Software-Defined Inter-domain Switching

by

Ashvanth Kumar Selvakumaran

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, 2016

AUTHOR’S DECLARATION FOR ELECTRONIC SUBMISSION OF A THESIS

Contents

List of Figures

List of Tables

Software-defined inter-domain switching

Master of Applied Science 2016

Ashvanth Kumar Selvakumaran

Computer Networks

Ryerson University

Abstract

Software Defined Networking (SDN) technology has garnered much attention in the field of networking. Even though there have been several SDN based data centre (domain) implementations, there is a need to inter-connect multiple SDN domains. In this thesis we will focus on enabling inter-domain layer 2 switching. We propose an approach wherein a central controller is responsible for inter-domain switching while the domain controllers are responsible for intra-domain switching in their respective domains. To achieve this, the central controller initially communicates with the domain controllers to gather the overall topology of the network. From the overall topology, the central controller can derive the domain-level topology, compute the domain-level spanning tree and install the tree on the topology. In addition, the central controller also computes the inter-domain shortest path between any pair of domains. The shortest path information are then pushed to the domain controllers in order to setup the network-wide shortest path. We demonstrate the viability of the proposed approach by implementing it in OpenDayLight, a popular SDN platform. To further demonstrate the flexibility and openness of the approach, we have also successfully implemented a user case to achieve inter-domain load balancing.

Chapter 1

Introduction

1.1 Introduction of SDN

Over the past decade the usage of network (Internet) has tremendously increased, which has consequently increased the number of physical network devices such as routers, switches, firewalls, load balancers and etc. Managing/Configuring these devices have always been a strenuous job in spite of the development of various management applications which are mostly vendor specific and not very robust. Apart from management there are also several other issues pertaining to the traditional networks. In traditional networks each device has its own independent view of the network based on the updates and advertisements it receives from the neighboring device, thus having minimum visibility. Different Vendors have their own implementation of traditional routing and switching protocols on their network devices. These implementations are hidden and hence none of the protocol implementations could be customized. Although most of the vendors these days offer programmability, they are not very flexible and cannot be applied over the entire network.

Most of the network related research don't make it to the real world due to lack of ways to experiment them in a realistic setup. SDN led to the advent of programmable networks by decoupling the control and data plane. The aim was to motivate vendors to implement a protocol (eg. OpenFlow) in the traditional networking devices such as routers and switches so that while network administrators could configure them running on the common network protocols, researchers could implement their ideas and run their experimental protocols at the same time. The traffic is separated into production flows and researcher flows and isolated from each other. The advent of SDN has promised to help to solve most of them. SDN offers Centralized network provisioning, fine grain traffic control, automation, reduced downtime, cost reduction etc.

In contrast to traditional networking (figure 1.1), SDN architecture (figure 1.2) separates the control plane from the traditional network device. The control plane is shifted to a centralized controller that can manipulate the forwarding decisions in the data plane of the devices.

As show in Figure 1.2 there is a single centralized controller that controls the dataplane of a group of devices. Since all the devices communicate with the controller, the controller has a complete view of the network and thus enables more felixibility when compared to a traditional network. This approach has brought a whole new dimension to the way networks can be conceived, controlled and managed.

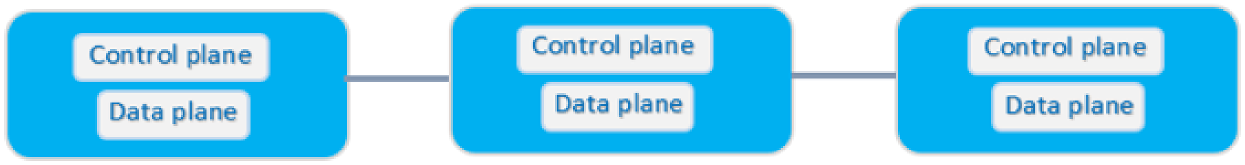


Figure 1.1: Traditional Networking

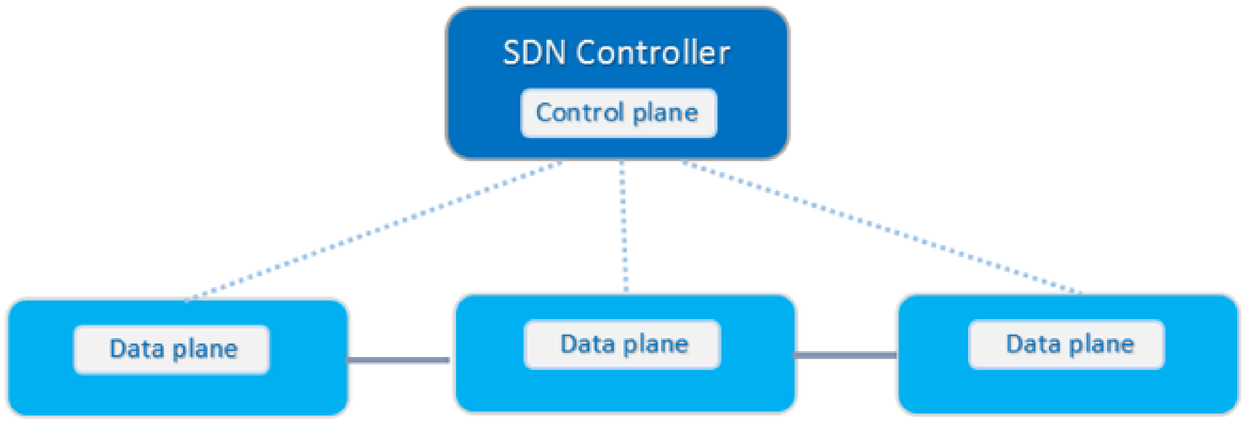


Figure 1.2: Software Defined Networking

There have been several successful SDN based data-centre implementations and applications in the real world by huge giants in the industry such as Google, facebook, AT & T etc and it is no longer a future or pilot technology.

In a distributed data centre environment there will be several controllers controlling several sets of devices. We call a group of devices under a SDN controller as its domain. Several proposals have been made to interconnect these domains. In this thesis we will discuss about the various approaches. We then propose a new approach and describe the implementation in detail.

1.2 Research Problem

SDN based Data centres have been scaling up rapidly. For better scalability and security a number of methods have been proposed to partition the SDN into multiple domains with each domain managed by an independent controller or a cluster of controllers. Apart from scalability, geographic distribution of data centres are also one other reasons for interconnecting multiple SDN domains. In order to setup the appropriate paths across multiple domains, the controllers must communicate with each other to exchange control information.

There are two approaches to enable communication between SDN controllers. First approach [1], [2] called the horizontal approach (figure 1.3), is to enable a communication protocol that is similar to a routing protocol such that the controller of a domain directly communicates with the controllers of the neighboring domains. The other approach [2], [3] is a hierarchical or vertical approach (figure 1.4) in which there is a central controller that communicates with the controllers of all the domains (domain controllers). In this approach, there is no inter-communication among the domain controllers.

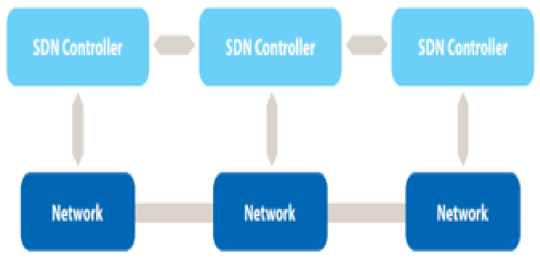


Figure 1.3 b): Horizontal Approach

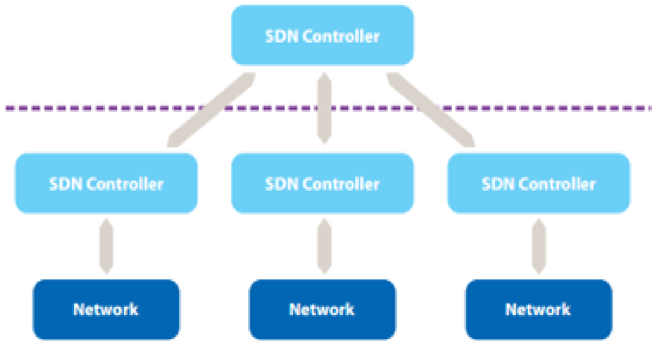


Figure 1.3 a): Vertical Approach

http://www.tcs.com/SiteCollectionDocuments/White%20Papers/Inter-SDN-Controller-Communication-Border-Gateway-Protocol-0314-1.pdf

We can use either layer-2 or layer-3 to interconnect multiple SDN domains. In this thesis, we propose to use layer-2 interconnection. This is because Zero Downtime is one of the most important requirements from various businesses these days. E-commerce and E-business are one among the many sectors that are most sensitive to network outages. Any minimal outages may lead to huge loss. Their data centres are often distributed across different geographic locations, which also serve the purpose of BCP (Business continuity planning). These Data centres could be connected using various layer 2 or layer 3 technologies. By using a layer-2 interconnection, we create a stretched layer 2 network which would facilitate live migration and many other major activities without any glitches in the production environment. This also means easy migration of servers without worrying about the subnetting restriction. Network Administrators would also find it easier to troubleshoot issues in such a network (with single L2 domain) where extending it is just like adding another switch. Apart from these they also aid stateful services such as local balancers and firewalls across the distributed domain. These services play a key role in ensuring zero downtime.

Chapter 2

Background

2.1 SDN

SDN [17] is the emerging technology that separates the control plane from the data plane, wherein real-time network devices could be programmed by intelligent software component. The goal of SDN is to leverage the centralized control plane in order to reduce the complexity of today's network and develop innovative ways in which network could be controlled and managed.

SDN architecture is divided into multiple layers as show in figure 2.1. We will introduce these layers in the subsequent sections.

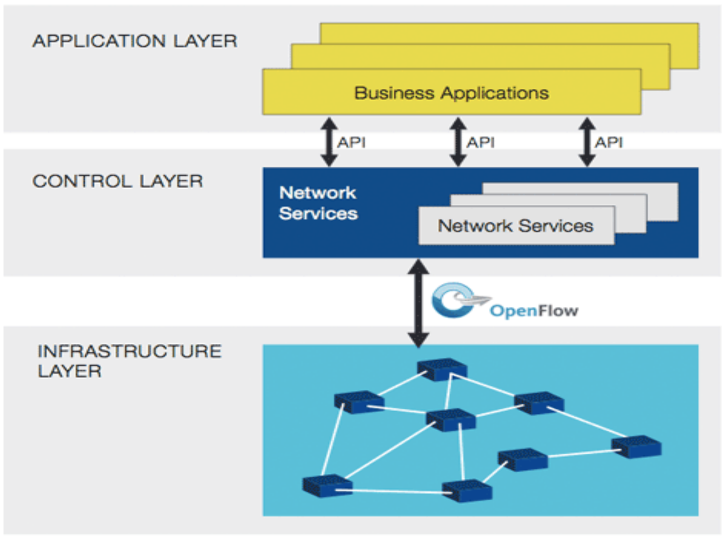


Figure 2. 1: SDN High-Level Architecture

https://www.opennetworking.org/sdn-resources/sdn-definition

2.1.1 Infrastructure Layer

The infrastructure layer is comprised of network devices. Network device may refer to the physical device or a virtual device. It may be implemented in hardware or software. This is the entity through which the actual user traffic/ data packets pass through. It receives data packets on its ports and may forward or discard or even alter them. The network device is a combination of ports, queues, memory and CPU. Switches, Routers and Firewalls are some of the common network devices.

A network device comprises of the Forwarding Plane and Operational Plane. The Forwarding plane also known as the data plane, deals with the handling of packets. Routing and Switching of packets are some of the common data plane functionalities. The Operational plane deals with operational state of a device such as status of the ports, number of packets transmitted/received over an interface, memory utilization, queue length etc.

2.1.2 Control Layer

The application that implements the functions of the control layer is known as the SDN controller. Its responsibility is to control the network devices in the infrastructure layer and enable paths for the user traffic. The control layer or the SDN controller has multiple components within itself. Their core functionalities could be divided into the Control plane and Management plane. The Control plane is the brain of the network which provides various services such as inventory management, topology management, Flow programming, route/path selection etc. The component that is used to communicate with the Forwarding plane of the network devices in the infrastructure layer is known as the Control-Plane Southbound Interface. This may be implemented as Protocol or API. ForCES (Forwarding and Control Element Separation) and OpenFlow are examples of Control-Plane Southbound protocols. The Management plane is a central point to collect data from the Operational plane of the network devices and provide various functionalities such as, Orchestration, Fault and Monitoring Management and Configuration management. This also plays an important role while implementing Network Function Virtualization (NFV). The Management plane has its own southbound interface to communicate with the Operational plane of the network device. Open vSwitch Database (OVSDB), NETCONF, SNMP and Syslog are few of the commonly used Management Plane Southbound protocols. The services provided by the control and data plane could be accessed by other applications and services through the Service Interface.

RESTful APIs and Remote Procedure call (RPC) are the most popular service interface implementations in the SDN controllers.

2.1.3Application Layer

SDN Applications, Business Applications, Cloud Orchestration, etc. reside on the application layer. They utilize the services offered by the control and management plane in the control layer using the service interface (RESTful APIs or RPC). This is the layer of innovation wherein several applications are built to solve today's business problems. Network Provisioning, Network Topology Map and Path reservation are some of the example applications.

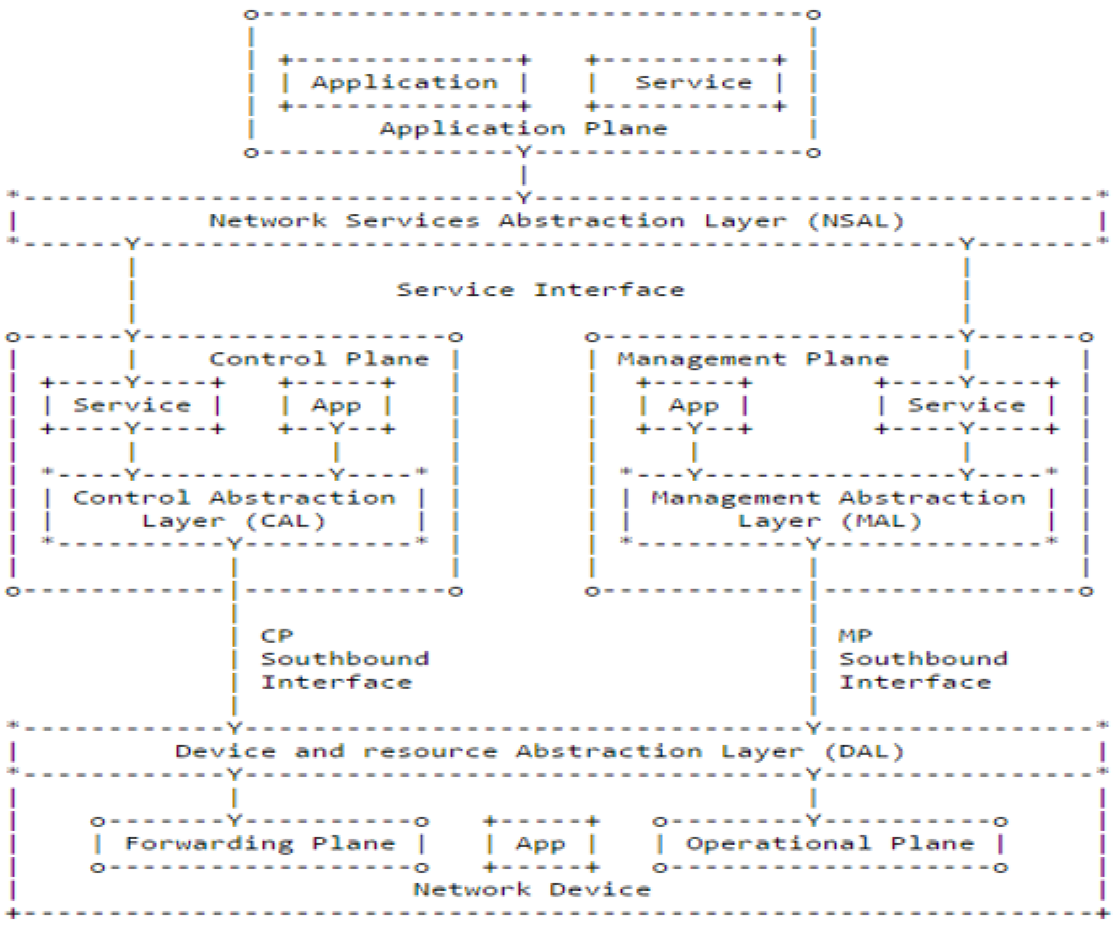


Figure 2. 2: SDN Architecture in Detail

https://tools.ietf.org/html/rfc7426

2.2 OpenFlow

OpenFlow [16] is one of the most popular SDN southbound protocol. It is an open standard protocol built with the intention to help researchers run their experimental protocols. Although we say that SDN removes the control plane functions from the devices in the infrastructure layer, some control protocols still need to be present so that the control layer and the infrastructure layer (handling the data plane) could talk with each other. OpenFlow is one of the protocols that the devices (infrastructure layer) and the SDN controller (Control Layer) use to enable communication between them. *The OpenFlow protocol has three messge types: controller-to-switch, asynchronous and symmetric.*

2.2.1 OpenFlow switch

OpenFlow switch is the fundamental component of the infrastructure layer. They initiate Asynchronous messages to notify the controller about network events and any changes in the state of the switch. As shown in the figure 2.3 the switch agent handles the communication with the SDN controller and the date plane. When a packet arrives at the switch in the data plane, the header information is attempted to match against the entries in the flow table (as shown in figure 2.4). Multiple tables may be present, but it goes through table 0 (default table) first. If there is a match then the instruction/action associated with the matched flow entry gets executed. If there is no match, the packet or the header of the packet is usually sent to the controller through the OFPT_PACKET_IN message [17] (Asynchronous message type) for further processing.

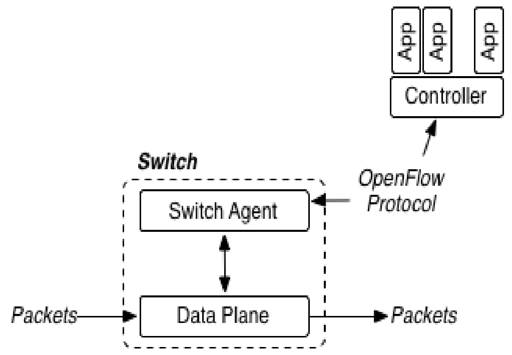


Figure 2.3 b): OpenFlow Switch Anatomy

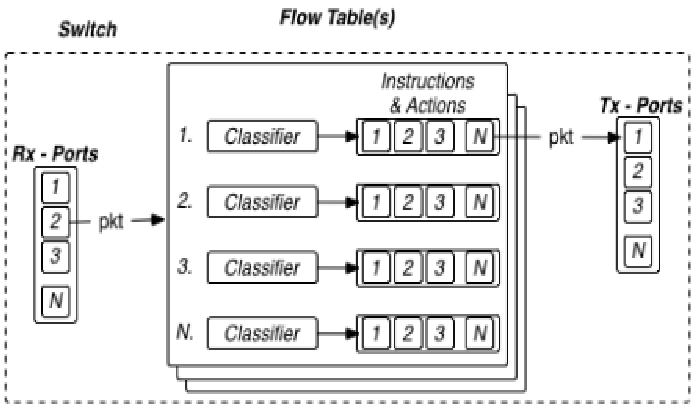


Figure 2.3 a): Data Plane

http://flowgrammable.org/sdn/openflow/#tab_switch

The major components of the flow table are shown in table 2.1.

Match Fields	Priority	Counters	Instructions	Timeouts	Cookie

Table 2: Flow Table Components

Match Fields: Some of the common match fields are Switch port, Source MAC, Destination MAC, Ether type, VLAN ID, Source IP, Destination IP, TCP source and destination port.

Priority: Higher the priority mean greater precendence

Counters: When a packet matches a flow entry, its counter is increased

Instructions: Forward packet to port(s), encapsulate and forward to controller, drop packet and send to normal processing pipeline are some of the commonly used instructions.

Time out: Time until the flow would remain unexpired

Cookie: non transparent data used by the controller

2.2.2 OpenFlow Controller

POX, NOX, Ryu, Floodlight, Contrail and Opendaylight are few of the well-known SDN controllers. All these controllers have implemented the OpenFlow protocol to communicate with the OpenFlow switches. These controllers possess various built-in modules that perform base network service functions (e.g. Topology Manager, Switch Manager etc.). These modules process the openflow packets sent by the switches and populate their database with Configuration data and Operational state data [15]. Switch ID, Port ID and Link information are some examples of Configuration data which is populated by the Topology Discovery module. Status of ports in a switch (“Up” or “Down") is an example of Operational state data which is populated by the Switch manager module. This data is exposed via APIs and can be accessed or manipulated by the Northbound Applications

2.3 SDN – Intra-Domain Switching

To enable communication between hosts in a SDN network, its controller must first be aware of the network topology. Hence topology Discovery is the foremost step to achieve this. Following that removing loops from the topology is another must requirement in a layer 2 network. In the following sections we will discuss how the various modules in the controller perform these key functionalities and how intra-domain switching can be achieved

2.3.1 Topology Discovery

SDN controller utilizes LLDP packets for Topology discovery. Specifically, it instructs the switches to send LLDP packets to their neighboring switches; when the neighboring switches receive the LLDP packets, they will forward the LLDP packet back to the controller. The controller using this LLDP information finds out the connections among switches and builds the overall topology of its domain. It should be noted that, by default, OpenFlow switches will not forward the received LLDP packets to the SDN controller. Hence the switches in the domain should be initially configured such that it sends all the packets to the controller.

2.3.2 Loop Remover

Switching loop is one of the common issues in a layer 2 domain which is caused by redundant paths in a layer 2 network. Switching loop will lead to broadcast storms due to flooding of Broadcast, Unknown unicast and Multicast (BUM) traffic. In a SDN network, the SDN controller computes the spanning-tree and installs it on the physical topology using the OpenFlow to configure the switch ports to either the flooding state or non-flooding state. A port in the flooding state forwards BUM traffic over its interface, whereas a port in the non-flooding drop the BUM traffic. Another way to setup the spanning tree is to install appropriate flow entries in the flow tables of the switches according to the tree topology.

2.3.3 Layer 2 Communication along the Shortest Path

In a traditional Ethernet network when a host wants to communicate with another host in the same network it sends out an ARP request. The ARP request & reply take the path built by the spanning tree.

The subsequent packets exchanged between them take the same path, which may not be the shortest path. Since the SDN controller has the complete view of the topology, hence when it receives an ARP reply packet, the controller can extract the source and destination information, identify the nodes they are connected to (using Host Tracker Service) and find the shortest path between them. It can then install the flow based on the shortest path computation. Most of the SDN controllers come with a module that does shortest path computation.

2.4 Literature Survey

[1] and [3] have the similar proposals to our work that follow the horizontal/serial and vertical/hierarchical approaches respectively. The design in [1] has a single controller which is logically centralized but physically distributed. In this approach all the SDN domain controllers have the complete state information of all the domains. This is achieved by propagating the events (OpenFlow message events) happening in each domain to the SDN controllers of other domain using an application called HyperFlow. This makes all the SDN domain controllers capable of taking over in case of failure of any SDN domain controller and thus providing scalability and redundancy. Thus any application which requires network state information can subscribe to HyperFlow and get the information. One of the major concerns of this approach is the overhead for the SDN controller due to huge increase in the number of events it needs to process for attaining network wide state information. This problem is addressed in [3]. In [3], the controller works on two different levels. One of them is the low-level controllers or the local controllers that takes care of local events (flow arrival, network statistics collection). The low-level controllers can be scaled linearly with the increase in size of the network. Local applications (example local policy enforcement, elephant flow detection, link layer discovery) are offloaded to the local processing resources of the local controller. The other one is the root controller that can subscribe to specific events in the local controllers. These are mostly events that are required to perform tasks that are non-local or that require network wide state information (example routing between SDN domains). The main objective of this approach is to scale the low level controllers and the local applications that rely on them.

SDNi [6] was one of the first works on SDN inter-domain communication. SDNi is a communication protocol that was developed to exchange information between SDN domain controllers. An IETF draft was submitted/published during the mid of June 2012, but it had expired at the end of the 2012 without any further development. OpenDaylight (ODL) project, one of the largest projects working on open source

SDN platform, announced their implementation on SDNi [2] during its yearly summit in July 2015. SDNi relies on the BGP protocol to exchange information such as reachability update, topology information, QoS information and Network events. The SDNi objective was only to transfer information so that they could be used to build applications which require network-wide state information.

There are few other proposals [7] [8] that follow the vertical approach. These approaches have an application that runs along with each SDN domain controller and handles the communication with other controllers. Apart from building the east-west communication protocol between SDN controllers they also deal with utilizing the information to build the end-to-end path between the hosts in different domains. In [7], the exchange of complete topology information does not take place, instead only virtualized/abstracted views of their network is communicated to its peer SDN domain controllers. In the use case of [7], Layer 3 routing between different SDN domains is achieved. [8] is another approach that is very similar to [7]. While [7] focuses on inter-domain communications in the same administrative domain, [8] aims to establish communication between different administrative domains by having a vendor independent communication protocol. Both the use cases of [8] are implemented on a Internet Service Provider environment where routes are exchanged by the SDN controllers and paths are built by the northbound applications.

The authors in [9] have proposed an east-west communication method (for exchanging control information) that is adaptable to network conditions such as high traffic or low bandwidth, link congestion and even link failures. They have also demonstrated a use case of migrating hosts between different SDN domains which is resilient to disruptions, using the distributed control plane information.

In [10] and [11], layer 3 routing is performed using the network-wide information obtained from routing exchanges among controllers. These papers are based on the proposal from [7] that leverage on its east-west communication protocol. Authors in [10] point out the shortcomings of BGP and proposes a solution to achieve fine-grained inter-domain routing using SDN. Unlike BGP routing decisions here can be made using TCP/UDP port number, protocol number and QoS attributes apart from just the destination prefix and AS number. Hence a single destination could have multiple paths and thus enables great flexibility for routing. [11] is an extension of [10] that addresses the concern of increased flow table entries caused after the enablement of inter-domain communication by adopting compression techniques. In this approach before a rule can make it to the switch's flow table they go through a decision table which is a multiple index table built using bloom filter. The decision table prevents the redundant rules from entering and tries to aggregate rules which are not redundant. The extensive inter-domain SDN testbed built for [10] and [11] and their results prove how SDN could be implemented in the large Internet Service Provider networks for exchanging enormous routing information.

2.5 Summary of proposed approach

Our work follows the vertical approach to enable inter-domain communication. We propose to have a central controller on top of all the SDN domain controllers. As show in the figure, each SDN based data centre is in a single layer 2 domain and they are connected to their neighboring data-centres/domains through a layer 2 link and thus extending the layer 2 connectivity. As a whole all of them belong to a single layer 2 domain, but each SDN domain is controlled by their respective SDN controller and they in turn are controlled by the central controller to enable inter-domain switching. Using a central controller might bring concerns about single point of failure and scalability issues, but as mentioned in [1], [5] and [2] the issues might only be a concern in Enterprise and Service Provider environment which have large number of domains. Since our emphasis is on Datacentres with the extended L2 network the number of domains will not be too many.

The central controller is a light-weight application with limited functionality in contrast to an SDN domain controller which has various functionalities other than just L2 switching. In this entire thesis we only consider a topology wherein all the data centres belong to a single layer 2 network.

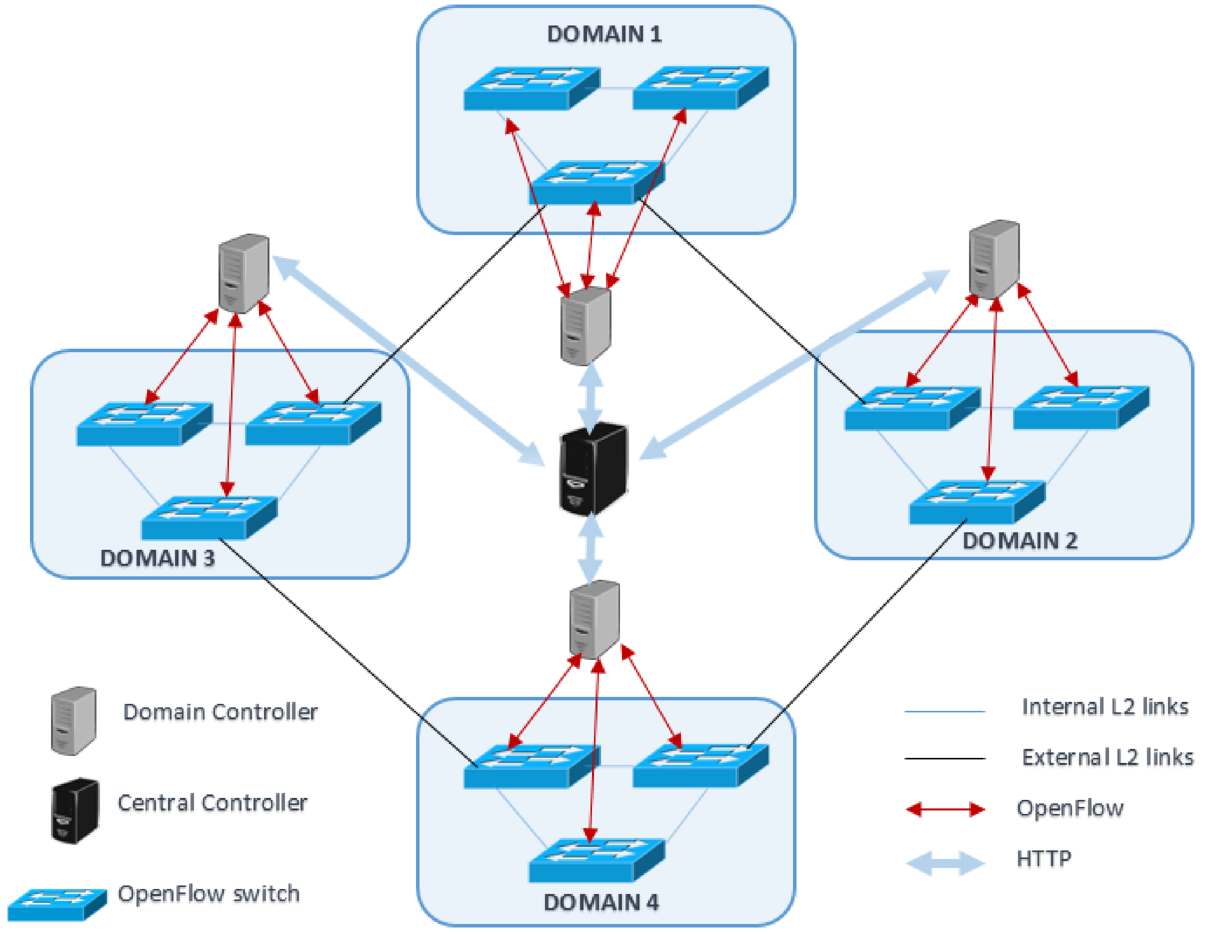


Figure 2. 3: SDN Inter-Domain Topology

Even though all the SDN domains belong to one Layer-2 network, the SDN domain controller of a domain/data centre would only be aware of the devices in its domain and not the other domains. Hence any intra-domain communication is handled by the corresponding SDN domain controller. Our goal is to design a mechanism to establish the layer-2 inter-domain paths to support inter-domain communication.

Chapter 3

Proposed Mechanism

As mentioned in section 2.5, our approach uses a central controller to communicate with the SDN domain controllers to enable inter-domain data forwarding. Following are the steps performed by the central controller to achieve inter-domain communication

i) Information exchange between the central controller and domain controllers
ii) Inter-Domain Topology Discovery & Loop removal
iii) Inter-Domain flow installation along the Shortest Path

3.1 Information Exchange

Each SDN domain controller has local information about the domain under its control. In order to achieve inter-domain communication, it is necessary for the central controller to communicate with the SDN domain controllers and acquire the local information of each domain. In our solution there is no requirement for any communication among the SDN domain controllers.

The central controller which is a northbound application communicates with all the SDN domain controllers via their exposed API's. Depending on the type of API that is exposed, corresponding calls can be made to access/modify the information stored in the domain controllers. If for instance the exposed APIs are REST APIs then appropriate HTTP calls can be made. The central controller needs to know IP addresses of all the SDN domain controllers in order to communicate with their APIs. These IP addresses need to be manually configured at the central controller. Also, depending on the type of API, security features need to be explored to authenticate the central controller. The security issue, however, is not in the scope of this thesis.

3.2 Inter-domain Topology Discovery & Loop Removal

In a topology involving multiple SDN domains, a switch can be an internal switch or border switch. An internal switch only has internal connections with hosts/servers and other switches that belong to the same domain. On the other hand, a border switch has at least one connection with the switch in the other domain. The port connecting to a switch in the other domain is referred as the External port.

Since the domain controller does not know the inter-domain topology, it's important that no BUM traffic is flooded through the external port before the setup of an inter-domain spanning tree. This is because any redundant inter-domain connection would lead to broadcast storms and duplication of traffic.

Since the devices in each domain are controlled by their respective domain controller, topology discovery and loop issues within a domain are not handled by the central controller Instead, it is the domain controller that builds the internal spanning tree for its domain. Here we defined the domain level topology as the abstract topology where each domain is represented by a logical switch as illustrated in figure 3.1b. The central controller's first focus is to install a spanning tree over the domain-level topology. We name such tree as the domain-level spanning tree. Thus, the complete spanning tree is the internal spanning trees of all the domains connected by the domain-level spanning tree. To build the domain-level spanning tree, the central controller first builds the domain-level topology using the *Topology building Algorithm* to be discussed next.

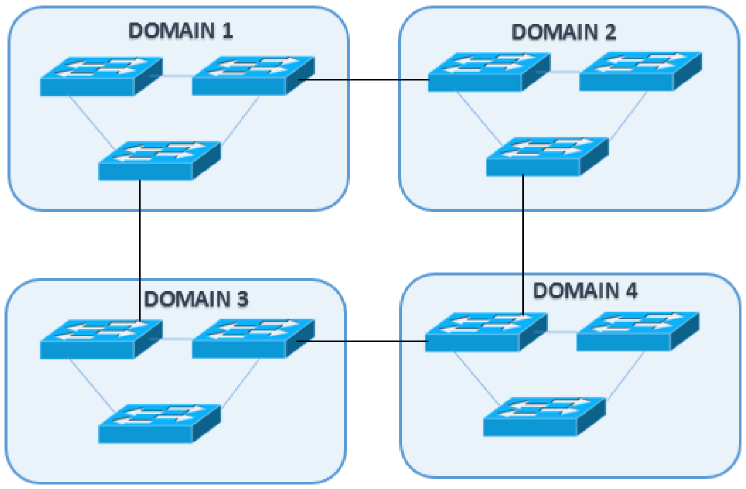


Figure 3.1 b): Actual Topology

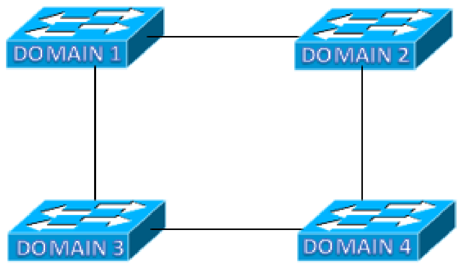


Figure 3.1 a): Central Controller's abstracted view

3.2.1 Topology Building Algorithm

The central controller runs the topology building algorithm to abstract the domain-level topology from the complete physical topology. To run the algorithm, the central controller first gathers the following information from the domain controllers.

Domains = $\{D_1, D_2, D_3 \dots D_i\}$

H_i = $\{h_{i1}, h_{i2}, h_{i3} \dots h_{in}\}$

S_i = $\{s_{i1}, s_{i2}, s_{i3} \dots s_{in}\}$

L_i = $\{ l^{m\leftrightarrow p}_{1_{s_{ix}\leftrightarrow s_{iy}}}, l^{m\leftrightarrow p}_{2_{s_{ix}\leftrightarrow s_{iy}}} \dots l^{m\leftrightarrow p}_{n_{s_{ix}\leftrightarrow s_{iy}}} \}$

Where D_i is the ID of domain i, H_i is the set of hosts in domain i, h_{in} is the ID of host n in domain i, S_i is the set of switches in domain i, s_{in} is the ID of switch n in domain i, L_i is the set of links in domain i, $l^{m\leftrightarrow \mathrm{p}}_{n_{s_{ix}\leftrightarrow s_{iy}}}$ is the ID of link , where m and p are source and destination ports, and s_{ix} and s_{iy} are source and destination switches. Note that the ID of the switch has two parts: the domain ID and the local ID. The controllers identify the switch based on its ID. The pseudocode of the Topology Building Algorithm is given in Fig. 3.2.

Algorithm Pseudocode for building the Domain-level Topology

```
1:  for each $D_i$ in Domains do
2:    add each domain as node to graph G
3:    collect the hosts, switches and links information from the domain $D_i$ and stores it in $H_i$, $S_i$ and
      $L_i$ respectively
4:  end for
5:  for each $D_i$ in Domains do
6:    for each $D_j$ in Domains do
7:      if $D_j$ is not $D_i$
8:        for each $s_{in}$ in $S_i$ Domains do
9:          for each $l^{m\leftrightarrow p}_{n_{s_{jx}\leftrightarrow s_{jy}}}$ in $L_j$ Domains do
10:           if ($s_{in} = s_{jx}$ in $l^{m\leftrightarrow p}_{n_{s_{jx}\leftrightarrow s_{jy}}}$) or ($s_{in} = s_{jy}$ in $l^{m\leftrightarrow p}_{n_{s_{jx}\leftrightarrow s_{jy}}}$)
11:             add edge between nodes $D_i$ and $D_j$ node in graph G and save border port ID
                information
12:           end if
13:         end for
14:       end for
15:     end if
16:   end for
17: end for
```

Figure 3. 2: Algorithm – Pseudocode for building the Domain-level Topology

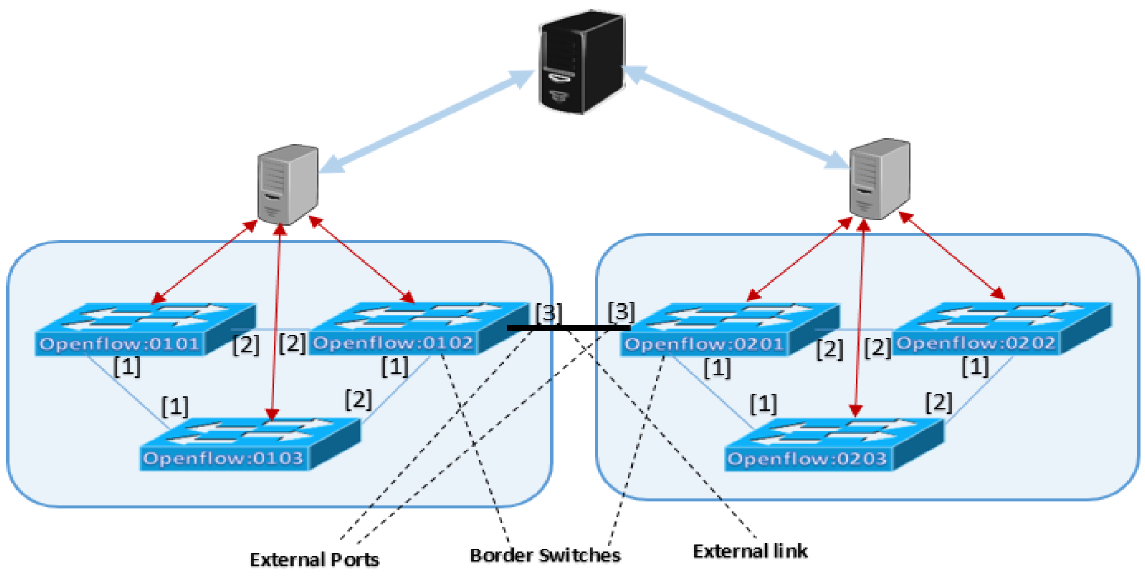


Figure 3. 3 Basic Topology (4 domains)

Let us use an example to illustrate the algorithm. We use the topology of Figure 3.3 for this and subsequent examples. In Fig 3.3, we identify a switch by its ID using 4 digits in the form of DDLL, where DD and LL identify the domain ID and switch number, respectively. A switch with the ID of “0102”, then, is a switch in domain 1 whose number is 2. In Figure 3.3, switch “0102” has an external connection with switch “0201”. In the topology discovery phase, switch “0102” will receive a LLDP packet on port 3 sent by switch “0201” on port 3 (and vice versa), it forwards the LLDP packet to its domain controller and the controller builds the link information. The format of the information of the link ($l_{n_{s_{jx}\leftrightarrow s_{jy}}}^{m\leftrightarrow \mathrm{p}}$) between switches “0102” and “0201” used in our implementation is displayed below.

```
"link-id": "openflow:0102:3",
        "source": {
          "source-tp": "openflow:0102:3",
          "source-node": "openflow:0102"
        },
        "destination": {
          "dest-node": "openflow:0201",
          "dest-tp": "openflow:0201:3"
        }
```

Based on the above information from the domain controller, the central controller can identify an external link that connect domain 1 and 2.

The central controller then builds a graph G where a_{mn} is a set that contains border switch and port IDs in domain m that are connected to domain n. Essentially, the information of the link connecting domains n and m is split into a_{mn} and a_{nm}.

$$G = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & a_{m3} & \ddots & a_{mn} \end{bmatrix}.$$

Continuing our example, we use the information of the link connecting domains 1 and 2 to derive a_{12} and a_{21}. The content of a_{12} consists of the border switch and its external port in domain 1; similarly, the content of a_{21} consists of the border switch and its external port in domain 2:

a_{12} = {openflow:0102:3} where 01 is the domain ID, 02 is the switch number and 3 is port number

a_{21} = {openflow:0201:3} where 02 is the domain ID, 01 is the switch number and 3 is port number

If there is no external connection between the domains m and n, a_{mn} and a_{nm} will be null.

After building the domain-level topology the central controller runs the loop removal algorithm and then overlay the tree by installing flows on the border switches based on the tree information.

3.2.2 Loop Removal Algorithm

Graph G is the output of domain-level topology building algorithm and the input to the method *Minimum_Spanning_Tree*. The method *Minimum_Spanning_Tree* implements the spanning tree algorithm and returns the border switches and port IDs through which BUM traffic should not be forwarded.

Algorithm Pseudocode for Loop Removal

```
1:      $M$ ← Minimum_Spanning_Tree ($G$)
2:      for each $a_{mn}$ in $M$ do
3:        if $a_{mn}$ is not $Null$
4:            NODE_ID ← Extract switch ID from $a_{mn}$
5:            PORT_NO ← Extract Port number from $a_{mn}$
6:            Call FlowInstall_Block (NODE_ID, PORT_NO)
7:            NODE_ID ← Extract switch ID from $a_{nm}$
8:            PORT_NO ← Extract Port number from $a_{nm}$
9:            Call FlowInstall_Block (NODE_ID, PORT_NO)
10:       end if
11:     end for
```

Figure 3. 4: Algorithm – Pseudocode for Loop Removal

Once the spanning tree is determined, the *FlowInstall_Block (switch ID, port number)* method takes switch ID and port number as arguments and installs a flow in the switch's flow table to block any inbound BUM traffic on that port. Continuing our previous example, if the algorithm determines that the external link between switches "0102" and "0201" should not be a part of the domain-level spanning tree, flows are installed to block any data traffic on port 3 of switch 0102 and port 3 of switch 0201. The installation of flows is again done using the RESTful API by sending HTTP PUT requests to the associated SDN domain controllers.

3.3 Inter-Domain flow installation along the Shortest Path

From the previous section we know that the central controller constructs the domain-level topology based on the information it received from all the SDN domain controllers then builds the domain-level spanning tree and blocks the ports that are not part of the tree

Whenever a host from one domain sends an ARP request broadcast to begin communication with another host in a remote domain, the ARP request broadcast travels across the local spanning-tree path built by the domain controller. Once the broadcast reaches the border switches, the decision to forward to other domain is based on the domain-level spanning tree installed by our central controller in the border switches. Thus the ARP request broadcast would flood over the complete spanning tree (Local spanning tree + the domain-level spanning tree). During this entire process whenever an ARP packet arrives at an OpenFlow switch, apart from forwarding the packet along the spanning tree path, the packet is also forwarded to their respective domain controller. Since ARP request is a broadcast packet and the domain controller knows that it has already constructed a spanning tree, the domain controller takes no action when the packet arrives.

When the ARP reply unicast is sent by the destination, it reaches the OpenFlow switch and the switch again forwards the packet to their respective domain controller. Now, unlike the ARP request broadcast packet whose destination address would have been a L2 broadcast address, the ARP reply packet has a reachable L2 address in its destination address field. Since there is information on both the source and destination, building an optimal (shortest) path between them is possible. Each domain controller will build a shortest path from the ingress switch to the egress switch within its domain. The ingress switch is the switch the controller receives the packet. If the domain is not the destination domain, the egress switch will be a border switch. The central controller, which is aware of the domain-level topology, will compute the domain-level shortest path and provide the IDs of the border switch and the external port to the domain controller.

We propose two approaches here to achieve inter-domain communication and ensure that optimal paths are taken.

3.3.1 First Approach

Figure 3.6 gives the flow chart of the first approach. In this approach we propose to install a local agent at each domain controller. This approach requires some modifications to be made in the domain controller to make it compatible with the client application. Depending on the vendor of the domain controller the changes that need to be made might vary, but these changes should be very minimal.

In this approach the central controller collects the end-host information of each domain from the domain controllers through their exposed APIs. It then uses the *Dijkstra's shortest-path algorithm* to determine the inter-domain shortest path to reach each domain. Based on that, it computes a databases for respective domains. Each entity of the database (table) of a domain contains the MAC address of a host and the corresponding Domain IDs. It also contains the border switch (with respect to the domain) that should be used based on the inter-domain shortest path to reach the host. It then populates the database to the agents of respective domains. The Central Controller registers with all the domain controllers for API change notification, so that whenever a domain controller discovers a new host it will notify the central controller. The central controller in turn would update the databases of the agents and thus ensuring that the databases stay up to date.

We will use an example to illustrate the concept. Figure 3.5 shows a four-domain topology. Let us concentrate on domain 3. The Table 3.1 shows the local agent database of domain 3. The table is populated by the central controller. Based on the domain-level topology, the shortest paths for domain 3 to reach domains 1 and 4 are through switches 31 and 33, respectively. There are two shortest paths from domain 1 to domain 2, through switches 31 and 33. Since hosts A, B and C are in domain 1. The central controller will push the entries associated with hosts A, B and C to the database of the agent in domain 3. Each of these entries points to switch 31 as the egress switch and port 4 as the external port. The other entries in the table are derived in the similar fashion. Since domain 2 can be reached from domain 3 via domain 1 and domain 4 (two different shortest paths), the entries associated with domain 2 have two options for egress switch. With this information load balancing could be easily achieved. Note that we can introduce additional attributes such as load of the links and reliability to the table. Thus, the proposed method is readily support more sophisticated forwarding mechanism.

Modifications should be made in the domain controllers such that, whenever they receive an ARP reply unicast and if they cannot locate the destination Layer 2 address in their local domain, they should contact the agent to get the inter-domain forwarding information. Now the agent would refer its database and fetch the egress switch and the external port information through which the destination could be reached based on the inter-domain shortest path. After receiving the information from the agent, the domain controller will call the shortest path flow installation algorithm to determine the shortest path between the ingress and egress switches within the domain and then install bidirectional flows on the switches according to the shortest path.

Note that when the ARP reply packet reach the transit domain, both the ingress and egress switches used to compute the intra-domain shortest path are the border switches of the transit domain. Also note that when the ARP reply packet reach the destination domain, the corresponding controller does not need to contact its agent for it knows the location of the destination host, thus the destination switch. Consequently, the controller just needs to build the intra-domain shortest path from the border switch to the destination switch.

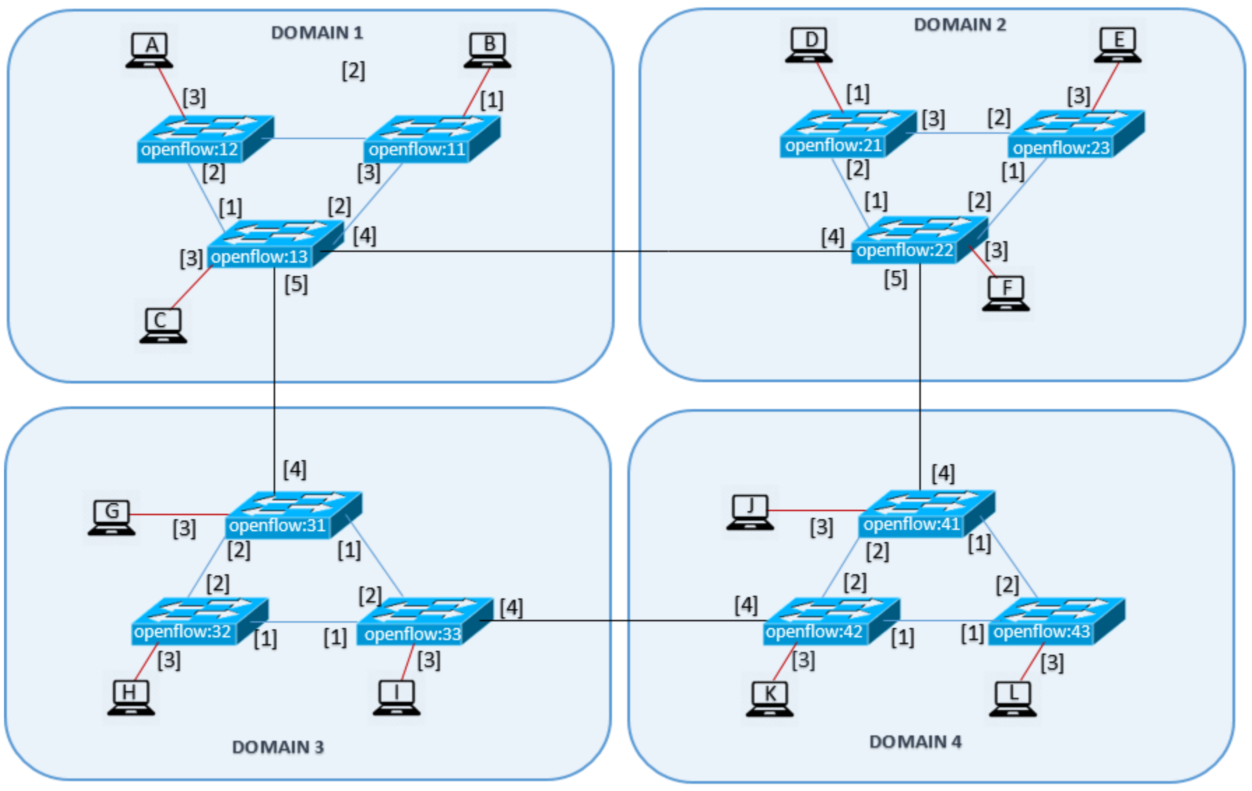
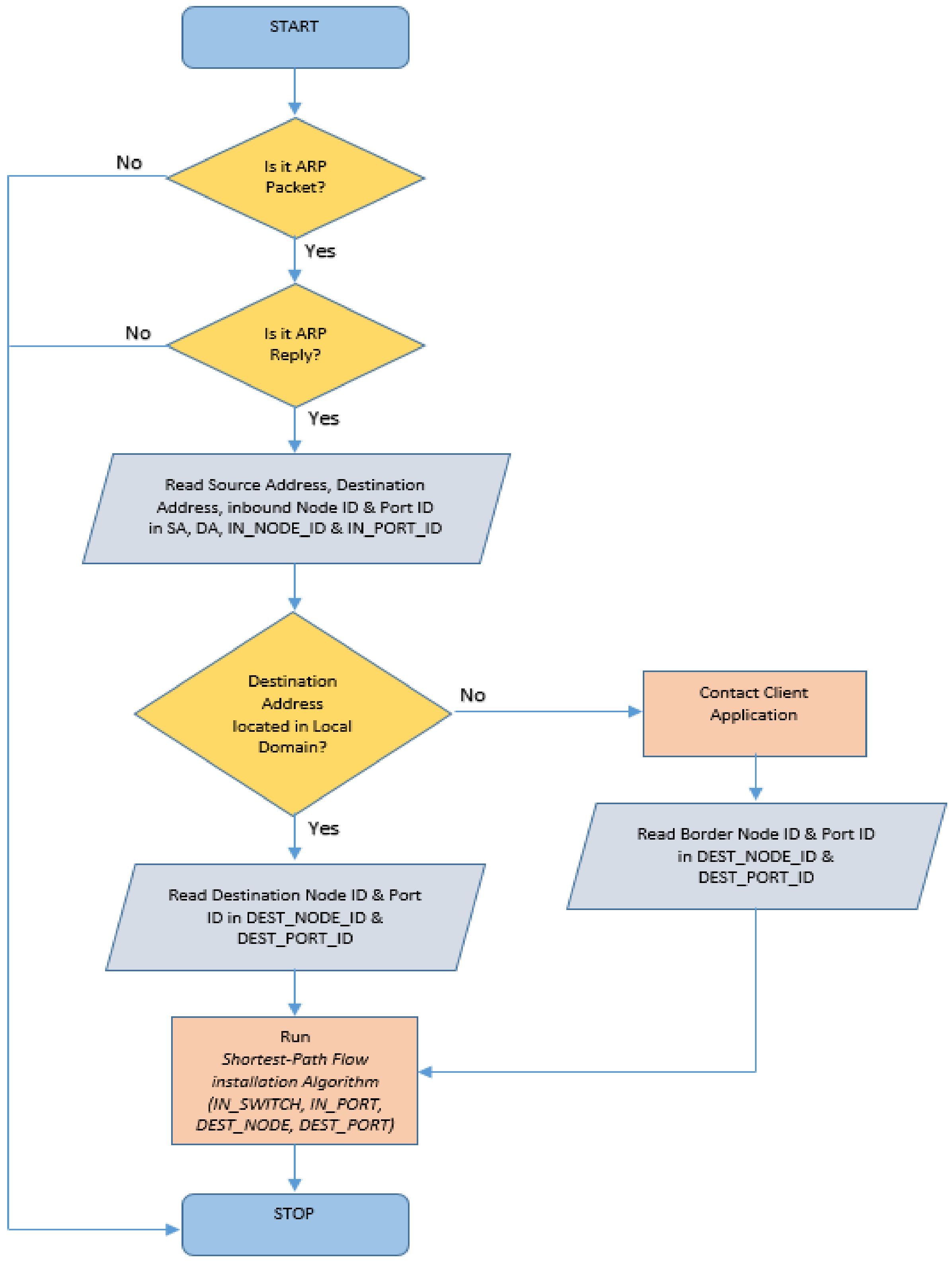


Figure 3. 5: Basic Topology (4 domains)

MAC Address	Domain ID	Exit Port
MAC - A	1	openflow:31:4
MAC - B	1	openflow:31:4
MAC - C	1	openflow:31:4
MAC - D	2	openflow:31:4 openflow:33:4
MAC - E	2	openflow:31:4 openflow:33:4
MAC - F	2	openflow:31:4 openflow:33:4
MAC - F	4	openflow:33:4
MAC - K	4	openflow:33:4
MAC - L	4	openflow:33:4

Table 3. 1: Domain 3 Agent's Database



START
Is it ARP Packet?
No
Yes
Is it ARP Reply?
No
Yes
Read Source Address, Destination Address, inbound Node ID & Port ID in SA, DA, IN_NODE_ID & IN_PORT_ID
Destination Address located in Local Domain?
No
Yes
Contact Client Application
Read Border Node ID & Port ID in DEST_NODE_ID & DEST_PORT_ID
Read Destination Node ID & Port ID in DEST_NODE_ID & DEST_PORT_ID
Run *Shortest-Path Flow installation Algorithm (IN_SWITCH, IN_PORT, DEST_NODE, DEST_PORT)*
STOP

Figure 3. 6: Work Flow for Approach 1

3.3.1.1 Shortest-Path Flow Installation Algorithm

Each domain controller runs the Shortest-Path Flow installation algorithm to install the intra-domain shortest path inside the domain. The Pseudo code of the algorithm is shown in Fig 3.7. Let SRC_MAC and DST_MAC denote Source MAC address and Destination MAC address respectively. Let INGRESS_NODE and I_PORT be the switch and port through which the SRC_MAC could be reached. Similarly, let EGRESS_NODE and E_PORT be the switch and port through which DST_MAC could be reached. The INGRESS_NODE and I_PORT information are derived when the ARP reply packet when it is received by the domain controller (The ARP reply packet arrived on I_PORT of INGRESS_NODE). The EGRESS_NODE and E_PORT information is retrieved by the domain controller from its local agent if the domain is not the destination domain. By running the shortest path algorithm, the domain controller derives a set, which contains a set of links that form the shortest path.

$Path$ = { $l^{m\leftrightarrow p}_{1_{s_{ix}\leftrightarrow s_{iy}}}$, $l^{m\leftrightarrow p}_{2_{s_{ix}\leftrightarrow s_{iy}}}$ … $l^{m\leftrightarrow p}_{n_{s_{ix}\leftrightarrow s_{iy}}}$ }

where $l^{m\leftrightarrow \mathrm{p}}_{n_{s_{ix}\leftrightarrow s_{iy}}}$ is the link in domain i , m and p are source and destination ports, and s_{ix} and s_{iy} are source and destination switches.

Once the shortest path is derived, the algorithm will install the flow on every switch along the path as described in $Path$.

Algorithm Pseudocode for Flow Installation along the Shortest Path

```
1:  if EGRESS_NODE == INGRESS_NODE
2:      NODE_ID ← EGRESS_NODE
3:      Call FlowInstall_Forward (NODE_ID, SRC_MAC, DST_MAC, E_PORT)
4:      Call FlowInstall_Forward (NODE_ID, DST_MAC, SRC_MAC, I_PORT)
5:  else
6:  $Path$ ← DijkstraShortestPath (INGRESS_NODE, EGRESS_NODE)
7:      For $l^{m\leftrightarrow p}_{n_{s_{ij}\leftrightarrow s_{ik}}}$ in $Path$
8:          if $s_{ix}$ == INGRESS_NODE
9:            NXT_NODE_ID ← $s_{iy}$
10:           Call FlowInstall_Forward ($s_{ix}$, SRC_MAC, DST_MAC, $m$)
11:           Call FlowInstall_Forward ($s_{ix}$, DST_MAC, SRC_MAC, I_PORT)
12:           Call FlowInstall_Forward ($s_{iy}$, DST_MAC, SRC_MAC, $p$ )
13:             if $s_{iy}$ == EGRESS_NODE_ID
14:               Call FlowInstall_Forward ($s_{iy}$, SRC_MAC, DST_MAC, E_PORT)
15:             end if
16:         else if $s_{iy}$ == INGRESS_NODE
17:           NXT_NODE_ID ← $s_{ix}$
18:           Call FlowInstall_Forward ($s_{iy}$, SRC_MAC, DST_MAC, $p$)
19:           Call FlowInstall_Forward ($s_{iy}$, DST_MAC, SRC_MAC, I_PORT)
20:           Call FlowInstall_Forward ($s_{ix}$, DST_MAC, SRC_MAC, $m$)
21:             if $s_{ix}$ == EGRESS_NODE_ID
22:               Call FlowInstall_Forward ($s_{ix}$, SRC_MAC, DST_MAC, E_PORT)
23:             end if
24:         else if $s_{iy}$ == NXT_NODE_ID
25:           NXT_NODE_ID ← $s_{ix}$
26:           Call FlowInstall_Forward ($s_{iy}$, SRC_MAC, DST_MAC, $p$)
27:           Call FlowInstall_Forward ($s_{ix}$, DST_MAC, SRC_MAC, $m$)
28:             if $s_{ix}$ == EGRESS_NODE
29:               Call FlowInstall_Forward ($s_{ix}$, SRC_MAC, DST_MAC, E_PORT)
30:             end if
31:         else if $s_{ix}$ == NXT_NODE_ID
32:           NXT_NODE_ID ← $s_{iy}$
33:           Call FlowInstall_Forward ($s_{ix}$, SRC_MAC, DST_MAC, $m$)
34:           Call FlowInstall_Forward ($s_{iy}$, DST_MAC, SRC_MAC, $p$)
35:             if $s_{iy}$ == EGRESS_NODE
36:               Call FlowInstall_Forward ($s_{iy}$, SRC_MAC, DST_MAC, E_PORT)
37:             end if
38:         end if
39:     end for
41:   end if
```

Figure 3. 7: Pseudocode for Flow Installation along the Shortest Path

3.3.2 Second Approach

In the second approach, we propose to update the database of the SDN domain controller directly using the APIs without the agent. Similar to the previous approach the Central controller fetches the host information from all the SDN domain controllers. Then, it computes the inter-domain shortest path tree for a given source and destination. After the computation of the shortest-path, the central controller updates the local databases (host tracker database) of all the SDN domain controllers with the host information of other domains.

In order to clarify the procedure further, we need to introduce the structure of the local database of the domain controller. Essentially, each entry of the local database consists of the MAC address of the host, the ID of the switch it is attached to and the associated switch port ID. For a single domain, all the hosts are internal hosts, that is, the hosts are attached to the switches that belong to the domain. Usually, the information in the database is derived by the domain controller. Our second approach uses the same database but allows the central controller to push the entries associated with the MAC addresses of the external hosts into the database. The IDs of switches and ports in these entries, however, will be the IDs of the corresponding border switches and external ports. In this way, the domain controller will treat the external hosts as internal hosts and build the intra-domain shortest path using the same procedure of building the shortest path between two internal hosts.

The following example demonstrates the approach. Referring to the network in Fig 3.8 and focusing on Domain 3. The controller of domain 3 will derive the local database of internal hosts as shown in Table 3.2. The central controller, in turn, will push the entries of external hosts to this database as shown in Table 3.3. With the extra information, the domain controller is capable of setting up the shortest path to the external host across its domain. For example, if host H sends a packet to host A, the domain controller will think that host A is attached to switch 31 on port 4. Consequently, it will setup a shortest path from switch 32, port 3 to switch 31, port 4. This setup will lead the packets from host H to host A to exit on port 4 of switch 31 to reach domain 1. Once they reach domain 1, they follow the shortest path installed by the domain controller of domain 1 to reach host A.

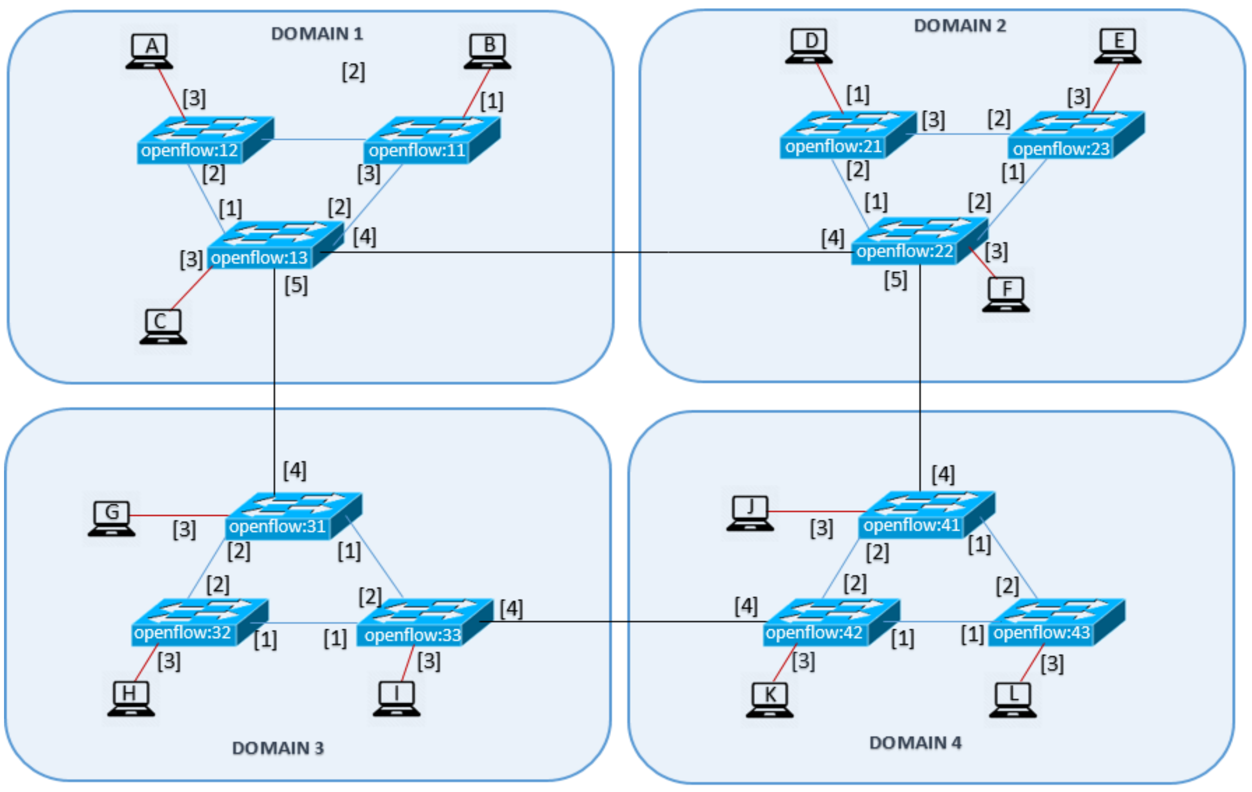


Figure 3. 8 : Basic Topology (4 domains)

MAC Address	**Port**
MAC – G	openflow:31:3
MAC – H	openflow:32:3
MAC – I	openflow:33:3

Table 3. 2: Domain 2's Controller Database

MAC Address	**Port**
MAC – G	openflow:31:3
MAC – H	openflow:32:3
MAC - I	openflow:33:3
MAC – A	openflow:31:4
MAC – B	openflow:31:4
MAC – C	openflow:31:4
MAC - D	openflow:31:4
MAC – E	openflow:31:4
MAC - F	openflow:31:4
MAC – J	openflow:33:4
MAC – K	openflow:33:4
MAC – L	openflow:33:4

Table 3. 3: Domain 2's Controller Database (After Central controller's update)

3.3.3 Comparison between the approaches

As mentioned before the second approach does not require any modification in the SDN controller application. Hence it is easily deployable when compared to the first approach which requires modification on the SDN controller application. In approach two, however not all inter-domain paths could be utilized i.e. if multiple paths exist to reach an external destination host, only one of them could be used, meaning load balancing is not possible. The central controller in this approach proactively updates the local database of the domain controllers. In approach one, the SDN domain controller reactively reaches our agent when it cannot find the destination address in its domain. The decision on path selection is made by the agent, based on the information it receives from the central controller. Since the decision making process is influenced by the central controller which is aware of the overall inter-domain topology, it is possible to achieve traffic-driven load balancing with this approach. Apart from shortest path and load balancing, Approach 1 is also more flexible and open for further enhancement. Because of these advantages of approach 1 over approach 2, we will implement approach 1 only and test our design in the OpenDayLight (ODL) SDN platform.

Chapter 4

Implementation

We will implement approach 1 using the OpenDayLight Platform. To test the implementation, we use Mininet to emulate a Layer 2 network with inter-domain topology.

4.1 System Environment

Mininet

Mininet is a software application used to build custom realistic network topologies. It runs on linux operating system and has been popularly used to simulate SDN networks and perform various tests in them. The network topology mostly comprises of Open vswitches, links and virtual hosts. Custom and very complicated networks could easily be built by using the python code. Open vswitch is an open source virtual switch that is used extensively in network virtualization under production environment.

Opendaylight

Opendaylight is one of largest open source Software Defined Networking projects. This was founded by huge consortium in the networking industry such as Cisco, Juniper, Brocade, Arista, Ericsson, HP, Microsoft, Red Hat etc. The controller runs in a JVM and hence can run on any operating system that supports Java. As show in the figure below OpenFlow is one among the many Southbound protocols that it supports. The SAL layer provides abstraction to the modules north of it. The various in-built modules utilizing the services provided by the SAL offer various network services. The controller exposes Northbound RESTful APIs to enable developer to write applications on top. It provides HTTP basic authentication to access them.

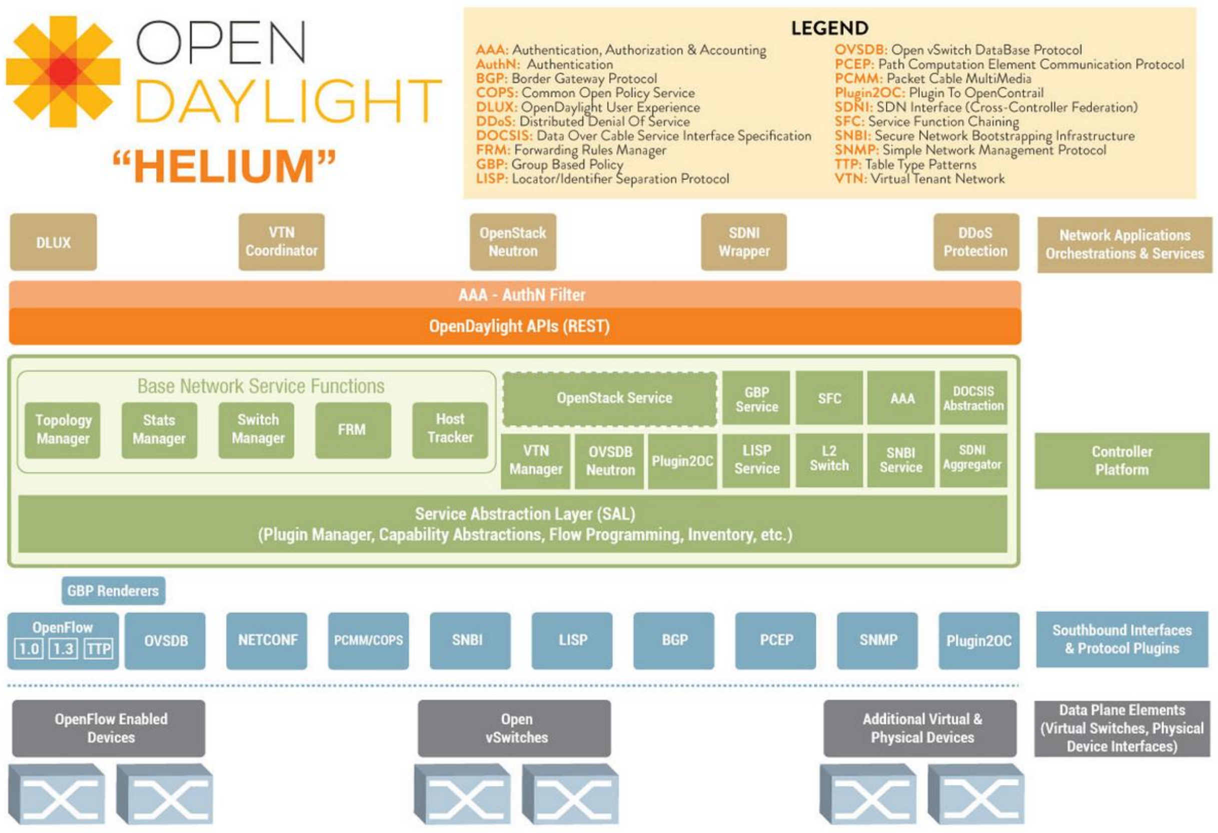

OPEN DAYLIGHT
"HELIUM"
LEGEND
AAA: Authentication, Authorization & Accounting
AuthN: Authentication
BGP: Border Gateway Protocol
COPS: Common Open Policy Service
DLUX: OpenDaylight User Experience
DDoS: Distributed Denial Of Service
DOCSIS: Data Over Cable Service Interface Specification
FRM: Forwarding Rules Manager
GBP: Group Based Policy
LISP: Locator/Identifier Separation Protocol
OVSDB: Open vSwitch DataBase Protocol
PCEP: Path Computation Element Communication Protocol
PCMM: Packet Cable MultiMedia
Plugin2OC: Plugin To OpenContrail
SDNI: SDN Interface (Cross-Controller Federation)
SFC: Service Function Chaining
SNBI: Secure Network Bootstrapping Infrastructure
SNMP: Simple Network Management Protocol
TTP: Table Type Patterns
VTN: Virtual Tenant Network
DLUX
VTN Coordinator
OpenStack Neutron
SDNI Wrapper
DDoS Protection
Network Applications Orchestrations & Services
AAA - AuthN Filter
OpenDaylight APIs (REST)
Base Network Service Functions
Topology Manager
Stats Manager
Switch Manager
FRM
Host Tracker
OpenStack Service
GBP Service
SFC
AAA
DOCSIS Abstraction
VTN Manager
OVSDB Neutron
Plugin2OC
LISP Service
L2 Switch
SNBI Service
SDNI Aggregator
Controller Platform
Service Abstraction Layer (SAL)
(Plugin Manager, Capability Abstractions, Flow Programming, Inventory, etc.)
GBP Renderers
OpenFlow 1.0 1.3 TTP
OVSDB
NETCONF
PCMM/COPS
SNBI
LISP
BGP
PCEP
SNMP
Plugin2OC
Southbound Interfaces & Protocol Plugins
OpenFlow Enabled Devices
Open vSwitches
Additional Virtual & Physical Devices
Data Plane Elements (Virtual Switches, Physical Device Interfaces)

Figure 4. 1: OpenDaylight Architecture

http://sdnhub.org/wp-content/uploads/2013/11/opendaylight_helium.jpg

4.2 System Design

The OpenDaylight controller has a number of built-in modules that support various basic network functions and services as show in figure 4.1. L2Switch is a one of the built-in modules within the OpenDaylight Controller. The module implements various layer 2 functionalities. Our implementation is based on this module. Following are the various components within the L2Switch module and their functions in brief.

- Packet Handler: This component is used to classify and decode the packets based on the EtherType. They possess decoders for Ethernet, ARP, IPv4 and IPv6

- Loop Remover: This component constructs a loop free graph using Spanning Tree protocol and removes network loops in a layer 2 domain. They move the status of all the switch ports either to “Forwarding” state or “Discarding” state (based on STP). They enable default flows on all the switches to forward all the LLDP packets to the SDN controller. They also listen to topology change events to update any changes in its loop free graph.

- ARP Handler: It installs a flow on all the switches to forward any ARP packets it receive to the controller. It processes the incoming ARP packets and forward them to ports that are in “Forwarding” state.

- Address Tracker: Subscribes for ARP, IPv4 and IPv6 packet notifications. Once they receive any of those packets, they update the addresses in their inventory.

- Host Tracker: Similar to address tracker, here the host attachment to the switch is updated to the topology

- L2Switch-Main: It installs flows based of MAC addresses in the switches dynamically when a packet arrives to the controller. Reactive Flow Writer a sub-components within L2Switch-main module subscribes to the Packet Handler module (which decodes ARP packets from the Ethernet packet) for receiving ARP packets. Once it receives the ARP packet, destination MAC address, Source MAC address and Ingress switch details are extracted from it. Following this the packet is sent to Inventory Reader module which finds out the location of the address within its domain. If the source and destination are attached to the same switch, a bidirectional flow is installed.

To achieve our goal, below changes were made to the sub-components of L2Switch-Main:

- Inventory Reader: We have modified the inventory reader such that if it cannot locate the Destination MAC address within the domain, it contacts the local agent application to fetch the location information (egress switch and port). Once the information is received, it is passed on to the Reactive Flow writer. The Reactive Flow writer or none of the other modules know that the address is remotely located. This is because the agent returns the border switch ID which is local to its own domain.

- Reactive Flow Writer: As mentioned before bidirectional flows are installed only when both the source and destination address are located in the same switch. We have modified the code here such that irrespective of where the destination is located corresponding bi-directional flows would be installed in the switches along the shortest path between the source and destination. We have implemented our *Flow Installation along the Shortest Path* algorithm in this module to achieve this. If the destination host is in the remote domain, then the flows will be installed only up to the border switch. Subsequently when the packet reaches the next domain, it will be handled by the corresponding domain controller and local agent.

Central Controller & Agents

The central controller application is a software program built using Python language. We have implemented this as two modules: 1) Loop Remover module 2) Shortest Path Computation module. Both of them use the *requests HTTP libraries* to make requests to OpenDayLight's RESTful APIs. They use HTTP GET request retrieves information from all the domain controllers through their respective RESTful API's. The response data is in JSON format which in turn is parsed to extract the required information (Nodes, Links and Hosts information). With the acquired information, the Loop Remover module implements the *topology building algorithm* and *loop removal algorithm.* It uses HTTP PUT requests to push the flows proactively via the controller, into the border switches, to build the spanning tree. The Shortest Path computation module implements *Dijkstra's shortest path algorithm* to compute the shortest path between a given pair of domains.

The agent is also written in Python language. It is invoked by its domain controller and is provided with the external host entry that the domain controller could not find in its own domain. The Agent connects to the central controller via SSH, invokes the controller application and provides it with the MAC information. The central controller then locates the entry associated with the MAC information, calculates the inter-domain level shortest path and returns the corresponding Border switches and Port IDs to the agent, which in turn returns the information to the domain controller.

4.3 Testbed Setup

The figure 4.1 depicts our testbed's set up. Following are the components and functionalities of each of the VM

- VM1 runs the Central Controller Application
- VMs 5, 6 and 7 run mininet application in each of them, simulating three different networks comprising open vswitches and virtual hosts
- VMs 2, 3 and 4 run Opendaylight and Agent application in each of them. They control the networks running in VMs 5, 6 and 7 respectively

VMs 1, 2, 3 and 4 have a single network adapter. All these virtual adapters are attached to a single "internal network" named Management network. The central controller and Opendaylight controllers communicate with each other through this network. VMs 5, 6 and 7 also have one of its adapters (eth2) attached to the Management network through which they communicate with their respective domain's Opendaylight controller.

VMs 5, 6 and 7 also have two additional adapters (eth0 and eth1) that are used to enable a layer 2 connection among themselves. These adapters are attached to LAN segments 1,2 or 3. To simulate a layer 2 point-to-point connection between a pair of domains, these adapters are attached to one of the open vswitches in their respective mininet topologies running on VMs 5, 6 and 7 as show in the Fig 4.2.

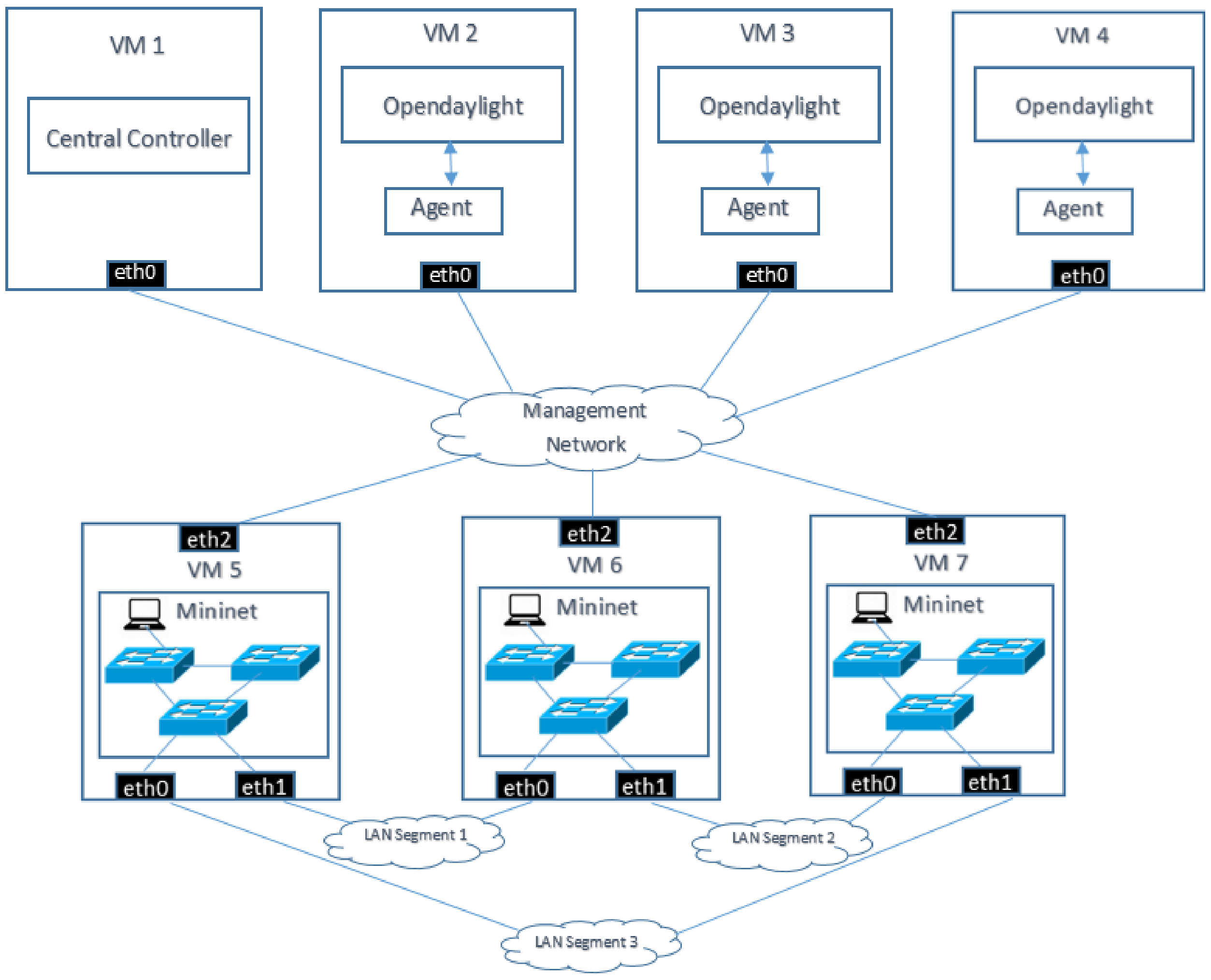


Figure 4. 2: Testbed Setup

To simulate a larger number of domains, the VM-'s' running mininet and Opendaylight just have to be replicated based on the number of domains that are required. In addition, we create a suitable number of point-to-point links for extra connections.

4.4 Experimental results and Analysis

Below figure shows the network topology that was simulated in our testbed. Since we had only 3 domains and each of them had only 3 devices we assumed only one digit each for representing the domain ID and the switch number. In the figure, switch ID “openflow:12”, the digit “1” represents domain ID and “2” represents switch number.

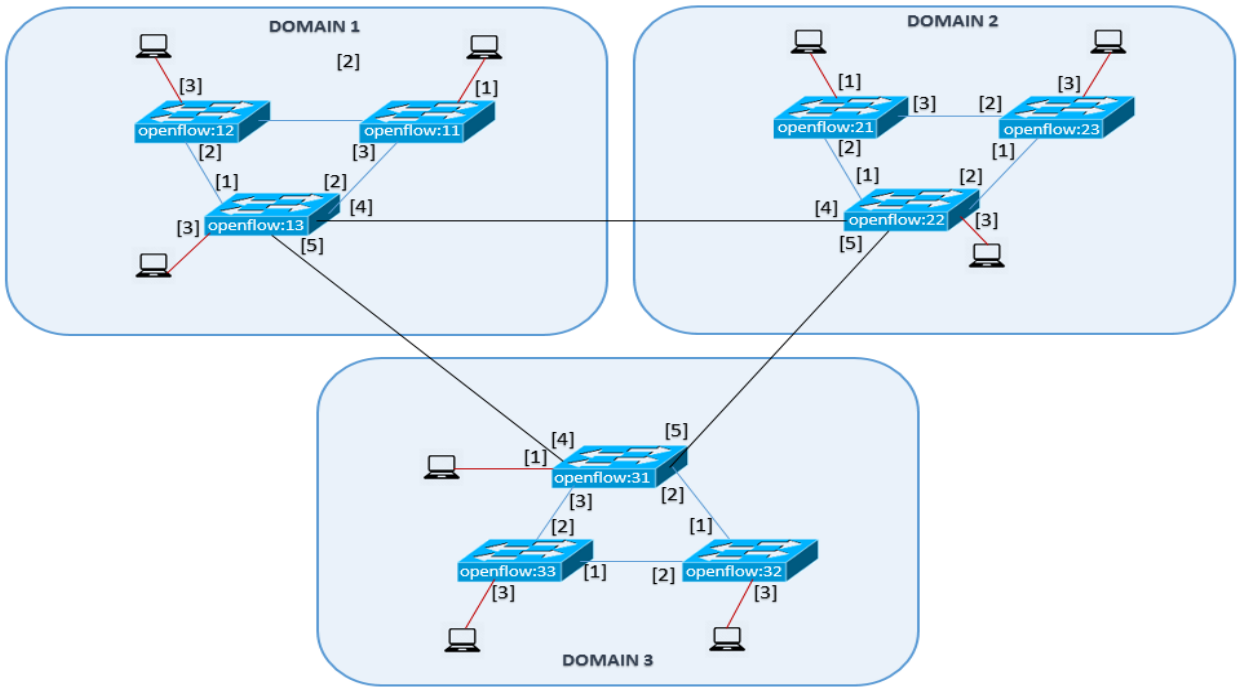
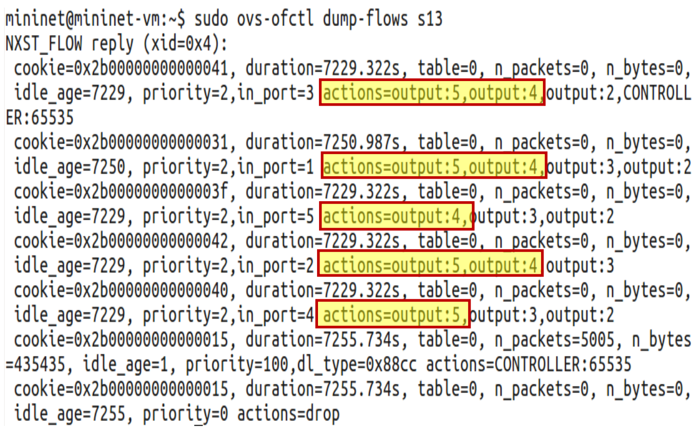


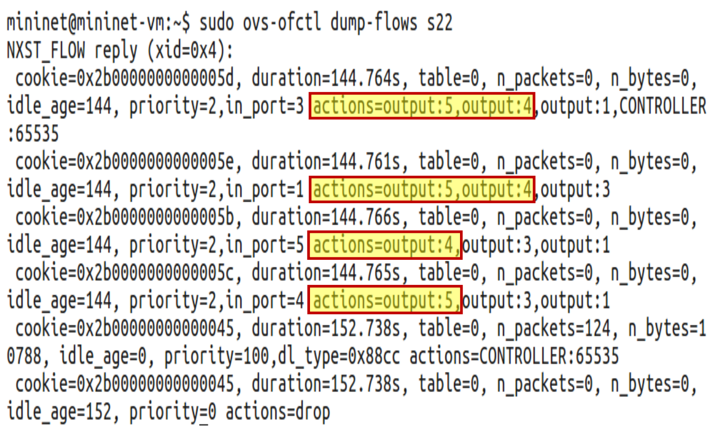
Figure 4. 3: Network Topology simulated in Testbed

Once the above devices got connected to their respective domain controllers (OpenDaylight), the controllers installed internal spanning trees to prevent loops within the domains. The flow tables of the border switches are shown in the tables 4.1a, 4.1b and 4.1c.



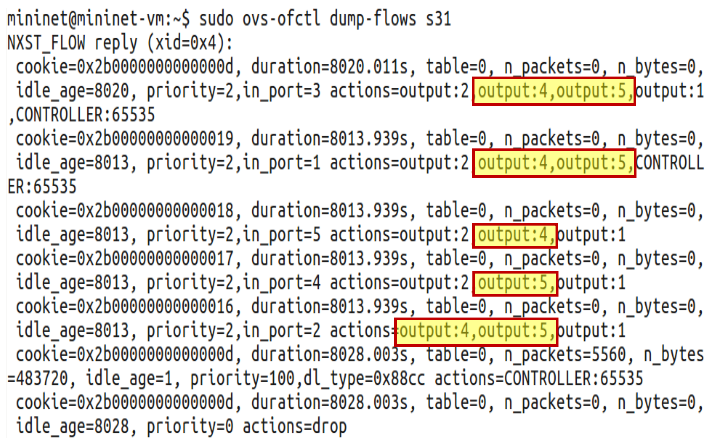
```
mininet@mininet-vm:~$ sudo ovs-ofctl dump-flows s13
NXST_FLOW reply (xid=0x4):
 cookie=0x2b00000000000041, duration=7229.322s, table=0, n_packets=0, n_bytes=0,
 idle_age=7229, priority=2,in_port=3 actions=output:5,output:4,output:2,CONTROLL
ER:65535
 cookie=0x2b00000000000031, duration=7250.987s, table=0, n_packets=0, n_bytes=0,
 idle_age=7250, priority=2,in_port=1 actions=output:5,output:4,output:3,output:2
 cookie=0x2b0000000000003f, duration=7229.322s, table=0, n_packets=0, n_bytes=0,
 idle_age=7229, priority=2,in_port=5 actions=output:4,output:3,output:2
 cookie=0x2b00000000000042, duration=7229.322s, table=0, n_packets=0, n_bytes=0,
 idle_age=7229, priority=2,in_port=2 actions=output:5,output:4 output:3
 cookie=0x2b00000000000040, duration=7229.322s, table=0, n_packets=0, n_bytes=0,
 idle_age=7229, priority=2,in_port=4 actions=output:5,output:3,output:2
 cookie=0x2b00000000000015, duration=7255.734s, table=0, n_packets=5005, n_bytes
=435435, idle_age=1, priority=100,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x2b00000000000015, duration=7255.734s, table=0, n_packets=0, n_bytes=0,
 idle_age=7255, priority=0 actions=drop
```

Table 4.1 e): Flow table - openflow:13



```
mininet@mininet-vm:~$ sudo ovs-ofctl dump-flows s22
NXST_FLOW reply (xid=0x4):
 cookie=0x2b0000000000005d, duration=144.764s, table=0, n_packets=0, n_bytes=0,
idle_age=144, priority=2,in_port=3 actions=output:5,output:4,output:1,CONTROLLER
:65535
 cookie=0x2b0000000000005e, duration=144.761s, table=0, n_packets=0, n_bytes=0,
idle_age=144, priority=2,in_port=1 actions=output:5,output:4,output:3
 cookie=0x2b0000000000005b, duration=144.766s, table=0, n_packets=0, n_bytes=0,
idle_age=144, priority=2,in_port=5 actions=output:4,output:3,output:1
 cookie=0x2b0000000000005c, duration=144.765s, table=0, n_packets=0, n_bytes=0,
idle_age=144, priority=2,in_port=4 actions=output:5,output:3,output:1
 cookie=0x2b00000000000045, duration=152.738s, table=0, n_packets=124, n_bytes=1
0788, idle_age=0, priority=100,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x2b00000000000045, duration=152.738s, table=0, n_packets=0, n_bytes=0,
idle_age=152, priority=0 actions=drop
```

Table 4.1 a): Flow table - openflow:22



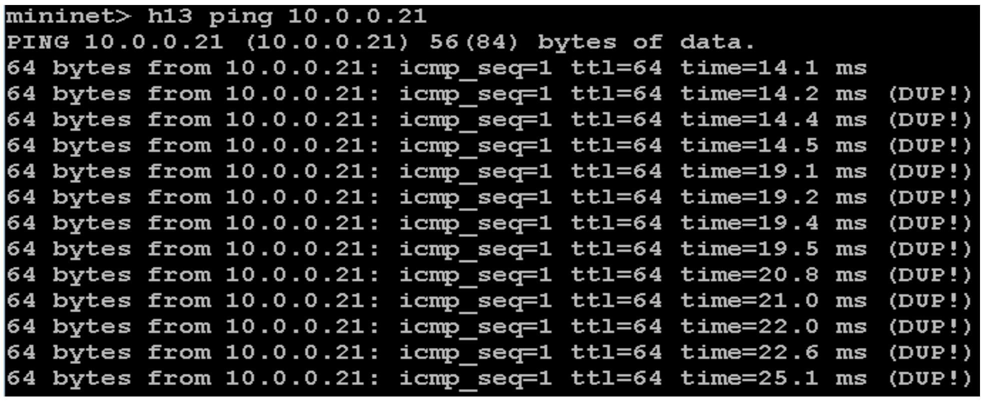
```
mininet@mininet-vm:~$ sudo ovs-ofctl dump-flows s31
NXST_FLOW reply (xid=0x4):
 cookie=0x2b0000000000000d, duration=8020.011s, table=0, n_packets=0, n_bytes=0,
 idle_age=8020, priority=2,in_port=3 actions=output:2,output:4,output:5,output:1
,CONTROLLER:65535
 cookie=0x2b00000000000019, duration=8013.939s, table=0, n_packets=0, n_bytes=0,
 idle_age=8013, priority=2,in_port=1 actions=output:2,output:4,output:5,CONTROLL
ER:65535
 cookie=0x2b00000000000018, duration=8013.939s, table=0, n_packets=0, n_bytes=0,
 idle_age=8013, priority=2,in_port=5 actions=output:2,output:4,output:1
 cookie=0x2b00000000000017, duration=8013.939s, table=0, n_packets=0, n_bytes=0,
 idle_age=8013, priority=2,in_port=4 actions=output:2,output:5,output:1
 cookie=0x2b00000000000016, duration=8013.939s, table=0, n_packets=0, n_bytes=0,
 idle_age=8013, priority=2,in_port=2 actions=output:4,output:5,output:1
 cookie=0x2b0000000000000d, duration=8028.003s, table=0, n_packets=5560, n_bytes
=483720, idle_age=1, priority=100,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x2b0000000000000d, duration=8028.003s, table=0, n_packets=0, n_bytes=0,
 idle_age=8028, priority=0 actions=drop
```

Table 4.1 i): Flow table - openflow:31

In the flow table of openflow:22 (Table 4.1b), there are no flows related to port 2 which is connected to port 1 of openflow:23. This is because the link between them is not a part of the internal spanning tree.

As highlighted in all tables above, flows are installed in all of them to forward traffic through the external ports 4 and 5 which are part of the inter-domain connectivity. Since none of them are blocked they form a loop. This is because their domain controllers are not aware of the domain level topology.

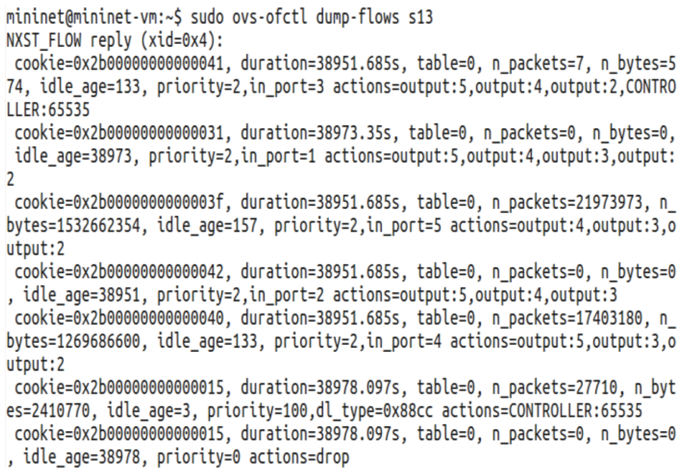
When we tried initiating a ping from host connected in openflow:13 (IP address 10.0.0.13/24) to host connected in openflow:21 (IP address 10.0.0.21/24) we found that the host received duplicate packets as expected.



```
mininet> h13 ping 10.0.0.21
PING 10.0.0.21 (10.0.0.21) 56(84) bytes of data.
64 bytes from 10.0.0.21: icmp_seq=1 ttl=64 time=14.1 ms
64 bytes from 10.0.0.21: icmp_seq=1 ttl=64 time=14.2 ms (DUP!)
64 bytes from 10.0.0.21: icmp_seq=1 ttl=64 time=14.4 ms (DUP!)
64 bytes from 10.0.0.21: icmp_seq=1 ttl=64 time=14.5 ms (DUP!)
64 bytes from 10.0.0.21: icmp_seq=1 ttl=64 time=19.1 ms (DUP!)
64 bytes from 10.0.0.21: icmp_seq=1 ttl=64 time=19.2 ms (DUP!)
64 bytes from 10.0.0.21: icmp_seq=1 ttl=64 time=19.4 ms (DUP!)
64 bytes from 10.0.0.21: icmp_seq=1 ttl=64 time=19.5 ms (DUP!)
64 bytes from 10.0.0.21: icmp_seq=1 ttl=64 time=20.8 ms (DUP!)
64 bytes from 10.0.0.21: icmp_seq=1 ttl=64 time=21.0 ms (DUP!)
64 bytes from 10.0.0.21: icmp_seq=1 ttl=64 time=22.0 ms (DUP!)
64 bytes from 10.0.0.21: icmp_seq=1 ttl=64 time=22.6 ms (DUP!)
64 bytes from 10.0.0.21: icmp_seq=1 ttl=64 time=25.1 ms (DUP!)
```

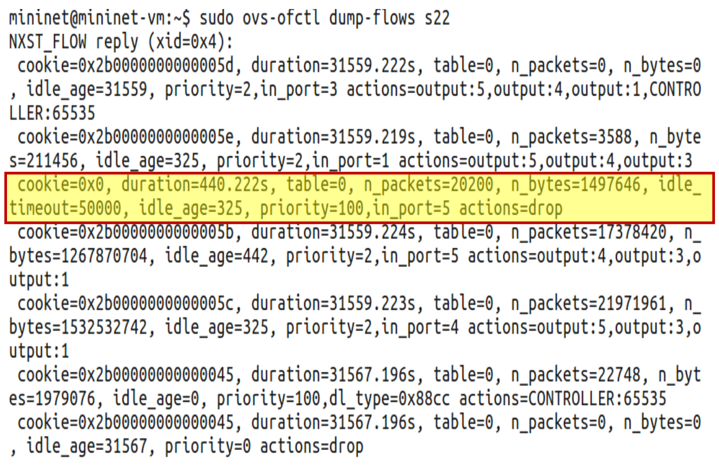
Figure 4. 4: Ping Results after initial setup

Below were the entries in the flow tables of the border switches after our central controller application's loop remover module was started.



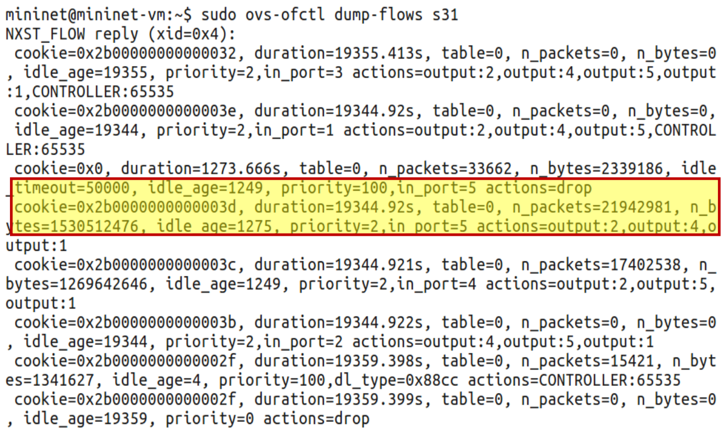
```
mininet@mininet-vm:~$ sudo ovs-ofctl dump-flows s13
NXST_FLOW reply (xid=0x4):
 cookie=0x2b00000000000041, duration=38951.685s, table=0, n_packets=7, n_bytes=5
74, idle_age=133, priority=2,in_port=3 actions=output:5,output:4,output:2,CONTRO
LLER:65535
 cookie=0x2b00000000000031, duration=38973.35s, table=0, n_packets=0, n_bytes=0,
 idle_age=38973, priority=2,in_port=1 actions=output:5,output:4,output:3,output:
2
 cookie=0x2b0000000000003f, duration=38951.685s, table=0, n_packets=21973973, n_
bytes=1532662354, idle_age=157, priority=2,in_port=5 actions=output:4,output:3,o
utput:2
 cookie=0x2b00000000000042, duration=38951.685s, table=0, n_packets=0, n_bytes=0
, idle_age=38951, priority=2,in_port=2 actions=output:5,output:4,output:3
 cookie=0x2b00000000000040, duration=38951.685s, table=0, n_packets=17403180, n_
bytes=1269686600, idle_age=133, priority=2,in_port=4 actions=output:5,output:3,o
utput:2
 cookie=0x2b00000000000015, duration=38978.097s, table=0, n_packets=27710, n_byt
es=2410770, idle_age=3, priority=100,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x2b00000000000015, duration=38978.097s, table=0, n_packets=0, n_bytes=0
, idle_age=38978, priority=0 actions=drop
```

Table 4.2 e) Flow Table - openflow:13



```
mininet@mininet-vm:~$ sudo ovs-ofctl dump-flows s22
NXST_FLOW reply (xid=0x4):
 cookie=0x2b0000000000005d, duration=31559.222s, table=0, n_packets=0, n_bytes=0
, idle_age=31559, priority=2,in_port=3 actions=output:5,output:4,output:1,CONTRO
LLER:65535
 cookie=0x2b0000000000005e, duration=31559.219s, table=0, n_packets=3588, n_byte
s=211456, idle_age=325, priority=2,in_port=1 actions=output:5,output:4,output:3
 cookie=0x0, duration=440.222s, table=0, n_packets=20200, n_bytes=1497646, idle_
timeout=50000, idle_age=325, priority=100,in_port=5 actions=drop
 cookie=0x2b0000000000005b, duration=31559.224s, table=0, n_packets=17378420, n_
bytes=1267870704, idle_age=442, priority=2,in_port=5 actions=output:4,output:3,o
utput:1
 cookie=0x2b0000000000005c, duration=31559.223s, table=0, n_packets=21971961, n_
bytes=1532532742, idle_age=325, priority=2,in_port=4 actions=output:5,output:3,o
utput:1
 cookie=0x2b00000000000045, duration=31567.196s, table=0, n_packets=22748, n_byt
es=1979076, idle_age=0, priority=100,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x2b00000000000045, duration=31567.196s, table=0, n_packets=0, n_bytes=0
, idle_age=31567, priority=0 actions=drop
```

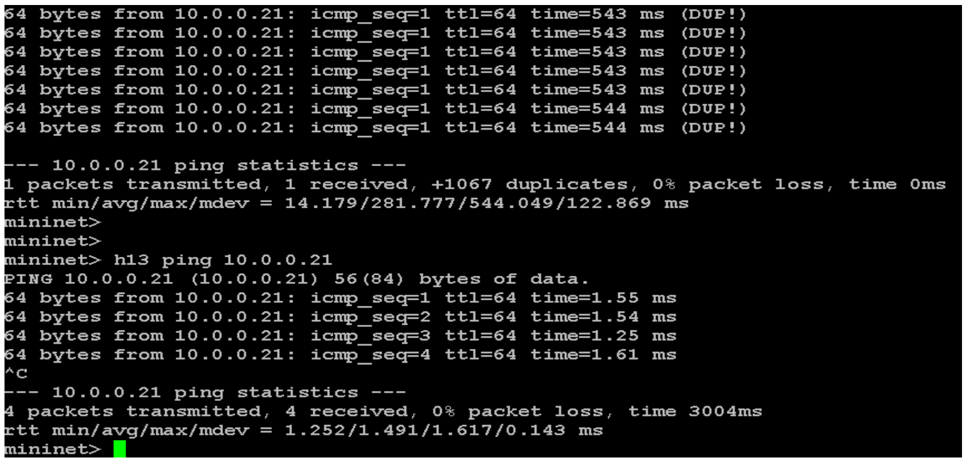
Table 4.2 a) Flow Table - openflow:22



```
mininet@mininet-vm:~$ sudo ovs-ofctl dump-flows s31
NXST_FLOW reply (xid=0x4):
 cookie=0x2b00000000000032, duration=19355.413s, table=0, n_packets=0, n_bytes=0
, idle_age=19355, priority=2,in_port=3 actions=output:2,output:4,output:5,output
:1,CONTROLLER:65535
 cookie=0x2b0000000000003e, duration=19344.92s, table=0, n_packets=0, n_bytes=0,
 idle_age=19344, priority=2,in_port=1 actions=output:2,output:4,output:5,CONTROL
LER:65535
 cookie=0x0, duration=1273.666s, table=0, n_packets=33662, n_bytes=2339186, idle
_timeout=50000, idle_age=1249, priority=100,in_port=5 actions=drop
 cookie=0x2b0000000000003d, duration=19344.92s, table=0, n_packets=21942981, n_b
ytes=1530512476, idle_age=1275, priority=2,in_port=5 actions=output:2,output:4,o
utput:1
 cookie=0x2b0000000000003c, duration=19344.921s, table=0, n_packets=17402538, n_
bytes=1269642646, idle_age=1249, priority=2,in_port=4 actions=output:2,output:5,
output:1
 cookie=0x2b0000000000003b, duration=19344.922s, table=0, n_packets=0, n_bytes=0
, idle_age=19344, priority=2,in_port=2 actions=output:4,output:5,output:1
 cookie=0x2b0000000000002f, duration=19359.398s, table=0, n_packets=15421, n_byt
es=1341627, idle_age=4, priority=100,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x2b0000000000002f, duration=19359.399s, table=0, n_packets=0, n_bytes=0
, idle_age=19359, priority=0 actions=drop
```

Table 4.2 i) Flow Table - openflow:31

The flow tables in openflow:22 (Table 4.2a) and openflow:31 (Table 4.2c) each had an additional entry to drop traffic on external port 5 after the loop removal module at the central controller is enabled. This is because, the link between domain 3 (openflow:31) and domain 2 (openflow:22) are not a part of the domain-level spanning tree. After a loop free topology was achieved, ping between the hosts were tested again and there were no duplication of packets this time. The ping results are show in the Figure 4.5



```
64 bytes from 10.0.0.21: icmp_seq=1 ttl=64 time=543 ms (DUP!)
64 bytes from 10.0.0.21: icmp_seq=1 ttl=64 time=543 ms (DUP!)
64 bytes from 10.0.0.21: icmp_seq=1 ttl=64 time=543 ms (DUP!)
64 bytes from 10.0.0.21: icmp_seq=1 ttl=64 time=543 ms (DUP!)
64 bytes from 10.0.0.21: icmp_seq=1 ttl=64 time=543 ms (DUP!)
64 bytes from 10.0.0.21: icmp_seq=1 ttl=64 time=544 ms (DUP!)
64 bytes from 10.0.0.21: icmp_seq=1 ttl=64 time=544 ms (DUP!)

--- 10.0.0.21 ping statistics ---
1 packets transmitted, 1 received, +1067 duplicates, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 14.179/281.777/544.049/122.869 ms
mininet>
mininet>
mininet> h13 ping 10.0.0.21
PING 10.0.0.21 (10.0.0.21) 56(84) bytes of data.
64 bytes from 10.0.0.21: icmp_seq=1 ttl=64 time=1.55 ms
64 bytes from 10.0.0.21: icmp_seq=2 ttl=64 time=1.54 ms
64 bytes from 10.0.0.21: icmp_seq=3 ttl=64 time=1.25 ms
64 bytes from 10.0.0.21: icmp_seq=4 ttl=64 time=1.61 ms
^C
--- 10.0.0.21 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3004ms
rtt min/avg/max/mdev = 1.252/1.491/1.617/0.143 ms
mininet>
```

Figure 4. 5: Ping Results after starting Loop Remover module

Thus the inter-domain communication had been enabled. But the communication took place via the spanning tree path which may not be optimal in all the cases. Fig 4.6 depicts the complete spanning tree topology (internal spanning tree + domain level spanning tree) built by the domain controllers and the central controller. The dotted blue lines represent the links blocked by domain controllers and the black dotted lines represent the link blocked by central controller. The arrows depict the sub-optimal path taken when host A (connected to openflow:32) communicated with host B (connected to openflow:23).

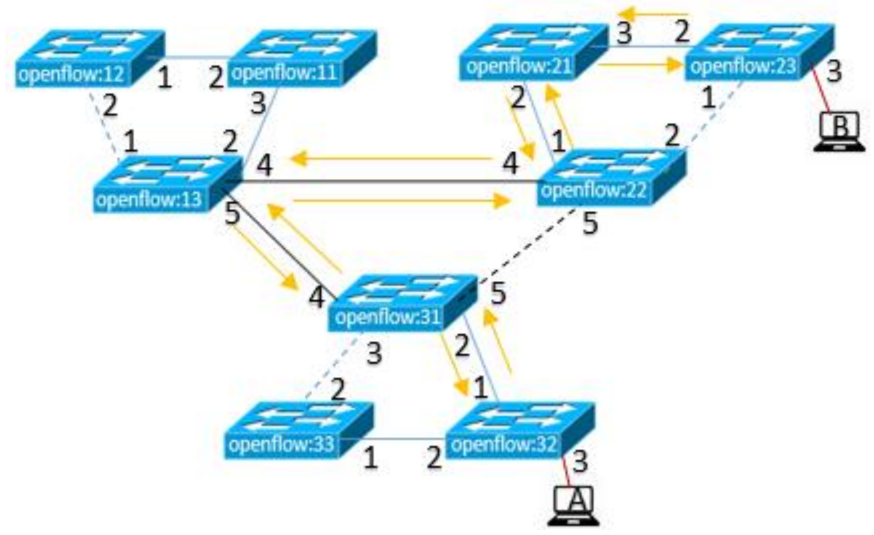
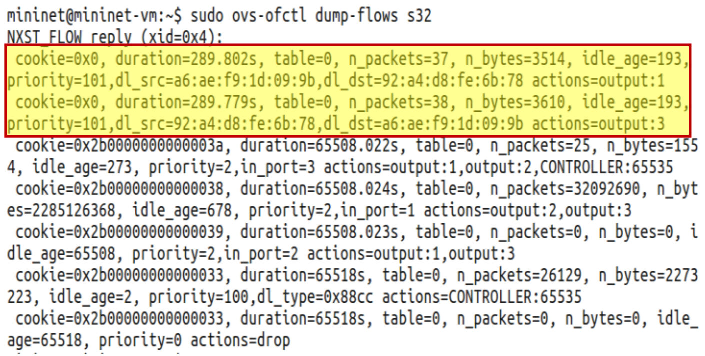


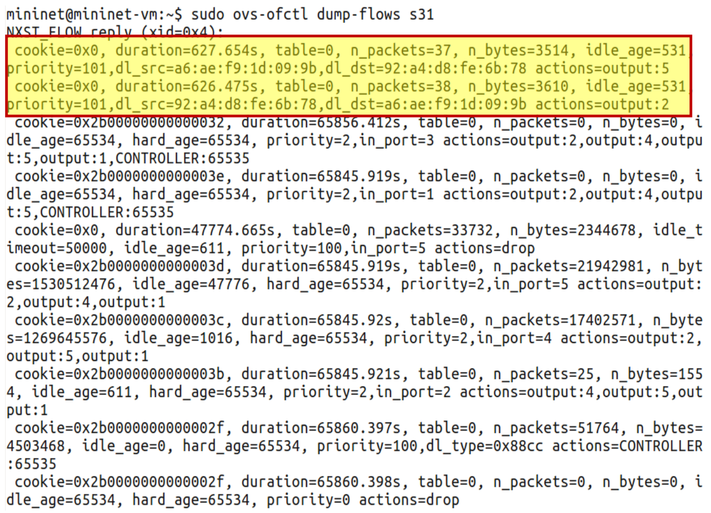
Figure 4. 6 Packet flow between host A and host B (after starting loop remover module)

After starting the central controller's shortest path computation module and their agents, we initiated ping traffic towards host B from host A. Bi-directional flows were dynamically installed in the switches along the shortest path. The following tables show the flow table entries of the switches involved in the shortest path.



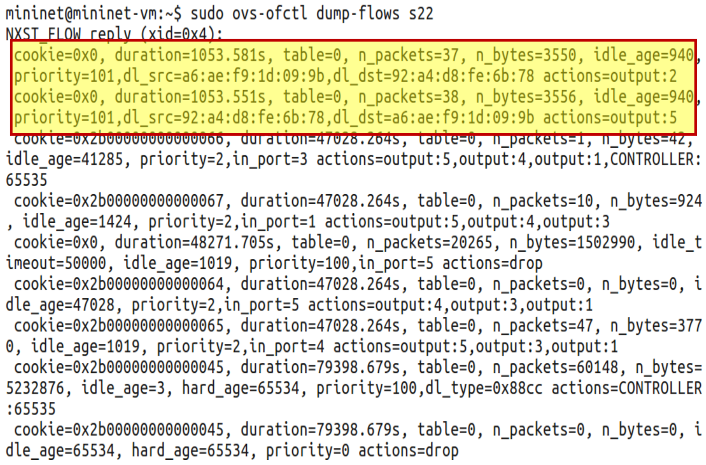
```
mininet@mininet-vm:~$ sudo ovs-ofctl dump-flows s32
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=289.802s, table=0, n_packets=37, n_bytes=3514, idle_age=193,
priority=101,dl_src=a6:ae:f9:1d:09:9b,dl_dst=92:a4:d8:fe:6b:78 actions=output:1
 cookie=0x0, duration=289.779s, table=0, n_packets=38, n_bytes=3610, idle_age=193,
priority=101,dl_src=92:a4:d8:fe:6b:78,dl_dst=a6:ae:f9:1d:09:9b actions=output:3
 cookie=0x2b0000000000003a, duration=65508.022s, table=0, n_packets=25, n_bytes=155
4, idle_age=273, priority=2,in_port=3 actions=output:1,output:2,CONTROLLER:65535
 cookie=0x2b00000000000038, duration=65508.024s, table=0, n_packets=32092690, n_byt
es=2285126368, idle_age=678, priority=2,in_port=1 actions=output:2,output:3
 cookie=0x2b00000000000039, duration=65508.023s, table=0, n_packets=0, n_bytes=0, i
dle_age=65508, priority=2,in_port=2 actions=output:1,output:3
 cookie=0x2b00000000000033, duration=65518s, table=0, n_packets=26129, n_bytes=2273
223, idle_age=2, priority=100,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x2b00000000000033, duration=65518s, table=0, n_packets=0, n_bytes=0, idle_
age=65518, priority=0 actions=drop
```

Table 4.3 c): Flow table – openflow:32



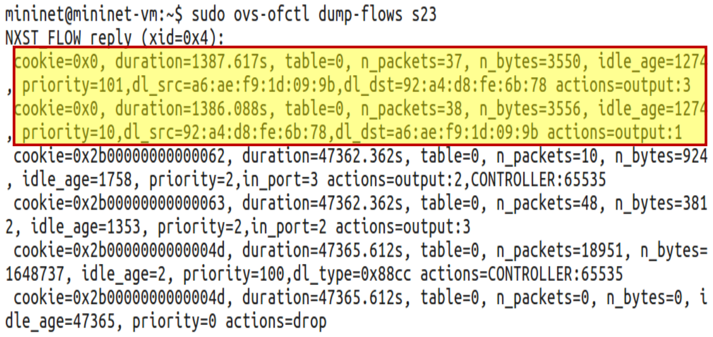
```
mininet@mininet-vm:~$ sudo ovs-ofctl dump-flows s31
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=627.654s, table=0, n_packets=37, n_bytes=3514, idle_age=531
priority=101,dl_src=a6:ae:f9:1d:09:9b,dl_dst=92:a4:d8:fe:6b:78 actions=output:5
 cookie=0x0, duration=626.475s, table=0, n_packets=38, n_bytes=3610, idle_age=531
priority=101,dl_src=92:a4:d8:fe:6b:78,dl_dst=a6:ae:f9:1d:09:9b actions=output:2
 cookie=0x2b00000000000032, duration=65856.412s, table=0, n_packets=0, n_bytes=0, i
dle_age=65534, hard_age=65534, priority=2,in_port=3 actions=output:2,output:4,outpu
t:5,output:1,CONTROLLER:65535
 cookie=0x2b0000000000003e, duration=65845.919s, table=0, n_packets=0, n_bytes=0, i
dle_age=65534, hard_age=65534, priority=2,in_port=1 actions=output:2,output:4,outpu
t:5,CONTROLLER:65535
 cookie=0x0, duration=47774.665s, table=0, n_packets=33732, n_bytes=2344678, idle_t
imeout=50000, idle_age=611, priority=100,in_port=5 actions=drop
 cookie=0x2b0000000000003d, duration=65845.919s, table=0, n_packets=21942981, n_byt
es=1530512476, idle_age=47776, hard_age=65534, priority=2,in_port=5 actions=output:
2,output:4,output:1
 cookie=0x2b0000000000003c, duration=65845.92s, table=0, n_packets=17402571, n_byte
s=1269645576, idle_age=1016, hard_age=65534, priority=2,in_port=4 actions=output:2,
output:5,output:1
 cookie=0x2b0000000000003b, duration=65845.921s, table=0, n_packets=25, n_bytes=155
4, idle_age=611, hard_age=65534, priority=2,in_port=2 actions=output:4,output:5,out
put:1
 cookie=0x2b0000000000002f, duration=65860.397s, table=0, n_packets=51764, n_bytes=
4503468, idle_age=0, hard_age=65534, priority=100,dl_type=0x88cc actions=CONTROLLER
:65535
 cookie=0x2b0000000000002f, duration=65860.398s, table=0, n_packets=0, n_bytes=0, i
dle_age=65534, hard_age=65534, priority=0 actions=drop
```

Table 4.3 b): Flow table – openflow:31



```
mininet@mininet-vm:~$ sudo ovs-ofctl dump-flows s22
NXST_FLOW reply (xid=0x4):
cookie=0x0, duration=1053.581s, table=0, n_packets=37, n_bytes=3550, idle_age=940,
priority=101,dl_src=a6:ae:f9:1d:09:9b,dl_dst=92:a4:d8:fe:6b:78 actions=output:2
cookie=0x0, duration=1053.551s, table=0, n_packets=38, n_bytes=3556, idle_age=940,
priority=101,dl_src=92:a4:d8:fe:6b:78,dl_dst=a6:ae:f9:1d:09:9b actions=output:5
 cookie=0x2b00000000000066, duration=47028.264s, table=0, n_packets=1, n_bytes=42,
idle_age=41285, priority=2,in_port=3 actions=output:5,output:4,output:1,CONTROLLER:
65535
 cookie=0x2b00000000000067, duration=47028.264s, table=0, n_packets=10, n_bytes=924
, idle_age=1424, priority=2,in_port=1 actions=output:5,output:4,output:3
 cookie=0x0, duration=48271.705s, table=0, n_packets=20265, n_bytes=1502990, idle_t
imeout=50000, idle_age=1019, priority=100,in_port=5 actions=drop
 cookie=0x2b00000000000064, duration=47028.264s, table=0, n_packets=0, n_bytes=0, i
dle_age=47028, priority=2,in_port=5 actions=output:4,output:3,output:1
 cookie=0x2b00000000000065, duration=47028.264s, table=0, n_packets=47, n_bytes=377
0, idle_age=1019, priority=2,in_port=4 actions=output:5,output:3,output:1
 cookie=0x2b00000000000045, duration=79398.679s, table=0, n_packets=60148, n_bytes=
5232876, idle_age=3, hard_age=65534, priority=100,dl_type=0x88cc actions=CONTROLLER
:65535
 cookie=0x2b00000000000045, duration=79398.679s, table=0, n_packets=0, n_bytes=0, i
dle_age=65534, hard_age=65534, priority=0 actions=drop
```

Table 4.3 a): Flow table – openflow:22

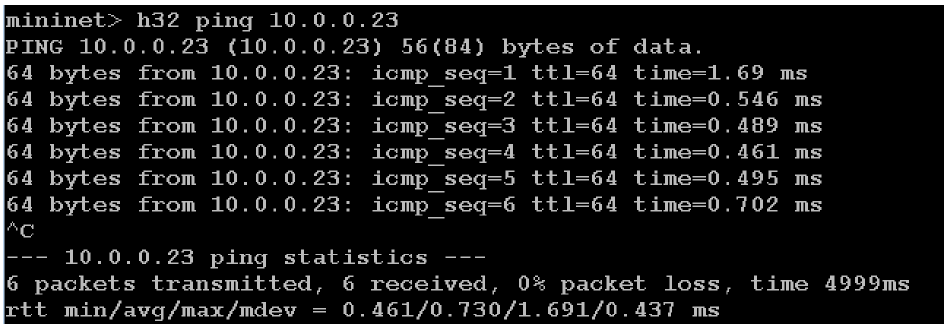


```
mininet@mininet-vm:~$ sudo ovs-ofctl dump-flows s23
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=1387.617s, table=0, n_packets=37, n_bytes=3550, idle_age=1274
, priority=101,dl_src=a6:ae:f9:1d:09:9b,dl_dst=92:a4:d8:fe:6b:78 actions=output:3
 cookie=0x0, duration=1386.088s, table=0, n_packets=38, n_bytes=3556, idle_age=1274
, priority=10,dl_src=92:a4:d8:fe:6b:78,dl_dst=a6:ae:f9:1d:09:9b actions=output:1
 cookie=0x2b00000000000062, duration=47362.362s, table=0, n_packets=10, n_bytes=924
, idle_age=1758, priority=2,in_port=3 actions=output:2,CONTROLLER:65535
 cookie=0x2b00000000000063, duration=47362.362s, table=0, n_packets=48, n_bytes=381
2, idle_age=1353, priority=2,in_port=2 actions=output:3
 cookie=0x2b0000000000004d, duration=47365.612s, table=0, n_packets=18951, n_bytes=
1648737, idle_age=2, priority=100,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x2b0000000000004d, duration=47365.612s, table=0, n_packets=0, n_bytes=0, i
dle_age=47365, priority=0 actions=drop
```

Table 4.3 d) Flow table – openflow:23

In the highlighted portion of Table 4.3a, we can see that any packet from host A (address: a6:ae:f9:1d:09:9b) to host B (address: 92:a4:d8:fe:6b:78) will be sent out on port 3 of switch "openflow32". When the packet arrives at switch "openflow31", it will be sent out on port 5 (Table 4.3b). Thus the flow follows the shortest path as shown in Fig. 4.8.

The ping result is show in Fig 4.7 shows, the time taken to receive ICMP replies packets had then decreased by 3 times when compared to sub-optimal path results show in figure 4.5.



```
mininet> h32 ping 10.0.0.23
PING 10.0.0.23 (10.0.0.23) 56(84) bytes of data.
64 bytes from 10.0.0.23: icmp_seq=1 ttl=64 time=1.69 ms
64 bytes from 10.0.0.23: icmp_seq=2 ttl=64 time=0.546 ms
64 bytes from 10.0.0.23: icmp_seq=3 ttl=64 time=0.489 ms
64 bytes from 10.0.0.23: icmp_seq=4 ttl=64 time=0.461 ms
64 bytes from 10.0.0.23: icmp_seq=5 ttl=64 time=0.495 ms
64 bytes from 10.0.0.23: icmp_seq=6 ttl=64 time=0.702 ms
^C
--- 10.0.0.23 ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 4999ms
rtt min/avg/max/mdev = 0.461/0.730/1.691/0.437 ms
```

Figure 4. 7 Ping Results (After starting the agents and shortest path computation module)

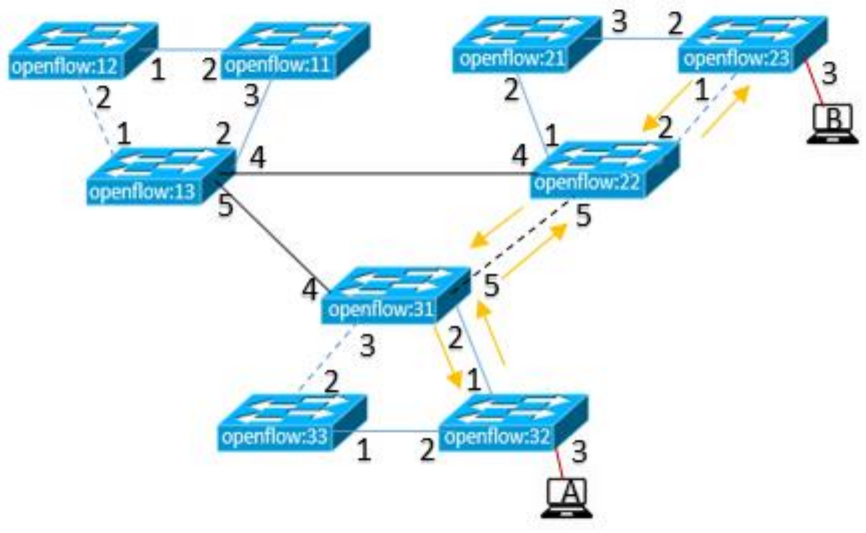


Figure 4. 8 Packet flow between host A and host B (after starting Shortest Path computation module)

4.5 Load balancing implementation and results

To demonstrate the flexibility and openness of our proposed approach, we add a simple load balancing mechanism in our implementation. The central controller runs a load balancing algorithm to select one of the equal-cost paths, thus one of the border switches. In our implementation, we have modified the Dijkstra's Algorithm to return all equal-cost paths. The equal-cost paths are then numbered consecutively starting with 1. If the number of equal-cost paths is M, then the path is chosen based on the below formula

Path number = **mod** (Hash (source MAC address + destination MAC address), M) + 1

In other words, we use a hash of the source and destination MAC addresses to select one of the equal-cost paths. Other more sophisticated load balancing algorithms can be used in the future.

Since our test-bed topology had only 3 domains, there was only one shortest path (or least-cost path) between any two domains if the costs of all links are equal. To test our load-balancing implementation, we create two equal-cost paths between domains 1 and 3 by assigning the cost of 2 to the link (bidirectional) connecting domain 1 and 3 and the cost of 1 to the links connecting domains 1 and 2 and domains 2 and 3 (Fig 4.9). Thus, a flow between hosts in domains 1 and 3 can choose either the path directly connecting domains 1 and 3 or the path through domain 2.

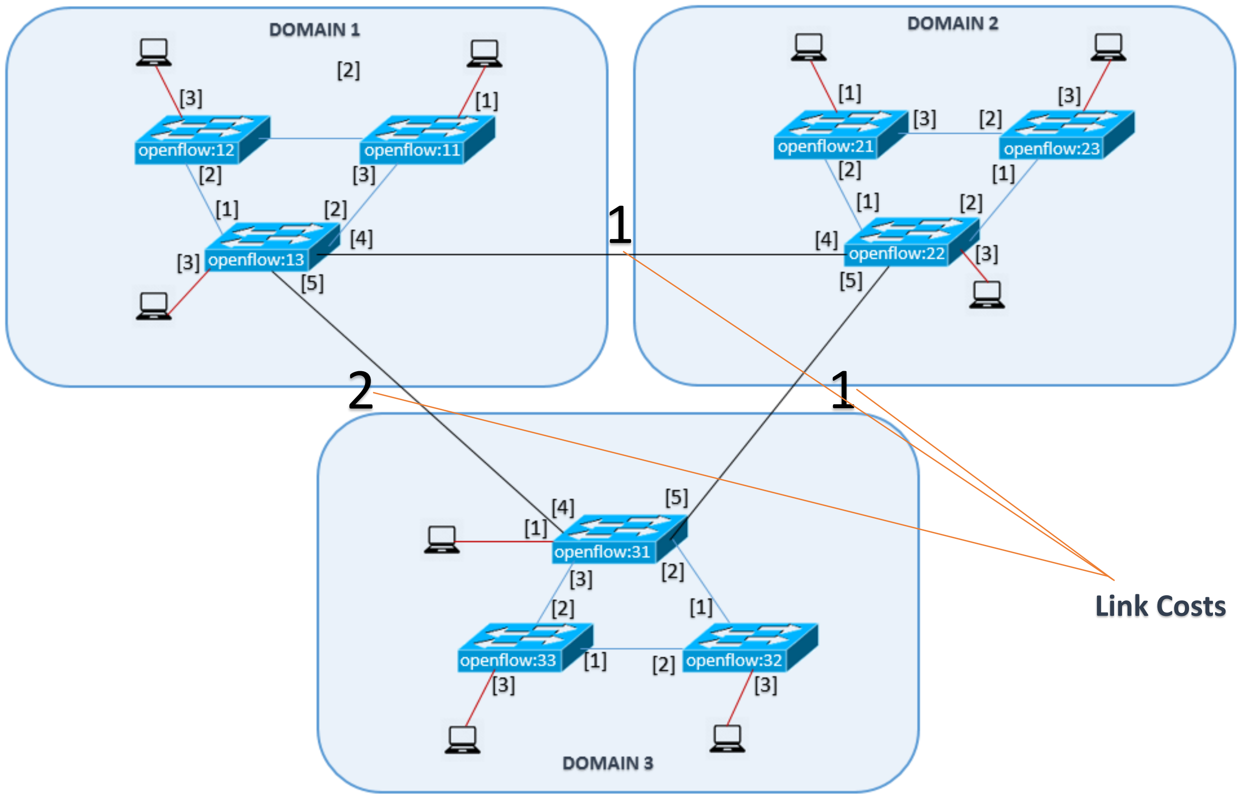


Figure 4. 9 Network Topology simulated in Testbed

Figures 4.10a and 4.10b shows path taken during communication between host A and C, and host A and D once the load balancer module was enabled.

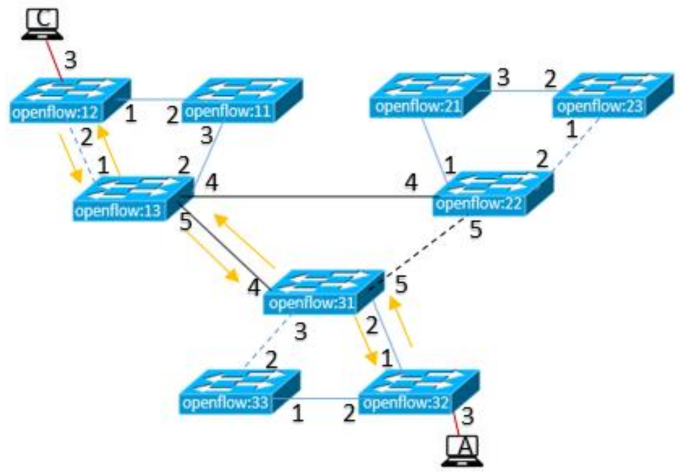


Figure 4.10 b) Packet flow between host A and C

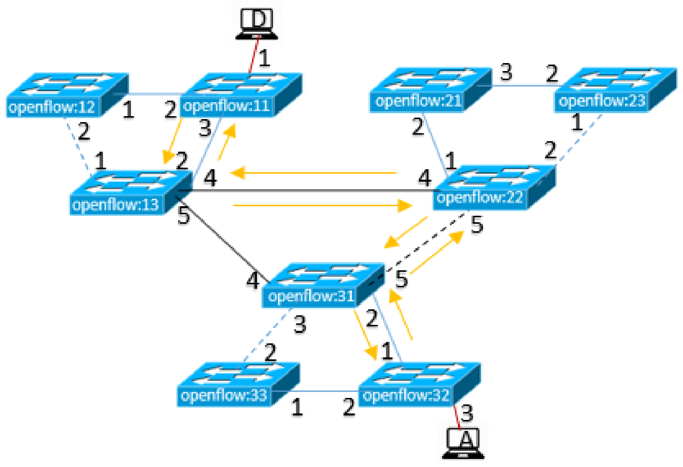
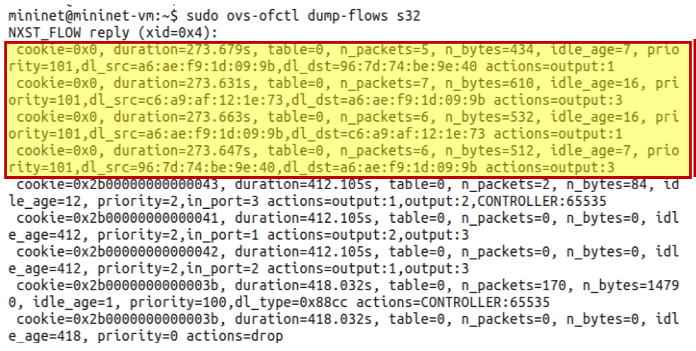


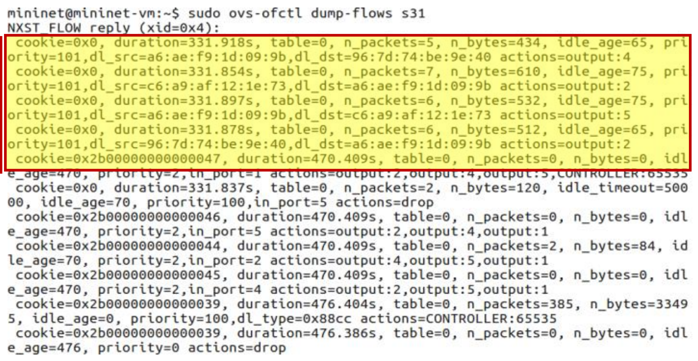
Figure 4.10 a) Packet flow between host A and D

The following tables show the flow table entries of the switches involved in the communication path between A to C and A to D.



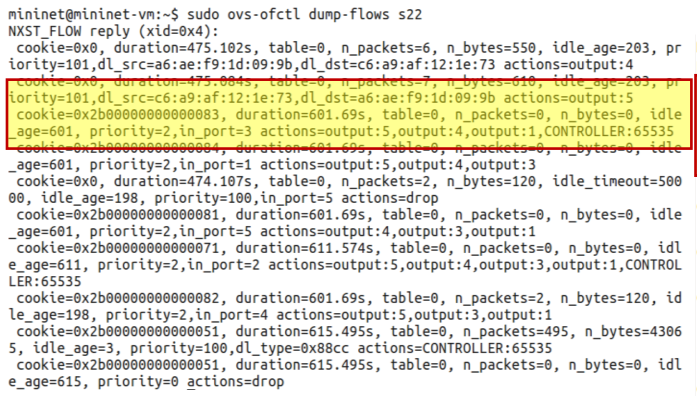
```
mininet@mininet-vm:~$ sudo ovs-ofctl dump-flows s32
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=273.679s, table=0, n_packets=5, n_bytes=434, idle_age=7, prio
rity=101,dl_src=a6:ae:f9:1d:09:9b,dl_dst=96:7d:74:be:9e:40 actions=output:1
 cookie=0x0, duration=273.631s, table=0, n_packets=7, n_bytes=610, idle_age=16, pri
ority=101,dl_src=c6:a9:af:12:1e:73,dl_dst=a6:ae:f9:1d:09:9b actions=output:3
 cookie=0x0, duration=273.663s, table=0, n_packets=6, n_bytes=532, idle_age=16, pri
ority=101,dl_src=a6:ae:f9:1d:09:9b,dl_dst=c6:a9:af:12:1e:73 actions=output:1
 cookie=0x0, duration=273.647s, table=0, n_packets=6, n_bytes=512, idle_age=7, prio
rity=101,dl_src=96:7d:74:be:9e:40,dl_dst=a6:ae:f9:1d:09:9b actions=output:3
 cookie=0x2b00000000000043, duration=412.105s, table=0, n_packets=2, n_bytes=84, id
le_age=12, priority=2,in_port=3 actions=output:1,output:2,CONTROLLER:65535
 cookie=0x2b00000000000041, duration=412.105s, table=0, n_packets=0, n_bytes=0, idl
e_age=412, priority=2,in_port=1 actions=output:2,output:3
 cookie=0x2b00000000000042, duration=412.105s, table=0, n_packets=0, n_bytes=0, idl
e_age=412, priority=2,in_port=2 actions=output:1,output:3
 cookie=0x2b0000000000003b, duration=418.032s, table=0, n_packets=170, n_bytes=1479
0, idle_age=1, priority=100,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x2b0000000000003b, duration=418.032s, table=0, n_packets=0, n_bytes=0, idl
e_age=418, priority=0 actions=drop
```

Table 4.4 a) Flow table – openflow:32



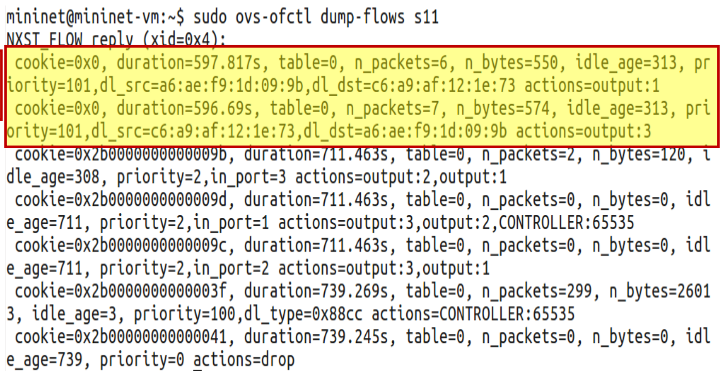
```
mininet@mininet-vm:~$ sudo ovs-ofctl dump-flows s31
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=331.918s, table=0, n_packets=5, n_bytes=434, idle_age=65, pri
ority=101,dl_src=a6:ae:f9:1d:09:9b,dl_dst=96:7d:74:be:9e:40 actions=output:4
 cookie=0x0, duration=331.854s, table=0, n_packets=7, n_bytes=610, idle_age=75, pri
ority=101,dl_src=c6:a9:af:12:1e:73,dl_dst=a6:ae:f9:1d:09:9b actions=output:2
 cookie=0x0, duration=331.897s, table=0, n_packets=6, n_bytes=532, idle_age=75, pri
ority=101,dl_src=a6:ae:f9:1d:09:9b,dl_dst=c6:a9:af:12:1e:73 actions=output:5
 cookie=0x0, duration=331.878s, table=0, n_packets=6, n_bytes=512, idle_age=65, pri
ority=101,dl_src=96:7d:74:be:9e:40,dl_dst=a6:ae:f9:1d:09:9b actions=output:2
 cookie=0x2b00000000000047, duration=470.409s, table=0, n_packets=0, n_bytes=0, idl
e_age=470, priority=2,in_port=1 actions=output:2,output:4,output:5,CONTROLLER:65535
 cookie=0x0, duration=331.837s, table=0, n_packets=2, n_bytes=120, idle_timeout=500
00, idle_age=70, priority=100,in_port=5 actions=drop
 cookie=0x2b00000000000046, duration=470.409s, table=0, n_packets=0, n_bytes=0, idl
e_age=470, priority=2,in_port=5 actions=output:2,output:4,output:1
 cookie=0x2b00000000000044, duration=470.409s, table=0, n_packets=2, n_bytes=84, id
le_age=70, priority=2,in_port=2 actions=output:4,output:5,output:1
 cookie=0x2b00000000000045, duration=470.409s, table=0, n_packets=0, n_bytes=0, idl
e_age=470, priority=2,in_port=4 actions=output:2,output:5,output:1
 cookie=0x2b00000000000039, duration=476.404s, table=0, n_packets=385, n_bytes=3349
5, idle_age=0, priority=100,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x2b00000000000039, duration=476.386s, table=0, n_packets=0, n_bytes=0, idl
e_age=476, priority=0 actions=drop
```

Table 4.4 b) Flow table – openflow:31



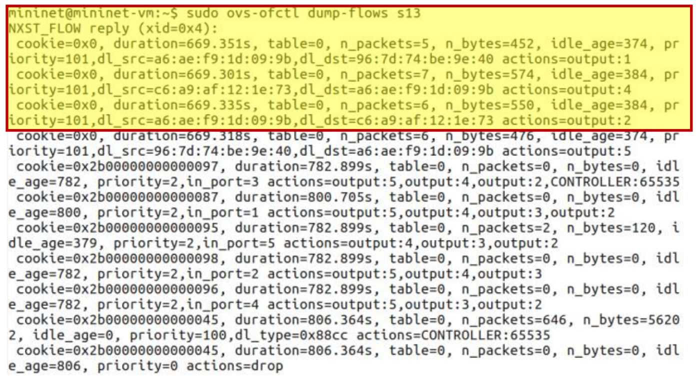
```
mininet@mininet-vm:~$ sudo ovs-ofctl dump-flows s22
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=475.102s, table=0, n_packets=6, n_bytes=550, idle_age=203, pr
iority=101,dl_src=a6:ae:f9:1d:09:9b,dl_dst=c6:a9:af:12:1e:73 actions=output:4
 cookie=0x0, duration=475.084s, table=0, n_packets=7, n_bytes=610, idle_age=203, pr
iority=101,dl_src=c6:a9:af:12:1e:73,dl_dst=a6:ae:f9:1d:09:9b actions=output:5
 cookie=0x2b00000000000083, duration=601.69s, table=0, n_packets=0, n_bytes=0, idle
_age=601, priority=2,in_port=3 actions=output:5,output:4,output:1,CONTROLLER:65535
 cookie=0x2b00000000000084, duration=601.69s, table=0, n_packets=0, n_bytes=0, idle
_age=601, priority=2,in_port=1 actions=output:5,output:4,output:3
 cookie=0x0, duration=474.107s, table=0, n_packets=2, n_bytes=120, idle_timeout=500
00, idle_age=198, priority=100,in_port=5 actions=drop
 cookie=0x2b00000000000081, duration=601.69s, table=0, n_packets=0, n_bytes=0, idle
_age=601, priority=2,in_port=5 actions=output:4,output:3,output:1
 cookie=0x2b00000000000071, duration=611.574s, table=0, n_packets=0, n_bytes=0, idl
e_age=611, priority=2,in_port=2 actions=output:5,output:4,output:3,output:1,CONTROL
LER:65535
 cookie=0x2b00000000000082, duration=601.69s, table=0, n_packets=2, n_bytes=120, id
le_age=198, priority=2,in_port=4 actions=output:5,output:3,output:1
 cookie=0x2b00000000000051, duration=615.495s, table=0, n_packets=495, n_bytes=4306
5, idle_age=3, priority=100,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x2b00000000000051, duration=615.495s, table=0, n_packets=0, n_bytes=0, idl
e_age=615, priority=0 actions=drop
```

Table 4.4 d) Flow table – openflow:22



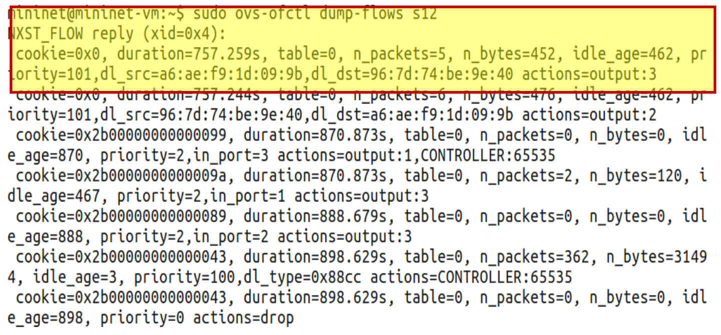
```
mininet@mininet-vm:~$ sudo ovs-ofctl dump-flows s11
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=597.817s, table=0, n_packets=6, n_bytes=550, idle_age=313, pr
iority=101,dl_src=a6:ae:f9:1d:09:9b,dl_dst=c6:a9:af:12:1e:73 actions=output:1
 cookie=0x0, duration=596.69s, table=0, n_packets=7, n_bytes=574, idle_age=313, pri
ority=101,dl_src=c6:a9:af:12:1e:73,dl_dst=a6:ae:f9:1d:09:9b actions=output:3
 cookie=0x2b0000000000009b, duration=711.463s, table=0, n_packets=2, n_bytes=120, i
dle_age=308, priority=2,in_port=3 actions=output:2,output:1
 cookie=0x2b0000000000009d, duration=711.463s, table=0, n_packets=0, n_bytes=0, idl
e_age=711, priority=2,in_port=1 actions=output:3,output:2,CONTROLLER:65535
 cookie=0x2b0000000000009c, duration=711.463s, table=0, n_packets=0, n_bytes=0, idl
e_age=711, priority=2,in_port=2 actions=output:3,output:1
 cookie=0x2b0000000000003f, duration=739.269s, table=0, n_packets=299, n_bytes=2601
3, idle_age=3, priority=100,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x2b00000000000041, duration=739.245s, table=0, n_packets=0, n_bytes=0, idl
e_age=739, priority=0 actions=drop
```

Table 4.4 c) Flow table – openflow:11



```
mininet@mininet-vm:~$ sudo ovs-ofctl dump-flows s13
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=669.351s, table=0, n_packets=5, n_bytes=452, idle_age=374, pr
iority=101,dl_src=a6:ae:f9:1d:09:9b,dl_dst=96:7d:74:be:9e:40 actions=output:1
 cookie=0x0, duration=669.301s, table=0, n_packets=7, n_bytes=574, idle_age=384, pr
iority=101,dl_src=c6:a9:af:12:1e:73,dl_dst=a6:ae:f9:1d:09:9b actions=output:4
 cookie=0x0, duration=669.335s, table=0, n_packets=6, n_bytes=550, idle_age=384, pr
iority=101,dl_src=a6:ae:f9:1d:09:9b,dl_dst=c6:a9:af:12:1e:73 actions=output:2
 cookie=0x0, duration=669.318s, table=0, n_packets=6, n_bytes=476, idle_age=374, pr
iority=101,dl_src=96:7d:74:be:9e:40,dl_dst=a6:ae:f9:1d:09:9b actions=output:5
 cookie=0x2b00000000000097, duration=782.899s, table=0, n_packets=0, n_bytes=0, idl
e_age=782, priority=2,in_port=3 actions=output:5,output:4,output:2,CONTROLLER:65535
 cookie=0x2b00000000000087, duration=800.705s, table=0, n_packets=0, n_bytes=0, idl
e_age=800, priority=2,in_port=1 actions=output:5,output:4,output:3,output:2
 cookie=0x2b00000000000095, duration=782.899s, table=0, n_packets=2, n_bytes=120, i
dle_age=379, priority=2,in_port=5 actions=output:4,output:3,output:2
 cookie=0x2b00000000000098, duration=782.899s, table=0, n_packets=0, n_bytes=0, idl
e_age=782, priority=2,in_port=2 actions=output:5,output:4,output:3
 cookie=0x2b00000000000096, duration=782.899s, table=0, n_packets=0, n_bytes=0, idl
e_age=782, priority=2,in_port=4 actions=output:5,output:3,output:2
 cookie=0x2b00000000000045, duration=806.364s, table=0, n_packets=646, n_bytes=5620
2, idle_age=0, priority=100,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x2b00000000000045, duration=806.364s, table=0, n_packets=0, n_bytes=0, idl
e_age=806, priority=0 actions=drop
```

Table 4.4 f) Flow table openflow:13



```
mininet@mininet-vm:~$ sudo ovs-ofctl dump-flows s12
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=757.259s, table=0, n_packets=5, n_bytes=452, idle_age=462, pr
iority=101,dl_src=a6:ae:f9:1d:09:9b,dl_dst=96:7d:74:be:9e:40 actions=output:3
 cookie=0x0, duration=757.244s, table=0, n_packets=6, n_bytes=476, idle_age=462, pr
iority=101,dl_src=96:7d:74:be:9e:40,dl_dst=a6:ae:f9:1d:09:9b actions=output:2
 cookie=0x2b00000000000099, duration=870.873s, table=0, n_packets=0, n_bytes=0, idl
e_age=870, priority=2,in_port=3 actions=output:1,CONTROLLER:65535
 cookie=0x2b0000000000009a, duration=870.873s, table=0, n_packets=2, n_bytes=120, i
dle_age=467, priority=2,in_port=1 actions=output:3
 cookie=0x2b00000000000089, duration=888.679s, table=0, n_packets=0, n_bytes=0, idl
e_age=888, priority=2,in_port=2 actions=output:3
 cookie=0x2b00000000000043, duration=898.629s, table=0, n_packets=362, n_bytes=3149
4, idle_age=3, priority=100,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x2b00000000000043, duration=898.629s, table=0, n_packets=0, n_bytes=0, idl
e_age=898, priority=0 actions=drop
```

Table 4.4 e) Flow table openflow:12

Let's concentrate on the flow table of switch openflow:31 (Table 4.4b), which is the border switch of domain 3. For the flow from host A to host C (address: 96:7d:74:be:9e:40), the output port is 4, which connects to domain 1 directly. On the other hand, for the flow from host A to host D (address:

c6:a9:af:12:1e:73), the output port is 5, which connects to domain 2. The results show that both equal-cost paths are utilized for the inter-domain communications between domains 1 and 3.

4.6 Concluding Remarks

In this chapter, we show the results of the implementation of our proposed approach for SDN inter-domain switching. The results demonstrate that the implementation is successful and the proposed approach is a viable method for inter-domain switching.

Chapter 5

Conclusion and Future Work

In this thesis we proposed and approach to achieve layer 2 communication between SDN domains. We achieved inter-domain switching by taking the vertical approach. In our proposal there is a central controller, logically on top of all the domain controllers. The central controller communicates with the domain controllers (through the RESTFul API interface) and acquires all the information required to enable inter-domain communication.

Using the acquired information, the central controller constructs an abstract domain-level topology and computes spanning tree for layer 2 loop removal. Based on the spanning tree results it blocks the ports which are not a part of the tree. Following the topology discovery and loop removal, we deal with two approaches to achieve inter-domain communication via the shortest path.

The first approach requires few changes to be made in the domain controller. Apart from this, agents (which communicate with central controller) need to be installed in each of the domain controllers. Each agent maintains a local database which is used for inter-domain traffic forwarding. This database can be easily expanded with multiple fields to perform more advanced traffic engineering. The second approach does not require any changes to be made in the domain controller. The central controller directly updates the local database of the domain controller, but this approach is proactive and not traffic driven.

Our implementation was based on the first approach. We validated our algorithms by implementing them in a test environment that comprised of 3 domains connected in a full mesh topology and controlled by ODL controllers.

Following are the some of the shortcomings that were noticed and suggestions on how they can be improved

- In our approach, since we used the first two digits of the switch ID to represent the domain ID, all the switches in a domain must be carefully named (manually). In future a mechanism could be devised in the central controller to distinguish the domains in an efficient way which is independent of any configuration in the switches or the domain controllers.

- The central controller’s focus is only on achieving shortest path in the abstracted domain-level topology, the complete path i.e the intra-domain shortest path + inter-domain shortest path may not always be the overall shortest path. To overcome this, the central controller can construct the complete topology and calculate the best path based on that.

References

[1] Amin Tootoonchian and Yashar Ganjali. HyperFlow: A Distributed Control Plane for OpenFlow. Conference: Proceedings of the 2010 internet network management conference on Research on enterprise networking, pages 3-3.

[2] Deepankar Gupta and Rafat Jahan. Inter-SDN Controller Communication: Using Border Gateway Protocol. Opendaylight Summit 2015.

[3] Soheil Hassas Yeganeh and Yashar Ganjali. Kandoo: A Framework for Efficient and Scalable Offloading of Control Applications. ACM SIGCOMM 2012, pages 19-24.

[4] DCI (Datacentre Interconnect): THE NEED FOR STRETCHED LAYER 2 https://www.packetmischief.ca/2013/04/09/dci-the-need-for-stretched-layer-2/

[5] Nobutaka Matsumoto and Michiaki Hayashi. Scalable OpenFlow Controller Redundancy Tackling Local and Global Recoveries. Keisuke Kuroki. The Fifth International Conference on Advances in Future Internet AFIN 2013, pages 61 - 66.

[6] H. Yin, H. Xie, T. Tsou, D. Lopez, P. A. Aranda, and R. Sidi. SDNi: A message exchange protocol for software defined networks (SDNs) across multiple domains. (IRTF Internet-Draft), draft-yin-sdn-sdni-00, 2012.

[7] Pingping Lin Jun Bi* Ze Chen Yangyang Wang Hongyu Hu Anmin Xu. WE-Bridge: West-East Bridge for SDN Inter-domain Network Peering. IEEE INFOCOM'2014, pages 111 – 112.

[8] Pavol Helebrandt, Ivan Kotuliak. Novel SDN Multi-domain Architecture. ICETA 2014 -12th IEEE International Conference on Emerging eLearning Technologies and Applications, pages 139 – 143.

[9] Kevin Phemius, Mathieu Bouet, Jeremie Leguay. DISCO : Distributed SDN Controllers in a Multi-domain Environment. NOMS 2014 - 2014 IEEE/IFIP Network Operations and Management Symposium, pages 1-2.

[10] Ze Chen, Jun Bi, Yonghong Fu, Yangyang Wang, Anmin Xu. MLV: A Multi-dimension Routing Information Exchange Mechanism for Inter-domain SDN. ISN: 1092-1648. 2015 IEEE 23rd International Conference on Network Protocols, pages 438 – 445.

[11] Yangyang Wang; Jun Bi; Keyao Zhang; Yangchun Wu SDI: a multi-domain SDN mechanism for fine-grained inter-domain routing, 2016 Eighth International Conference on Ubiquitous and Future Networks (ICUFN) pages 751 – 756.

[12] Chen Chen, Bo Li, Dong Lin, and Baochun Li. Software-Defined Inter-Domain Routing Revisited. 2016 IEEE International Conference on Communications (ICC), pages 1 – 6.

[13] Data Center Interconnect: Layer 2 Extension between Remote Data Centers, Cisco, Document ID:1457308692080625, Jan, 2014.

[14] Openvswitch
http://openvswitch.org/pipermail/discuss/2015-April/017283.html

[15] An Architecture for Network Management Using NETCONF and YANG, IETF, RFC6244, ISSN: 2070-1721.

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

[17] Software-Defined Networking (SDN): Layers and Architecture Terminology, RFC7426, Page 7, ISSN: 2070-1721.

[18] OpenFlow Switch Specification
https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.1.pdf

[19] OpenDaylight Wiki
https://wiki.opendaylight.org/view/Main_Page

[20] Lu You; Li Wei; Luo Junzhou; Jiang Jian; Xia Nu.An inter-domain multi-path flow transfer mechanism based on SDN and multi-domain collaboration. 2015 IFIP/IEEE International Symposium on Integrated Network Management (IM), pages 758-761.

[21] Lin P, Bi J, Wang Y. WEBridge: west-east bridge for distributed heterogeneous SDN NOSes peering. 2014 IEEEConference on Computer Communications Workshops, pages 111 – 112.

[22] Pingping Lin; Jun Bi; Stephen Wolff; Yangyang Wang; Anmin Xu; Ze Chen;Hongyu Hu; Yikai Lin. A west-east bridge based SDN inter-domain testbed. IEEE Communications Magazine (Volume: 53, Issue: 2, Feb. 2015), pages 190 – 197.

[23] Wun-Yuan Huang; Ta-Yuan Chou; Jen-Wei Hu; Te-Lung Liu. Automatical End to End Topology Discovery and Flow Viewer on SDN. Advanced Information Networking and Applications Workshops (WAINA), 2014 28th International Conference on, pages 910-915.