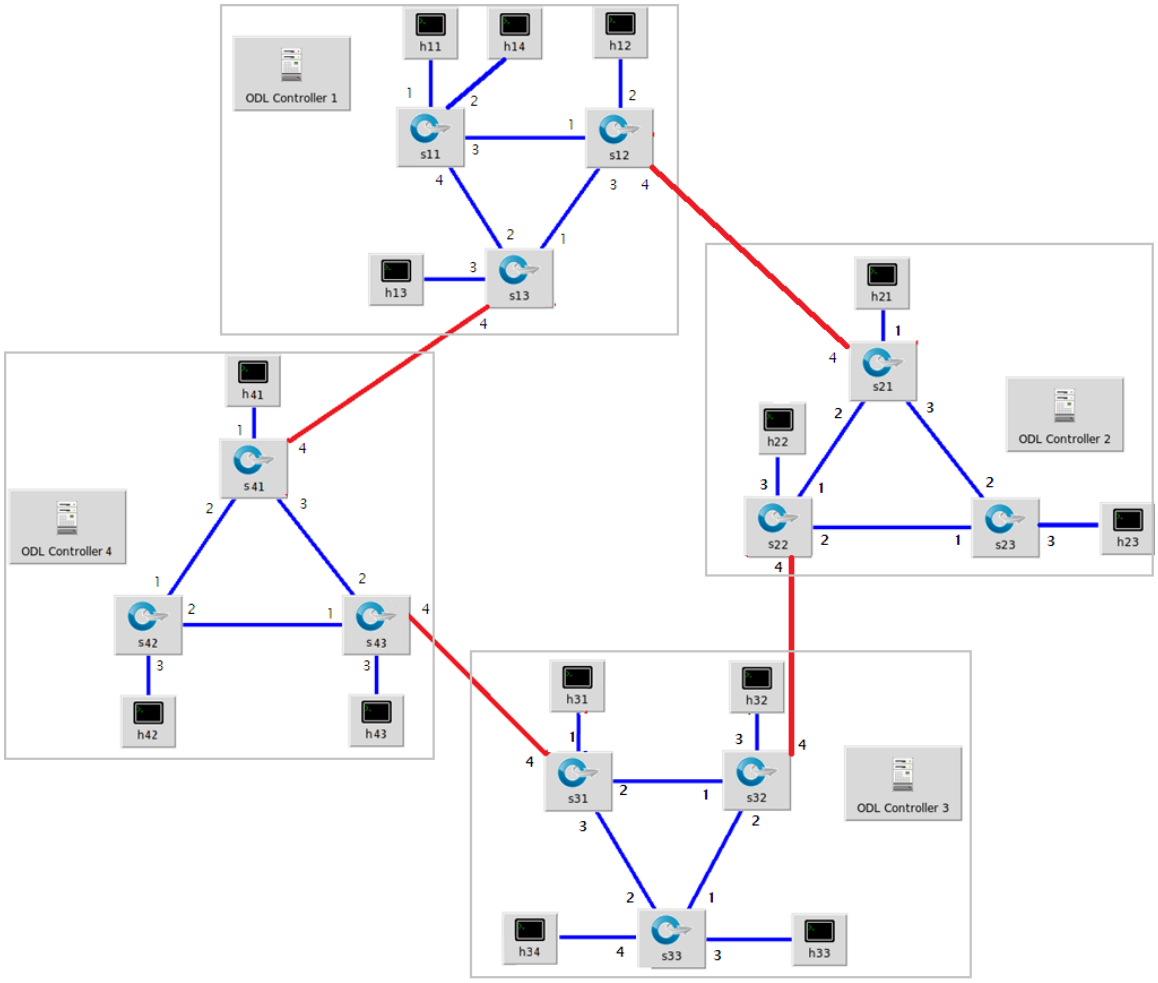

Figure 12 Topology for Inter-domain Communication and Load Balancing Test

4.3.2 OpenDayLight L2switch module modification

In ODL, L2Switch is an important basic network service, which provides Layer2 switch functionality. L2Switch module is comprised of the following Components:

1) Packet Handler: Decodes the packets coming to the controller and dispatches them appropriately. In this component, some decoder classes can decode Ethernet, ARP, IPv4 or IPv6 packet based their Ethertype respectively.
2) Loop Remover: This component removes loops in the network by writing STP (Spanning Tree Protocol) status of "forwarding" or "discarding" to each link in the Topology data tree. Forwarding links can forward packets and discarding links cannot forward packets. The component uses Link Layer Discovery Protocol (LLDP) to learn topology information, so it can change STP status of a link based on changes in the network.
3) Arp Handler: It handles and processes the controller's incoming ARP packets and send the ARP packet back into the network.
4) Address Tracker: This component learns the Addresses (MAC and IP) of packets in the network. its class AddressObserverUsingArp, AddressObserverUsingIpv4, AddressObserverUsingIpv6 registers for Arp, IPv4, IPv6 packet notification respectively.
5) Host Tracker: It tracks the locations of hosts (MAC addresses of the hosts as the primary ID) in the network and gathers information about the traffic flowing to a host.
6) L2Switch Main: It installs flows based on MAC addresses learned on each switch when packets come to the controller. If the destination of a coming packet is unknown, it will send a broadcast message in the network, otherwise, if the destination is known, L2Switch module will forward the packet to the destination.

In our approach, we change the Address Tracker component of L2Switch module to enable it to recognize the inter-domain packets. Firstly, as mentioned in section 3.4, we configure a static gateway in the routing table and ARP table of each host. The following python script is used to configure the static gateway (Mac address is "00:00:00:00:00:64") in each host of a domain.

```
# Congfigure dummy routers
info( '*** Congfigure dummy routers\n' )
Host11.cmd("sudo route add default gw 10.1.1.100 h11-eth0")
Host11.cmd("sudo arp -s 10.1.1.100  00:00:00:00:00:64")
Host12.cmd("sudo route add default gw 10.1.1.100 h12-eth0")
Host12.cmd("sudo arp -s 10.1.1.100  00:00:00:00:00:64")
Host13.cmd("sudo route add default gw 10.1.1.100 h13-eth0")
Host13.cmd("sudo arp -s 10.1.1.100  00:00:00:00:00:64")
Host14.cmd("sudo route add default gw 10.1.1.100 h14-eth0")
Host14.cmd("sudo arp -s 10.1.1.100  00:00:00:00:00:64")
```

If the destination of a packet is not in the current domain, the destination MAC address will be the default gateway MAC address. Thus, L2Switch can identify the inter-domain traffic based on this MAC address. We configure and change the address tracker component of L2Switch module

such that it can capture and process inter-domain packets when packet-in messages arrive at the controller. If the L2Switch component receives these inter-domain packets, it calls our python application (which will be represented in following section) for layer 3 forwarding. Packets destined inside the current domain will be handled by the default L2Switch layer 2 forwarding. The following function is the key java code in the modification of ODL L2Switch's Address Tracker component.

```
public void onIpv4PacketReceived(Ipv4PacketReceived packetReceived) {
  if(packetReceived == null || packetReceived.getPacketChain() == null) {
    return;
  }
  RawPacket rawPacket = null;
  EthernetPacket ethernetPacket = null;
  Ipv4Packet ipv4Packet = null;
  for(PacketChain packetChain : packetReceived.getPacketChain()) {
    if(packetChain.getPacket() instanceof RawPacket) {
      rawPacket = (RawPacket) packetChain.getPacket();
    } else if(packetChain.getPacket() instanceof EthernetPacket) {
      ethernetPacket = (EthernetPacket) packetChain.getPacket();
    } else if(packetChain.getPacket() instanceof Ipv4Packet) {
      ipv4Packet = (Ipv4Packet) packetChain.getPacket();
    }
  }
  if(rawPacket == null || ethernetPacket == null || ipv4Packet == null) {
    return;
  }
  if(!IPV4_IP_TO_IGNORE.equals(ipv4Packet.getSourceIpv4().getValue())) {
    addressObservationWriter.addAddress(ethernetPacket.getSourceMac(),
        new IpAddress(ipv4Packet.getSourceIpv4().getValue().toCharArray()),
        rawPacket.getIngress());
    MacAddress destMac = ethernetPacket.getDestinationMac();
    if(!ignoreThisMac(destMac)) {
      String checkDummyMac = destMac.getValue();
      if ( checkDummyMac.equals("00:00:00:00:00:64") ) {
          Runtime run = Runtime.getRuntime();
          String command = "python /home/user/multi_domain_router.py";
          String line = null;
          Process pr = run.exec(command);
          pr.waitFor();
          BufferedReader buf = new BufferedReader(new InputStreamReader(pr.getInputStream()));
          while ((line=buf.readLine())!=null) {
             _logger.debug("Run Python Router Application Info : {}", line);
          }
      }
    }
  }
}
```

4.3.3 Topology Discovery Based on SDNi

By setting up SDNi, each domain controller can obtain the domain topology information of the other domains by communicating with respective domain controllers. After obtaining the information topology of the inter-domain network, each domain controller can independently build a domain level topology and identify the border switches and the corresponding external links. The domain-level topology will be used to determine the shortest path between source and destination domains.

4.3.4 Inter-Domain Shortest Path

We can get the domain level topology from section 4.3.3 and form the domain level network graph. In the graph, domain IDs as nodes are inserted as graph vertices and the external links are inserted as the graph edges. We created the dictionary variables (subIPDomains and domainSubIPs) for mapping the different domains and their network address. Based on the domain-level network graph, we use the python *NetworkX* package to compute the inter-domain shortest path as the following code:

```
# h1: source host's ip address; h2: destination host's ip address
srcSubnet = IP(h1).make_net('255.255.255.0').strNormal()
srcDomain = subIPDomains[srcSubnet]
dstSubnet = IP(h2).make_net('255.255.255.0').strNormal()
dstDomain = subIPDomains[dstSubnet]
dmShorted_path = nx.dijkstra_path(domainGraph, source = srcDomain, target = dstDomain)
```

For a transit domain controller, it can use the following code to get the domain level shortest path between the current transit domain to the destination domain:

```
curDomain = ipDomains[curDomainIP]
dstSubnet = IP(h2).make_net('255.255.255.0').strNormal()
dstDomain = subIPDomains[dstSubnet]
dmShorted_path = nx.dijkstra_path(domainGraph, source = curDomain, target = dstDomain)
```

4.3.5 Inter-Domain Flows along the Shortest Path

The following *pushFlowRules* function illustrates our method to push the inter-domain flows along the domain level shortest path.

```
# h1: source ip address; h2: destination ip address
def pushFlowRules(dmshortestPath,h1,h2):
    global globalFlowId
    flowNum = 100
    curDomain = ipDomains[curDomainIP]
    localSubnet = domainSubIPs[curDomain]
    bottom = len(dmshortestPath)
    for currentNode in range(0,bottom):
        if curDomain == dmshortestPath[currentNode]
            print "Intra-domain shortest path : "
            print dmshortestPath
            if currentNode == 0:
                borderLink = borderDomainLink[dmshortestPath[1]]
                portnumlst = linkPorts[borderLink]
                portNum = portnumlst.split("::")[1]
                borderSwitch = borderLink.split("::")[1]
                bSwID = borderSwitch.split(":")[1]
                swNode = 's' + bSwID
                flowName = h1 + 'output_to' + h2
                globalFlowId = globalFlowId + 1
                flowURL =  http_url + curDomainIP + ':8181/restconf/config/opendaylight-
inventory:nodes/node/'+ borderSwitch + '/table/0/flow/' + str(globalFlowId)
                 jsonBody = makeFlowInstr(globalFlowId, flowName, portNum, dummyRouterMac,h1,h2)
                print flowURL
                print jsonBody
                response = requests.put(flowURL, auth=auth, data = jsonBody, headers = headers)
                print response

            elif currentNode == bottom - 1:
```

```
                borderLink = borderDomainLink[dmshortestPath[bottom - 2]]
                portnumlst = linkPorts[borderLink]
                portNum = portnumlst.split("::")[1]
                borderSwitch = borderLink.split("::")[1]
                bSwID = borderSwitch.split(":")[1]
                swNode = 's' + bSwID
                flowName = h1 + 'output_to' + h2
                globalFlowId = globalFlowId + 1
                flowURL =  http_url + curDomainIP + ':8181/restconf/config/opendaylight-
inventory:nodes/node/'+ borderSwitch + '/table/0/flow/' + str(globalFlowId)
                jsonBody = makeFlowInstr(globalFlowId, flowName, portNum, dummyRouterMac,h2,h1)
                response = requests.put(flowURL, auth=auth, data = jsonBody, headers = headers)
                print response
            else :
                borderLink = borderDomainLink[dmshortestPath[currentNode+1]]
                portnumlst = linkPorts[borderLink]
                portNum = portnumlst.split("::")[1]
                borderSwitch = borderLink.split("::")[1]
                bSwID = borderSwitch.split(":")[1]
                swNode = 's' + bSwID
                flowName = h1 + 'output_to' + h2
                globalFlowId = globalFlowId + 1
                flowURL =  http_url + curDomainIP + ':8181/restconf/config/opendaylight-
inventory:nodes/node/'+ borderSwitch + '/table/0/flow/' + str(globalFlowId)
                jsonBody = makeFlowInstr(globalFlowId, flowName, portNum, dummyRouterMac,h1,h2)
                response = requests.put(flowURL, auth=auth, data = jsonBody, headers = headers)
                print response
                #push reversed flow
                borderLink = borderDomainLink[dmshortestPath[currentNode-1]]
                portnumlst = linkPorts[borderLink]
                portNum = portnumlst.split("::")[1]
                borderSwitch = borderLink.split("::")[1]
                bSwID = borderSwitch.split(":")[1]
                swNode = 's' + bSwID
                flowName = h2 + 'output_to' + h1
                globalFlowId = globalFlowId + 1
                flowURL =  http_url + curDomainIP + ':8181/restconf/config/opendaylight-
inventory:nodes/node/'+ borderSwitch + '/table/0/flow/' + str(globalFlowId)
                jsonBody = makeFlowInstr(globalFlowId, flowName, portNum, dummyRouterMac,h2,h1)
                response = requests.put(flowURL, auth=auth, data = jsonBody, headers = headers)
                print response
```

In the function, *dmshortestPath* is the domain level shortest path which is obtained from section 4.3.3. If *currentNode* is the first node of the domain level shortest path, it is the source domain, and the source domain controller identifies the border switch and port to downstream peer domain and then push the forwarding flow; if *currentNode* is the last node of the domain level shortest path, it is the destination domain, and the destination domain controller identifies the border switch and port to upstream peer domain and then push the reversed flow; otherwise, *currentNode* is transit domain, and its controller identifies the border switches and ports that are linked to downstream and upstream domain, and then push the forwarding and reserved flows into the border switches.

4.3.5 Load Balancing base on Distributed Inter-domain SDN

We use python flask framework to create the RESTful API web services in each domain. These REST APIs provides the domain's load metric calculation (defined by section 3.5) and calls the downstream domain's load metric API if the current domain is not destination domain. When the source domain controller gets all the results from all downstream domains' APIs in equal-cost shortest paths, the controller chooses the path with the minimum load metric and pushes the inter-domain flows along the optimal path using the function of section 4.3.4. The following codes depict how to implement the load balancing based on our distributed inter-domain SDN.

```
    mLst = getcurrentDomainLoadMetric(PeerDomains,borderSws)
    LoadMetricList = []
    for peerDomainIP in PeerDomains:
        odl_url = http_url + peerDomainIP + loadmetric_url
        response = requests.get(odl_url, auth=auth)
        LoadMetric = json.loads(response.content)
    LoadMetricList.append(LoadMetric)
    minMetricPathID = get_min_path(LoadMetricList, mLst)
    dmshortestPath = shortestPaths[minMetricPathID]
    pushFlowRules(dmshortestPath,h1,h2)
```

4.2 Result Analysis

4.2.1 Inter-domain Communication Test

We test the inter-domain communication by pinging from h11 of domain 1 to h23 in domain 2 and h31 in domain 3 in the Figure 12's topology. The ping process was found successful (Figure 13). After the ping, we checked the border switches' flow tables. Figure 14 shows that the application can identify border switches between domains and push the proper flows to those switches. The flow table entries that are underlined are the flows installed by the controller in the border switch used for inter-domain routing. The underlined entries show that the inter-domain packets will be forwarded along the inter-domain/intra-domain shortest path.

```
mininet> h11 ping 10.1.2.3
PING 10.1.2.3 (10.1.2.3) 56(84) bytes of data.
64 bytes from 10.1.2.3: icmp_seq=1 ttl=64 time=1.27 ms
64 bytes from 10.1.2.3: icmp_seq=2 ttl=64 time=0.500 ms
64 bytes from 10.1.2.3: icmp_seq=3 ttl=64 time=0.441 ms
64 bytes from 10.1.2.3: icmp_seq=4 ttl=64 time=0.584 ms
64 bytes from 10.1.2.3: icmp_seq=5 ttl=64 time=0.455 ms
64 bytes from 10.1.2.3: icmp_seq=6 ttl=64 time=0.519 ms
64 bytes from 10.1.2.3: icmp_seq=7 ttl=64 time=0.598 ms
^C
--- 10.1.2.3 ping statistics ---
7 packets transmitted, 7 received, 0% packet loss, time 6002ms
rtt min/avg/max/mdev = 0.441/0.623/1.270/0.271 ms
mininet> h11 ping 10.1.3.1
PING 10.1.3.1 (10.1.3.1) 56(84) bytes of data.
64 bytes from 10.1.3.1: icmp_seq=1 ttl=64 time=2.06 ms
64 bytes from 10.1.3.1: icmp_seq=2 ttl=64 time=0.927 ms
64 bytes from 10.1.3.1: icmp_seq=3 ttl=64 time=1.15 ms
64 bytes from 10.1.3.1: icmp_seq=4 ttl=64 time=1.11 ms
64 bytes from 10.1.3.1: icmp_seq=5 ttl=64 time=0.987 ms
64 bytes from 10.1.3.1: icmp_seq=6 ttl=64 time=1.05 ms
64 bytes from 10.1.3.1: icmp_seq=7 ttl=64 time=0.993 ms
^C
--- 10.1.3.1 ping statistics ---
7 packets transmitted, 7 received, 0% packet loss, time 6013ms
rtt min/avg/max/mdev = 0.927/1.185/2.063/0.366 ms
mininet>
```

Figure 13 Ping Result After Running Program

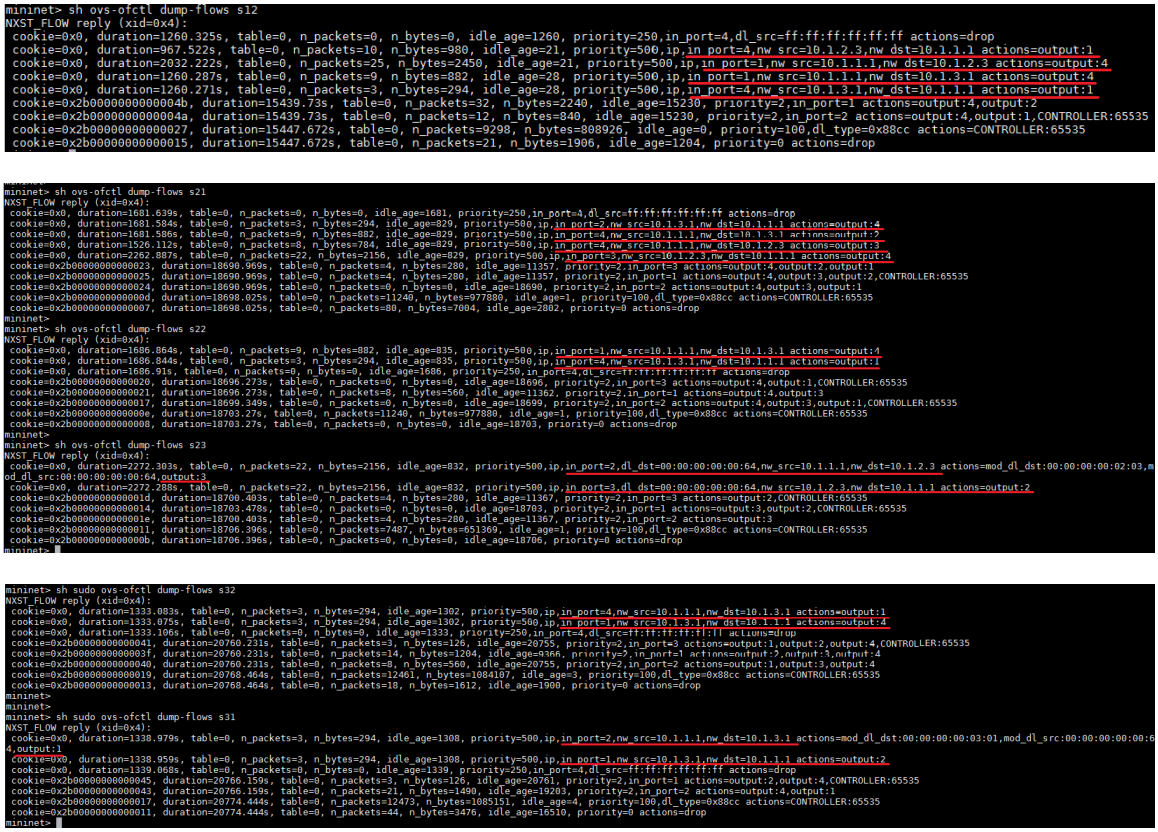

Figure 14 Flow Table in Related Switches Pushed by Application

4.2.2 Load Balancing Test

After integrating the load balancing application to all the domain controllers, we test the new load balancing method and compare it with Round Robin load balancing strategy. Firstly, we use iperf to generate large traffic in the transit domain - Domain 4 and make the domain congested. We issue a sequence of pings according to the following order:

1) From h11 to h33
2) From h11 to h34

Based on the topology shown in Figure 12, the equal-cost domain-level shortest path from Domain1 to Domain 3 are found as follows:

1) Domain 1-> Domain 2->Domain 3

2) Domain 1-> Domain 4->Domain 3

We tested Round Robin load balancing strategy and get the flow table of the related switches as Figure 15, so we can get the paths which are allocated to these hosts according to Round Robin as the following Table 6 and Figure16.

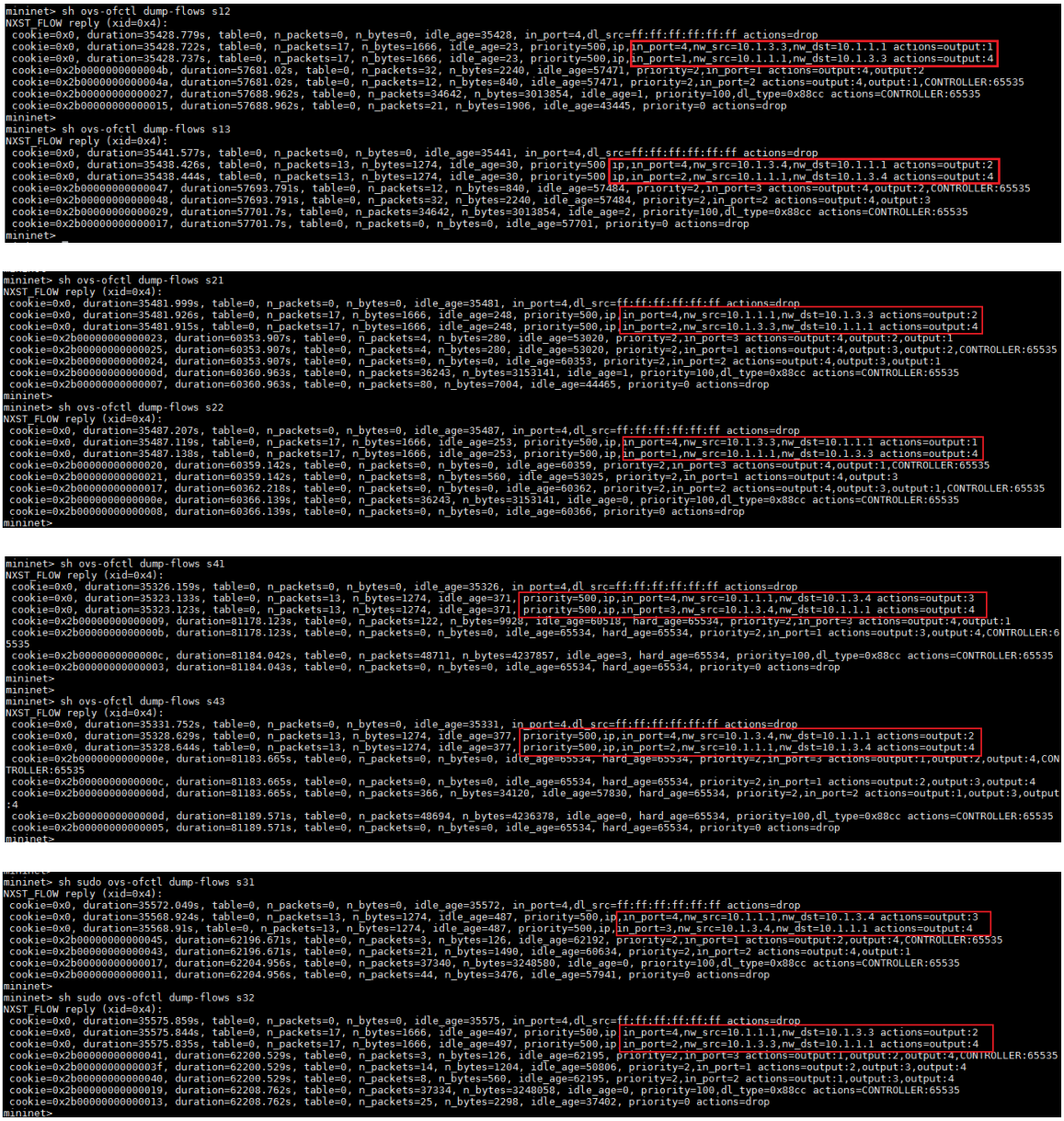

Figure 15 Flow Table of Related Switches in Round Robin Load Balancing Strategy

Table 6 Path of Round Robin Load Balancing Strategy

Source	Destination	Path
h11	h33	Domain 1-> Domain 2->Domain 3
h11	h34	Domain 1-> Domain 4->Domain 3

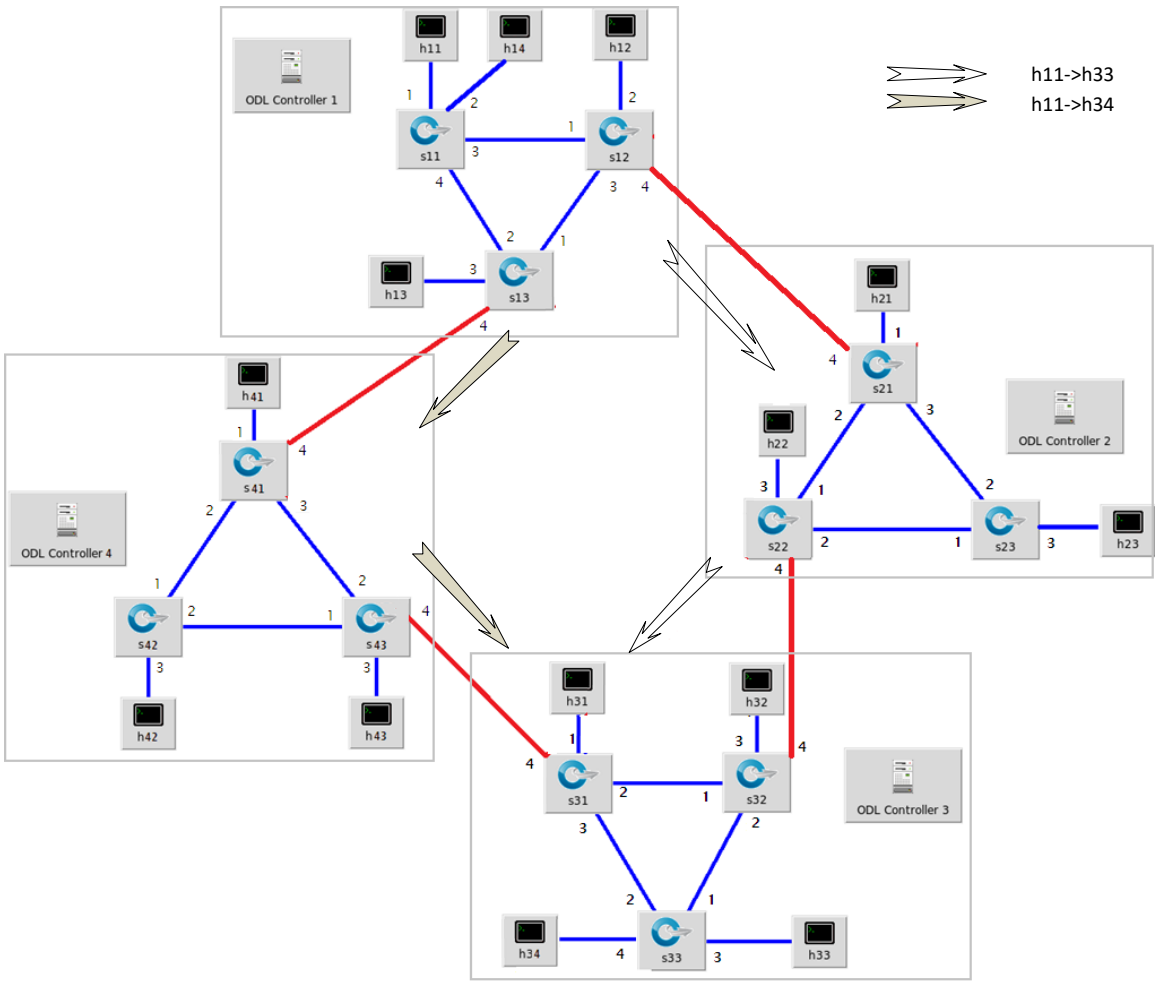

Figure 16 Round Robin Paths between Source/Destination Host Pairs

Round Robin load balancing method does not consider traffic condition, so it won't avoid the congested domain.

After we completed the all ping operation using our new load balancing application. We check the flow table on each border switches, and then we got the following real paths between these source/ destination host pairs.

```
mininet> sh ovs-ofctl dump-flows s12
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=985.246s, table=0, n_packets=71, n_bytes=6958, idle_age=218, priority=500,ip,in_port=4,nw_src=10.1.3.3,nw_dst=10.1.1.1 actions=output:1
 cookie=0x0, duration=910.53s, table=0, n_packets=80, n_bytes=7840, idle_age=215, priority=500,ip,in_port=1,nw_src=10.1.1.1,nw_dst=10.1.3.4 actions=output:4
 cookie=0x0, duration=910.521s, table=0, n_packets=80, n_bytes=7840, idle_age=215, priority=500,ip,in_port=4,nw_src=10.1.3.4,nw_dst=10.1.1.1 actions=output:1
 cookie=0x0, duration=985.28s, table=0, n_packets=71, n_bytes=6958, idle_age=218, priority=500,ip,in_port=1,nw_src=10.1.1.1,nw_dst=10.1.3.3 actions=output:4
 cookie=0x0, duration=985.324s, table=0, n_packets=0, n_bytes=0, idle_age=985, in_port=4,dl_src=ff:ff:ff:ff:ff:ff actions=drop
 cookie=0x2b0000000000004b, duration=78967.155s, table=0, n_packets=10722, n_bytes=16174316, idle_age=9052, hard_age=65534, priority=2,in_port=1 actions=output:4,output:2
 cookie=0x2b0000000000004a, duration=78967.155s, table=0, n_packets=5608, n_bytes=370192, idle_age=9052, hard_age=65534, priority=2,in_port=2 actions=output:4,output:1,CONTR
OLLER:65535
 cookie=0x2b00000000000027, duration=78975.097s, table=0, n_packets=47422, n_bytes=4125714, idle_age=1, hard_age=65534, priority=100,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x2b00000000000015, duration=78975.097s, table=0, n_packets=21, n_bytes=1906, idle_age=64732, hard_age=65534, priority=0 actions=drop
mininet>
mininet> sh ovs-ofctl dump-flows s13
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=988.321s, table=0, n_packets=0, n_bytes=0, idle_age=988, in_port=4,dl_src=ff:ff:ff:ff:ff:ff actions=drop
 cookie=0x2b0000000000047, duration=78970.107s, table=0, n_packets=12, n_bytes=840, idle_age=65534, hard_age=65534, priority=2,in_port=3 actions=output:4,output:2,CONTROLLE
R:65535
 cookie=0x2b00000000000048, duration=78970.107s, table=0, n_packets=16318, n_bytes=16543668, idle_age=9055, hard_age=65534, priority=2,in_port=2 actions=output:4,output:3
 cookie=0x2b00000000000029, duration=78978.016s, table=0, n_packets=47412, n_bytes=4124844, idle_age=0, hard_age=65534, priority=100,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x2b00000000000017, duration=78978.016s, table=0, n_packets=656329, n_bytes=627383136, idle_age=212, hard_age=65534, priority=0 actions=drop
mininet>
```

```
mininet> sh ovs-ofctl dump-flows s21
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=963.066s, table=0, n_packets=80, n_bytes=7840, idle_age=350, priority=500,ip,in_port=2,nw_src=10.1.3.4,nw_dst=10.1.1.1 actions=output:4
 cookie=0x0, duration=963.108s, table=0, n_packets=80, n_bytes=7840, idle_age=350, priority=500,ip,in_port=4,nw_src=10.1.1.1,nw_dst=10.1.3.4 actions=output:2
 cookie=0x0, duration=1005.79s, table=0, n_packets=71, n_bytes=6958, idle_age=352, priority=500,ip,in_port=4,nw_src=10.1.1.1,nw_dst=10.1.3.3 actions=output:2
 cookie=0x0, duration=1005.783s, table=0, n_packets=71, n_bytes=6958, idle_age=352, priority=500,ip,in_port=2,nw_src=10.1.3.3,nw_dst=10.1.1.1 actions=output:4
 cookie=0x0, duration=1005.86s, table=0, n_packets=0, n_bytes=0, idle_age=1005, in_port=4,dl_src=ff:ff:ff:ff:ff:ff actions=drop
 cookie=0x2b0000000000023, duration=81549.617s, table=0, n_packets=4, n_bytes=280, idle_age=65534, hard_age=65534, priority=2,in_port=3 actions=output:4,output:2,output:1
 cookie=0x2b0000000000025, duration=81549.617s, table=0, n_packets=4, n_bytes=280, idle_age=65534, hard_age=65534, priority=2,in_port=1 actions=output:4,output:3,output:2,C
ONTROLLER:65535
 cookie=0x2b0000000000024, duration=81549.617s, table=0, n_packets=0, n_bytes=0, idle_age=65534, hard_age=65534, priority=2,in_port=2 actions=output:4,output:3,output:1
 cookie=0x2b000000000000d, duration=81556.673s, table=0, n_packets=48962, n_bytes=4259694, idle_age=0, hard_age=65534, priority=100,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x2b00000000000007, duration=81556.673s, table=0, n_packets=11663, n_bytes=16238034, idle_age=9186, hard_age=65534, priority=0 actions=drop
mininet>
mininet> sh ovs-ofctl dump-flows s22
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=1010.612s, table=0, n_packets=71, n_bytes=6958, idle_age=357, priority=500,ip,in_port=4,nw_src=10.1.3.3,nw_dst=10.1.1.1 actions=output:1
 cookie=0x0, duration=967.91s, table=0, n_packets=80, n_bytes=7840, idle_age=354, priority=500,ip,in_port=1,nw_src=10.1.1.1,nw_dst=10.1.3.4 actions=output:4
 cookie=0x0, duration=967.899s, table=0, n_packets=80, n_bytes=7840, idle_age=354, priority=500,ip,in_port=4,nw_src=10.1.3.4,nw_dst=10.1.1.1 actions=output:1
 cookie=0x0, duration=1010.625s, table=0, n_packets=71, n_bytes=6958, idle_age=357, priority=500,ip,in_port=1,nw_src=10.1.1.1,nw_dst=10.1.3.3 actions=output:4
 cookie=0x0, duration=1010.667s, table=0, n_packets=0, n_bytes=0, idle_age=1010, in_port=4,dl_src=ff:ff:ff:ff:ff:ff actions=drop
 cookie=0x2b0000000000020, duration=81554.474s, table=0, n_packets=0, n_bytes=0, idle_age=65534, hard_age=65534, priority=2,in_port=3 actions=output:4,output:1,CONTROLLER:6
5535
 cookie=0x2b0000000000021, duration=81554.474s, table=0, n_packets=8, n_bytes=560, idle_age=65534, hard_age=65534, priority=2,in_port=1 actions=output:4,output:3
 cookie=0x2b0000000000017, duration=81557.55s, table=0, n_packets=0, n_bytes=0, idle_age=65534, hard_age=65534, priority=2,in_port=2 actions=output:4,output:3,output:1,CONT
ROLLER:65535
 cookie=0x2b000000000000e, duration=81561.471s, table=0, n_packets=48964, n_bytes=4259868, idle_age=1, hard_age=65534, priority=100,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x2b00000000000008, duration=81561.471s, table=0, n_packets=0, n_bytes=0, idle_age=65534, hard_age=65534, priority=0 actions=drop
mininet>
```

```
mininet> sh sudo ovs-ofctl dump-flows s31
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=1069.868s, table=0, n_packets=0, n_bytes=0, idle_age=1069, in_port=4,dl_src=ff:ff:ff:ff:ff:ff actions=drop
 cookie=0x2b0000000000045, duration=83295.352s, table=0, n_packets=3, n_bytes=126, idle_age=65534, hard_age=65534, priority=2,in_port=1 actions=output:2,output:4,CONTROLLER
:65535
 cookie=0x2b0000000000043, duration=83295.352s, table=0, n_packets=21, n_bytes=1490, idle_age=65534, hard_age=65534, priority=2,in_port=2 actions=output:4,output:1
 cookie=0x2b0000000000017, duration=83303.637s, table=0, n_packets=49999, n_bytes=4349913, idle_age=4, hard_age=65534, priority=100,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x2b0000000000011, duration=83303.637s, table=0, n_packets=658836, n_bytes=630121526, idle_age=491, hard_age=65534, priority=0 actions=drop
mininet> sh sudo ovs-ofctl dump-flows s32
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=1034.307s, table=0, n_packets=80, n_bytes=7840, idle_age=500, priority=500,ip,in_port=2,nw_src=10.1.3.4,nw_dst=10.1.1.1 actions=output:4
 cookie=0x0, duration=1034.329s, table=0, n_packets=80, n_bytes=7840, idle_age=500, priority=500,ip,in_port=4,nw_src=10.1.1.1,nw_dst=10.1.3.4 actions=output:2
 cookie=0x0, duration=1072.235s, table=0, n_packets=71, n_bytes=6958, idle_age=502, priority=500,ip,in_port=4,nw_src=10.1.1.1,nw_dst=10.1.3.3 actions=output:2
 cookie=0x0, duration=1072.218s, table=0, n_packets=71, n_bytes=6958, idle_age=502, priority=500,ip,in_port=2,nw_src=10.1.3.3,nw_dst=10.1.1.1 actions=output:4
 cookie=0x0, duration=1072.246s, table=0, n_packets=0, n_bytes=0, idle_age=1072, in_port=4,dl_src=ff:ff:ff:ff:ff:ff actions=drop
 cookie=0x2b0000000000041, duration=83297.766s, table=0, n_packets=3, n_bytes=126, idle_age=65534, hard_age=65534, priority=2,in_port=3 actions=output:1,output:2,output:4,C
ONTROLLER:65535
 cookie=0x2b000000000003f, duration=83297.766s, table=0, n_packets=14, n_bytes=1204, idle_age=65534, hard_age=65534, priority=2,in_port=1 actions=output:2,output:3,output:4
 cookie=0x2b0000000000040, duration=83297.766s, table=0, n_packets=8, n_bytes=560, idle_age=65534, hard_age=65534, priority=2,in_port=2 actions=output:1,output:3,output:4
 cookie=0x2b0000000000019, duration=83305.999s, table=0, n_packets=49995, n_bytes=4349565, idle_age=2, hard_age=65534, priority=100,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x2b00000000000013, duration=83305.999s, table=0, n_packets=25, n_bytes=2298, idle_age=58499, hard_age=65534, priority=0 actions=drop
mininet>
```

```
mininet> sh ovs-ofctl dump-flows s41
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=1095.763s, table=0, n_packets=0, n_bytes=0, idle_age=1095, in_port=4,dl_src=ff:ff:ff:ff:ff:ff actions=drop
 cookie=0x2b0000000000009, duration=102465.136s, table=0, n_packets=383472, n_bytes=569923244, idle_age=563, hard_age=65534, priority=2,in_port=3 actions=output:4,output:1
 cookie=0x2b000000000000b, duration=102465.136s, table=0, n_packets=279330, n_bytes=57886406, idle_age=563, hard_age=65534, priority=2,in_port=1 actions=output:3,output:4,C
ONTROLLER:65535
 cookie=0x2b000000000000c, duration=102471.055s, table=0, n_packets=61473, n_bytes=5348151, idle_age=0, hard_age=65534, priority=100,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x2b00000000000003, duration=102471.056s, table=0, n_packets=11589, n_bytes=16231426, idle_age=9406, hard_age=65534, priority=0 actions=drop
mininet> sh ovs-ofctl dump-flows s43
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=1099.323s, table=0, n_packets=0, n_bytes=0, idle_age=1099, in_port=4,dl_src=ff:ff:ff:ff:ff:ff actions=drop
 cookie=0x2b000000000000e, duration=102468.65s, table=0, n_packets=325330, n_bytes=482231964, idle_age=567, hard_age=65534, priority=2,in_port=3 actions=output:1,output:2,o
utput:4,CONTROLLER:65535
 cookie=0x2b000000000000c, duration=102468.65s, table=0, n_packets=70634, n_bytes=104872252, idle_age=7477, hard_age=65534, priority=2,in_port=1 actions=output:2,output:3,o
utput:4
 cookie=0x2b000000000000d, duration=102468.65s, table=0, n_packets=270162, n_bytes=43505118, idle_age=567, hard_age=65534, priority=2,in_port=2 actions=output:1,output:3,ou
tput:4
 cookie=0x2b000000000000d, duration=102474.556s, table=0, n_packets=61462, n_bytes=5347194, idle_age=0, hard_age=65534, priority=100,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x2b00000000000005, duration=102474.556s, table=0, n_packets=0, n_bytes=0, idle_age=65534, hard_age=65534, priority=0 actions=drop
mininet>
```

Figure 17 Flow Tables on Border Switches in New Load Balancing Method

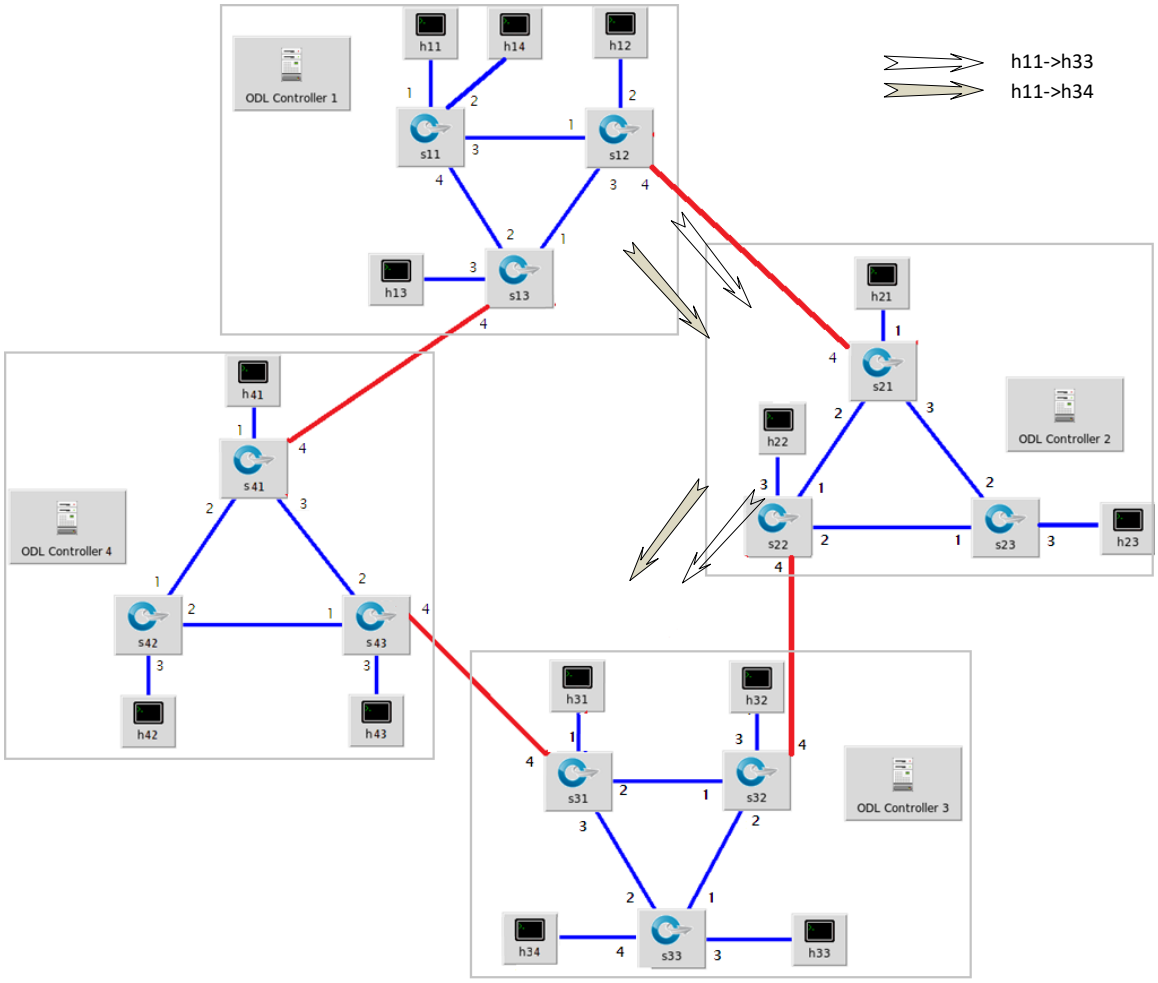

Figure 18 Real Paths between Source/Destination Host Pairs

The test result is consistent with our expectations.

4.2.3 Performance Comparison

We used iperf to simulate network congestion for 20 seconds in Domain 4 as Figure 18.

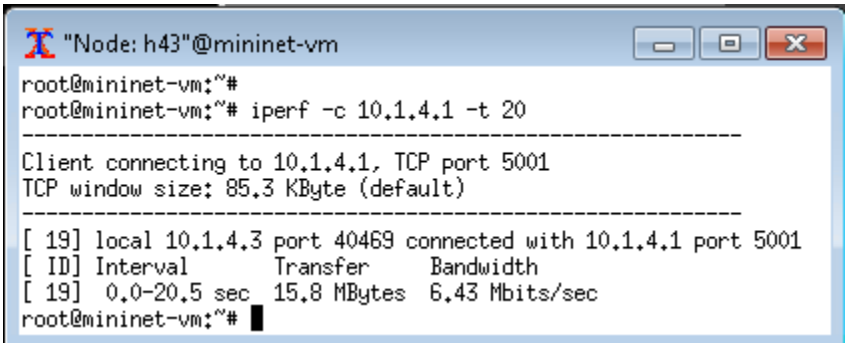

Figure 19 Iperf Simulating Network Congestion

When Iperf was simulating the congestion, we tested ping h33 & h34 from h11 respectively. As there is a congested domain at the path from h11 to h34 in Round Robin method, the ping (from h11 to h34) is slow and spent more time with the accumulation of network traffic. It also has 9% packet loss. However, another ping (from h11 to h33) is fast and no packet loss as there is no congested domain along its path.

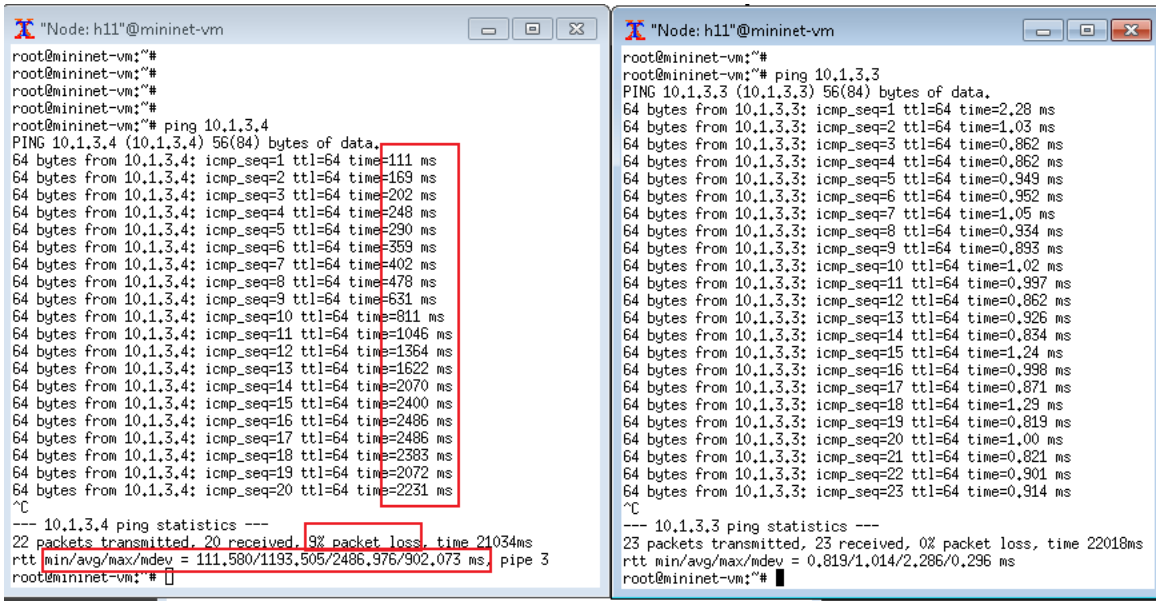

Figure 20 Performance of Round Robin Method

For the same scenario, our load balancing method got better performance than Round Robin method as Figure 20 because the new method can detect the network traffic congestion and choose an optimized path.

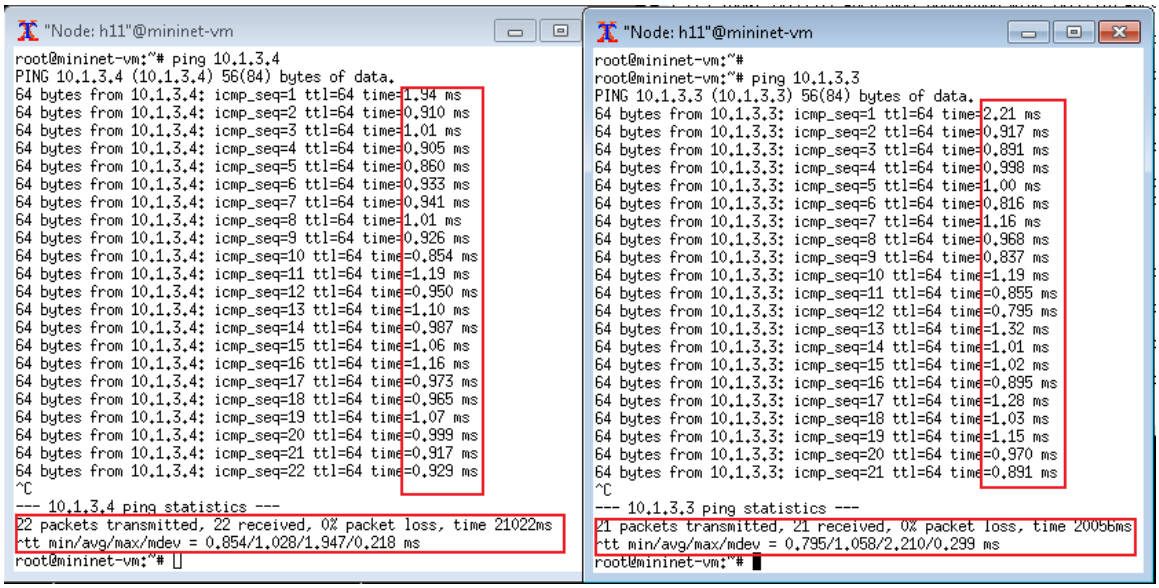

Figure 21 Performance of Proposed Load Balancing Method

5. Conclusion and Future work

We designed and implemented a distributed inter-domain routing mechanism in the SDN environment. The implementation was found to be successful. In addition, we proposed, designed and implemented an efficient load balancing scheme, which performs load balancing based on the traffic load of the network. Again, we found that the implementation was successful, and the controller can select a better path to avoid the congested part of the network. We emphasize that the OpenDayLight controller can be deployed in the real network. For that reason, our implementation can be easily tested and deployed in the production environment.

In the future, we will continue to improve this application to deal with more complicated topology. We will also combine the segment routing module into our approach to expedite the recovery for some link failure.

References

[1] S. Civanlar, E. Lokman, B. Kaytaz, A. Murat Tekalp, "Distributed management of service-enabled flow-paths across multiple sdn domains", *IEEE European Conf. on Networks and Communications (EuCNC)*, pp. 360-364, 2015.

[2] Benzekki Kamal et al. Software-defined networking (SDN): a survey. Security and Communication Networks 9, no. 18 (2016): 5803-5833.

[3] "White Papers". Opennetworking.org. Retrieved 26 January 2017.

[4] "OpenDaylight - An Open Source Community and Meritocracy for Software-Defined Networking". https://www.opendaylight.org. Retrieved 7 February 2017.

[5] OpenFlow: Enabling Innovation in Campus Networks, ACM SIGCOMM Computer Communication Review Volume 38 Issue 2, April 2008 Pages 69-74.

[6] "The Python Tutorial". www.python.org/. Retrieved 17 January 2017.

[7] https://wiki.opendaylight.org/view/ODL-SDNiApp:User_Guide_for_Beryllium_Release. Retrieved 7 May 2017.

[8] https://wiki.opendaylight.org/view/ODL-SDNi_App:Main. Retrieved 7 May 2017.

[9] Bob Lantz, Brandon Heller, and Nick McKeown. A Network in a Laptop: Rapid Prototyping for Software-Defined Networks. 9th ACM Workshop on Hot Topics in Networks, October 20-21, 2010, Monterey, CA.

[10] Tootoonchian A, Ganjali Y. HyperFlow: a distributed control plane for OpenFlow. Proceedings of the 2010 Internet Network Management Conference on Research on Enterprise Networking. USENIX Association,2010, p. 3.

[11] Teemu Koponen , Martin Casado , Natasha Gude , Jeremy Stribling , Leon Poutievski , Min Zhu , Rajiv Ramanathan , Yuichiro Iwata , Hiroaki Inoue , Takayuki Hama , Scott Shenker, Onix: a distributed control platform for large-scale production networks, Proceedings of the 9th USENIX conference on Operating systems design and implementation, p.1-6, October 04-06, 2010, Vancouver, BC, Canada

[12] S. Hassas Yeganeh, Y. Ganjali, "Kandoo: A Framework for Efficient and Scalable Offloading of Control Applications", *Proc. HotSON '12 Wksp.*, pp. 19-24, 2012.

[13] Ashvanth Kumar Selvakumaran. Software-Defined Inter-Domain Switching. Ryerson University, MASc. Thesis, 2016

[14] M. F. Ramdhani, S. N. Hertiana, and B. Dirgantara, "Multipath routing with load balancing and admission control in softwaredefined networking (SDN)," in Proc. 4th Int. Conf. Inf. Commun. Technol. (ICoICT), Bandung, Indonesia, 2016, pp. 1–6.

[15] Tim Huang. Path Computation Enhancement in SDN Networks. Ryerson University, MASc. Thesis, 2015.

[16] Y. L. Lan, K. Wang and Y. H. Hsu, "Dynamic load-balanced path optimization in SDN-based data center networks," 2016 10th International Symposium on Communication Systems, Networks and Digital Signal Processing (CSNDSP), Prague, 2016, pp. 1-6.

[17] P. Amaral, P. F. Pinto, L. Bernardo and F. Silva, "SDN based traffic engineering without optimization: A centrality based approach," 2017 IEEE International Conference on Communications (ICC), Paris, 2017, pp. 1-7.

[18] IEEE Standard 802.1AB Station and Media Access Control Connectivity Discovery, IEEE Computer Society, 28 June 2005.