Design of SDN Oriented Policy-Based Slicing in Virtualized Service Provider Networks

A Project Report Presented to

The faculty of the Department of Electrical Engineering

San Jose State University

In Partial Fulfillment

Of the Requirements for the Degree

Master of Science

By
Sohail Virani
009426404
sohail.virani@sjsu.edu
(408)-315-7080

Goutham Prasanna
009400404
goutham.prasanna@sjsu.edu
(408)-455-3902

Department Approval

Dr. Balaji Venkatraman
Project Advisor

Date

Project Advisor

Dr. Nader F.Mir
Project Co-Advisor

Date

Dr.Thuy Le
Graduate Advisor

Date

Graduate Advisor

Department of Electrical Engineering
Charles W. Davidson College of
Engineering San Jose State University
San Jose, CA 95192-0084

ABSTRACT

In the last decade or so our networks have gone through a transitional phase wherein we see an huge increase in network bandwidths, improved network performance and a greater network usage. This in turn has given rise to an on-demand culture where in multitude of users want unlimited access to the network. With service providers at the helm of it all have two main goals, to attract most customers by providing services at competitive rates and to operate their networks as efficiently as possible while adhering to customer's requirements. Hence, different specialized applications like virtual private networks, virtual private lines, virtual LANs were developed to cater to this growing need. Although there was an increase in network utilization efficiency the design of the networks were still unchanged and over time that has led to a present day scenario wherein our networks are widely distributed, complex and highly unmanageable. Therefore, there is a need for the design of future networks to be more flexible, manageable and efficient. Service providers could look towards SDN oriented virtualization as a scalable solution to achieve their goals without having to completely change their infrastructure and hence making their networks easily manageable and enhancing the overall quality of service. However, much research has to be done to apply such concepts to a multi-layered service provider network.

ACKNOWLEDGEMENT

We would like to express our deep gratitude to Dr. Balaji Venkatraman, our research supervisor for agreeing to be part of the project. We would like to thank him for his guidance, enthusiastic encouragement, patience and useful critiques of this research work.

We would also like to thank Dr. Nader F. Mir and Dr. Thuy T. Le our project co-advisor and graduate advisor for always being there for us at time of difficulty and pointing us in the right way.

We would also like to extend our warm regards to Dr. Greg M. Bernstein, for his advice and assistance in the beginning of the research to narrow down the research domain.

Finally, we would like to thank the department of electrical engineering at SJSU for providing us with necessary facilities and timely approvals for successful completion of the project.

Table of Contents

List of Figures

CHAPTER - 1

1.1 Introduction

In today's age, it is of prime importance that all devices and gadgets are connected to the internet, transferring Exabyte's of Data each day. This digital leap has rendered our networks congested, rigid and unmanageable, ultimately leading to network ossification. So, there is a need for a scalable solution that can make our networks more flexible, customizable and dynamically configurable. Software Defined Network (SDN) holds great promise in terms of simplifying network manageability by separating control plane from the data plane. Hence, lowering the total operational cost of managing enterprise in carrier networks by allowing active programmability of network services. However, one fundamental challenge of SDN is to handle performance, programmability and flexibility simultaneously in a multi-layered service provider network. Here, performance refers to the throughput and latency of the network node while programmability is the ability to adapt systems to support and deploy new and unique features dynamically. This paper talks about designing a SDN oriented policy-based slicing in a virtualized multi-tenant service provider network that promises to allocate part of the network to customers based on on-demand service model in a way that allows multiple customers to use the same underlying infrastructure to run multiple distinct applications simultaneously.

The present day service providers have been successful over the past few decades in supporting multitude of distributed applications and a wide variety of network technologies. However, this feature of service provider networks has become impediment to its further growth. Due to its multi-layered and distributed architecture, adopting a new design or modification to the existing one is a becoming increasingly difficult leading to ossification. It is believed that network virtualization along with SDN will eradicate this ossification in service provider networks and improve their overall network performance, ease of manageability and ultimately pave way for new technological innovations in the field of internetworking.

1.2 Network Virtualization

Network virtualization has become a go-to domain for researchers when describing projects based on active-programmable networks or private virtual networks. In simple terms it can be referred to as isolation of the multiple virtual networks and their co-existence over the same physical network infrastructure. These virtual networks are a slice of the underlying physical network and consists of a collection of virtual nodes and virtual links that provide end to end connectivity. This ability can be used by service providers to automatically configure and deploy services across their network in a timely manner. Therefore, once this virtualization is achieved the only thing left to do is packet forwarding which resembles the operation of data plane in SDN. This also enables us to use multiple independent virtual networks over the same physical hardware resulting in improved network performance and resource utilization besides lowering operational costs [7]. Furthermore, these multiple virtual networks have advantage of features like network sharing, resource sharing, and isolation, having exclusive control over their network which in turn enables innovation.

The above mentioned features allow service providers to operate on multiple heterogeneous virtual networks that are isolated from each other and yet share the same physical resources. The concept of multiple co-existing networks have been around for a long time and can be broadly classified into four classes: virtual local area networks, virtual private networks, active and programmable networks and overlay networks.

Virtual Local Area Network

A virtual local network (VLAN) is a group of logical networked hosts which come under a single broadcast domain irrespective of their physical connectivity. Within a particular VLAN, the frames are tagged with same VLAN ID. The switches that handle these frames must be VLAN enabled and make use of both MAC address and the VLAN ID to process these frames. Since VLANs are just logical separation, the administration, reconfiguration and management become extremely easy than physical migration. Moreover, VLANs enhance the security of LANs.

Virtual Private Networks

A virtual private network (VPN) is a dedicated network connecting networks at multiple location into a single network using private and secure tunnels instead of shared and transparent communication channels such as Internet. Usually, VPNs connect multiple geographically separated network over secure tunnels. Each location consists of multiple Customer Edge (CE) devices that are connected to either one or more Provider Edge (PE) routers.

Based on the protocol used to achieve tunnels in the data plane, VPNS can be further classified into –

- **Layer 1 VPN** – This level of VPN is required to extend the more classical L2/L3 VPNs to circuit switched domains.
- **Layer 2 VPN** – Layer 2 VPNs transport the frames only between the participating sites, as it imposes no restrictions on the Layer 3 payload, it is very flexible. However, the connectivity across the VPNs cannot be achieved due to the absence of the control plane.
- **Layer 3 VPN** – As the name suggests this uses Layer 3 protocols in the VPN backbone to carry data between the distributed customer edges. There are further two types of L3

VPNs. In Customer Edge (CE) based approach, the Provider Edge devices do not participate in the VPN configuration and hence are not aware of the existence of the service. In the Provider Edge (PE) based approach, the PE device determine the VPN configuration and management. The connected CE devices may behave as though they are directly connected to the private network.

Active and Programmable Networks

Active and programmable networks provide programmability and promote concepts of isolated environments that allow multiple parties to run mutually conflicting codes on the same network nodes without causing instability. This can be implemented in two ways-

- **Open Signaling Approach:** It defines an abstraction layer for physical network devices so that they can act as distributed computing environments. They must contain well-defined open programming interfaces that will allow service providers to manipulate these devices when required.
- **Active Networks Approach:** Active networks allow the customization of network services based on the contents of the active packets transport granularity and offer more flexibility and granularity than the open signaling approach. However, this is a complex model.

Overlay Networks

It is a logical network built on top of one or more existing physical networks. In the existing networks, overlays are typically implemented at the application layer, however, they can also be implemented on lower layers of the networking stack.

Overlay networks do not make any changes to the underlying physical network. Consequently, overlays provide a relatively easy and inexpensive means to deploy new features within the network. A multitude of application layer overlay designs address issues such as high performance, multicasting, Quality of Service (QoS) and certain Bandwidth guarantees. Overlays are also used as testbeds to design and evaluate new architectures. Below figure showcases how multiple service providers use network virtualization in their networks.

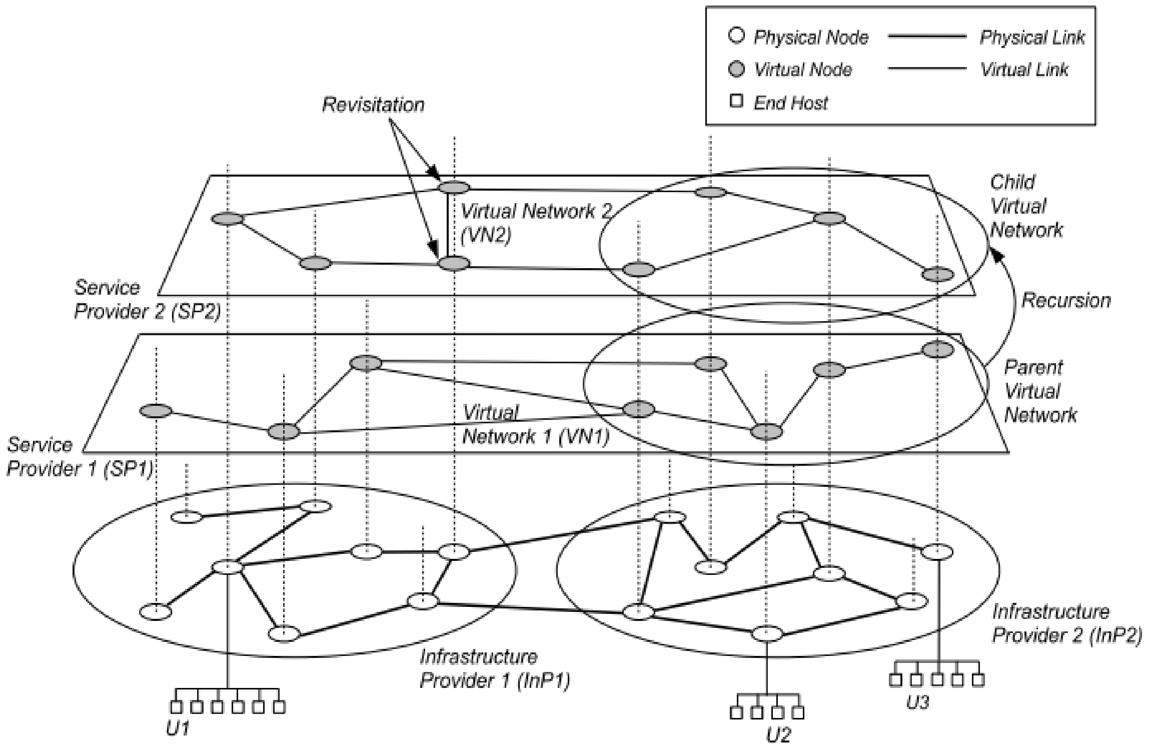


Figure 1. Network Virtualization

Service Providers achieve this level of virtualization by combining the concepts of Overlay Networks and Network Function Virtualization into a collective and cohesive technology. It not only supports running of multiple overlay networks on top of each other but also allows for the virtualization of the network functions that ensure that services that were initially running on the physical network now run seamlessly on the virtual network. To achieve this, these networks must have certain characteristics that are detailed below.

- **Isolation.** Each virtual network must be isolated from it neighboring virtual networks with which it shares the network resources. This is absolutely crucial to ensure that information from one virtual network does not leak into the other thus causing a security issue or a conflict in the network behavior. Logical isolations based on address space and protocols provide an ease of administration while physical isolation based on physical layer provides for very high security. VIOLIN and UCLP are examples for logical and physical isolations respectively.
- **Flexibility and Heterogeneity.** The virtual networks must be high flexible to accommodate for frequent dynamic changes. This allows the service providers to dynamically adapt to the customer demands. At the same time, they must also be highly heterogeneous to accommodate wide contrast of network applications that run on each service provider.
- **Scalable.** The virtual networks must be programmable to dynamically grow and shrink based on the demands of the service provider. With the increasing provider network traffic, service providers must be able to either expand their virtual networks or fall back to earlier design without administrative intervention and heavy configuration changes.

- **Resource Allocation and Scheduling.** The virtual networks share the same physical resources. Hence, it is imperative that every network resource allows for scheduling the packets coming from every virtual network running on it. Also, every resource must allocate minimum guaranteed storage and compute capabilities to ensure packets from different virtual networks are guaranteed attendance and do not starve or get declined.

1.3 Software Defined Networking

Ever-increasing traffic volume, an emphasis on achieving high network reliability and the need to cater to the widening spectrum of customer demands from the network have constantly annoyed the service providers. Some of their approach at dealing with these problems such as carefully controlling the paths to deliver traffic have been primitive at best. The close interaction of the control and the data planes within routers and switches made it extremely challenging to perform network management tasks.

Software Defined Networking (SDN) is an approach that decouples these control and data planes by abstracting the higher level network applications from the lower level traffic forwarding functions. The software-based control plane allows the implementation of wide range of network services over different software platforms. At the same time, the architecture at the data plane is greatly simplified as it does not handle any control decisions. This decoupling of the planes promotes the horizontal integration model that is desired by the current service providers [4] [1] [6]. By separating the control device, called the Controller, and the traffic handling devices such as switches and routers, it allows the controller to dynamically change the network behavior towards incoming traffic and not be worried about the actual forwarding action. There are two ways in which the above action can be achieved:

- *An open interface to communicate between the control and data planes.* This open interface allows the control unit and all the network nodes to speak with each other. Further, standardization of this interface allows for a single control unit to handle all the devices simultaneously. One such example is the OpenFlow standard.

- *A logical and centralized control unit for the entire network.* The single control unit now maintains a complete view of the network and allows for greater flexibility in managing and controlling how the networking nodes at the data plane operate.

Through SDN, service providers can now globally control and monitor the performance of all their network nodes. This centralization greatly eases service provider's ability to manage and troubleshoot their networks that may be customized as per their specific requirements. At the same time, the simplified data plane operation allows these providers to eliminate inter-operability issues by maintaining an infrastructure with similar data plane nodes.

1.3.1 Components

Some of the important components of the SDN are as follows:-

- **Network Application.** These applications are responsible for generating the services desired from a network by communicating their requirements and the expected network behavior to the controller using pre-defined Application Level Interfaces (APIs). These applications have an abstracted view of the network that allows for decision-making.

- **Controller.** It is the logical unit that separates the network services from the underlying network devices. Its major functionality includes (i) translating the requirements received from the SDN Applications down to the SDN data plane and (ii) providing the abstract view of the SDN data plane up to the SDN applications.

- **Datapath.** It is a network device that is responsible for forwarding the packets based on the rules set in its forwarding table. It advertises its forwarding capabilities to the SDN controller suitably modifies them to set rules on how to deal with further incoming traffic.

- **Southbound Interface.** This is the interface defined between the SDN Controller and the underlying data plane nodes. It provides (i) capabilities to handle all forwarding operations (ii) feature to advertise device capabilities and (iii) trigger event notifications.

- **Northbound Interface.** This is the interface through which the network applications communicate with the SDN Controller. A wide set of APIs can be defined to allow the controller to understand specifications provided from various SDN applications.

- **Management Plane.** This includes the subset of network applications that leverage the northbound APIs to establish network-wide control and maintain operational logic.

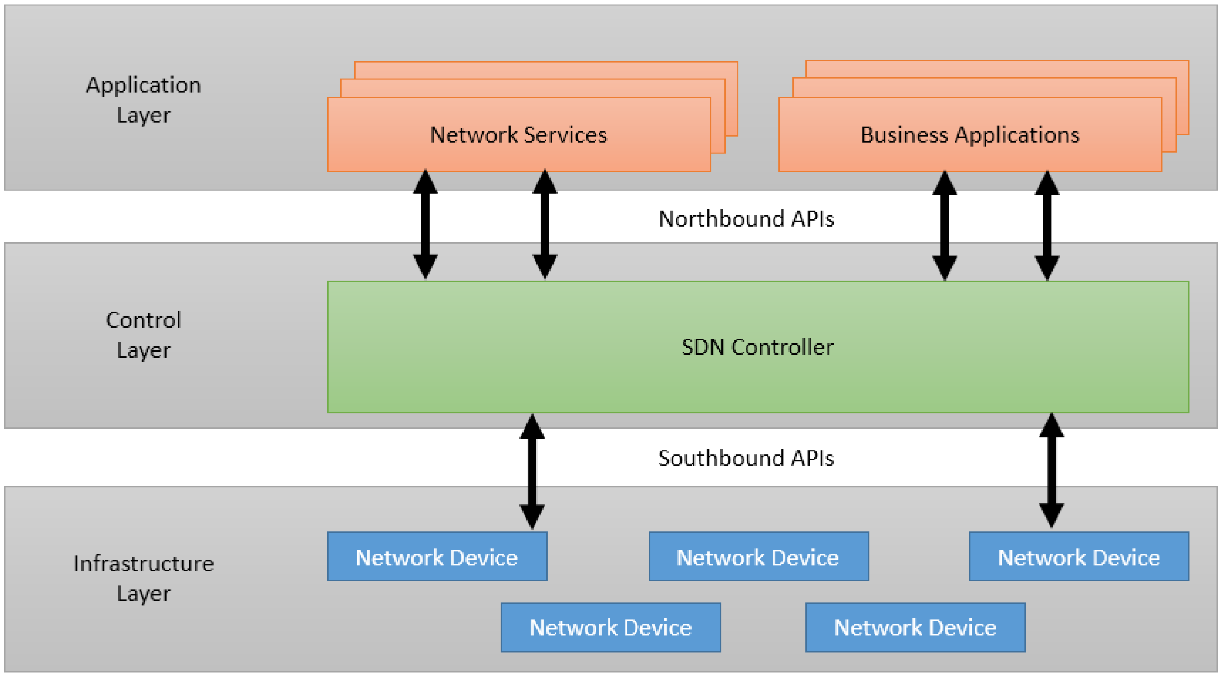


Figure 2. Architecture of Software Defined Networks

1.3.2 Operation Overview

The SDN data plane devices only forward packets based on their forwarding table entries and do not include any decision making capabilities. However, all the network traffic passes through them. Whenever a new packet arrives at an open switch, it checks its forwarding table for any rules associated with the concerned packet based on its packet header information. If a suitable rule is found, the associated actions are taken for that packet and a flow rule is established. Otherwise, it queries the SDN controller for the action it must take using the SDN Southbound interfaces.

The SDN controller is the sole decision making device in the network. It receives all the queries from the underlying data plane devices. It suitably translates and sends them to the network applications that are waiting for this information using the Northbound APIs. Each network application must register with the SDN controller and define the network information that it requires during the registration. This allows the Controller to select the right network applications and prevents any leakage of valuable network information to unsubscribed applications.

The network application running on the controller analyze the information received from it and performs the necessary decision-making logic defined in them. The corresponding action to be taken is then informed back to the controller using the same northbound APIs. The controller performs the necessary translation to suitably modifying it into a flow rule that is understood by the underlying data plane and then forwards it to them using the southbound APIs. This flow rule is stored by the devices residing in the data plane for further packets pertaining to same flow that it may receive and performs the associated action.

1.4 Quality of Service

Quality of Service refers to the capability of the network to provide some additional transmission characteristics to the existing network connections. Accordingly, based on the subscription of connection to any class of service based on service level agreements (SLA) its Quality of Service parameters would then include either one or more from the following set-

- Throughput or Minimal transmission rate
- Packet Loss Ratio
- Delay in packet
- Variation in packet delay (or Jitter)

Service providers have to deal with wide range of network applications that demand different subsets of above parameter set. Very stringent Quality of Service requirements are generally associated with Real-Time traffic, which require very low packet loss, low packet delay and minimal delay jitter. On the other hand, data transmission requires assurance of minimal guaranteed bit rate. To provide these parameters to the traffic, two models are defined –

1.4.1 Integrated Services

Integrated services model (or) IntServ relies on some type of reservation protocol for negotiating transmission parameters and allocating the resources accordingly. This is usually the job of a Resource Reservation Protocol (RSVP) protocol. The prime requirement is that the negotiation and reservation be conducted for each connection or either a group of such connections.

Broadly classified there are three classes of service:

- **Best Effort:** This does not assure any guarantees.
- **Controller-load Service:** This is applicable for low delay and loss sensitive applications and is better than best effort service.
- **Guaranteed Service:** For application that require assured bandwidth and has stringent delay constraints, this is generally the preferred class.

IntServ makes use of end-to-end signally, followed by reservation and admission to provision the above class of services. It also places some constraints that all the network devices in the intermediate paths must also support RSVP protocol. Admission control is required at each hop and simultaneously the reservation are periodically refreshed. It houses a packet scheduler that is responsible for forwarding of the packets at each node based on the reserved class. All these constraints put on the network places an additional overhead on all the network nodes. IntServ has thus proven to work in small networks but becomes increasing complex with lot of overhead for large networks.

1.4.2 <u>Differentiated Services</u>

Differentiated Services (DiffServ) offers different types of services for the transmitted traffic. Its architecture consists of edge routers and DiffServ enabled core routers. Core routers differentiate only few predefined number of traffic classes based on Class of Service. Flows are aggregated within the network and the Quality of Service is provided for each class and not for individual flows. It has two major components: packet marking and PHBs.

To mark the Class of Service six bits in the IP header are allocated and used. The queuing, shaping and scheduling behaviors of all the intermediate nodes is referred to as Per Hop Behaviors (PHBs). Edge routers are responsible for policy matching and packet marking while the core routers process all the incoming packets based on the policies defined in them and the markings on the packet.

Since all the packet marking occurs at the edge routers, the core routers must be able to provide the necessary class of service. In addition, the simplification of the working of the core routers speeds up the process, decreases the overhead and makes it more scalable. Since DiffServ uses only the class of service and does not store any flow information, it cannot provide QoS guarantees.

To provide real time link sharing services at the same time, many hierarchical scheduling algorithms have been defined, including HTB, HFSC and HPFQ. These hierarchical scheme provide guarantees of requested bandwidth for each traffic class and also ensure that the excess bandwidth is properly shared among the currently active classes. In this project, HTB queuing technique present within the Linux kernel is used to provide differentiated bandwidth classes for normal data traffic and special video traffic that demand high bandwidth.

CHAPTER – 2

2.1 Mininet

Mininet is a network emulator tool used to create realistic virtual network. It uses process based virtualization and Linux namespaces to run different hosts and switches on a single OS kernel. This allows it to provide different network interfaces, routing tables and ARP tables to each of the hosts and switch instances. It connects all the network devices using virtual Ethernet (veth) pairs. Also, it allows the creation of custom network topologies for cloning and testing virtual network environments that makes it a suitable testbed for this project.

Within this project, Mininet was run on a virtual machine within Linux OS. Further, OpenFlow and Open vSwitch were installed within mininet to emulate the SDN environment required for this project.

2.2 OpenFlow

OpenFlow defines the protocol for exchange of information between the Open vSwitch and the SDN controller. It is an open source standard for message exchange in the SDN southbound APIs. Within the data plane, every datapath must be OpenFlow enabled to be able to send and interact with a control plane device using this protocol.

The OpenFlow enabled switch uses a simple message format to handle packets in this protocol. Every incoming packet is matched against the flow rules. If not found, the packet header information along with the datapath statistical information is passed to the controller. The controller then responds with the packet header, actions to be taken and the corresponding flow statistical information.

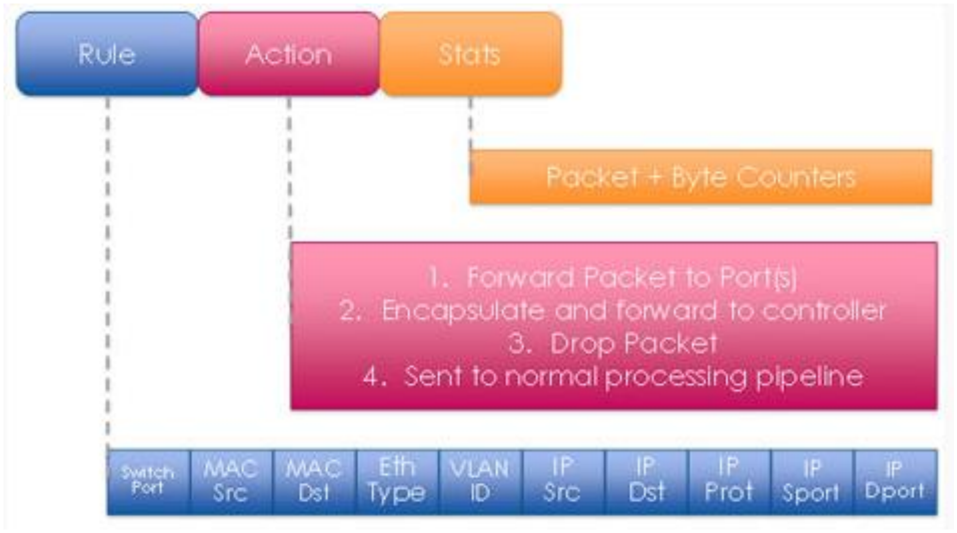


Figure 3. Flow Table entry in OpenFlow enabled Switches.

2.3 FlowVisor

FlowVisor is a special purpose SDN controller that acts as a transparent proxy between OpenFlow enabled switches and multiple SDN controllers. FlowVisor creates rich 'slices' of physical network resources based on any combination of switch ports (layer 1), src/dst ethernet address or type (layer 2), src/dst IP address or type (layer 3), and src/dst TCP/UDP port or ICMP code/type (layer 4). Further, it delegates control of each slice to a different controller. FlowVisor enforces isolation between each slice, i.e., traffic between the slices are kept separated and do not get mixed.

All connections to underlying virtual switches and SDN controllers are handled through a single poll loop. It will constantly poll to check if any of these devices have any messages to be exchanged between them. FVClassifier instance is created per each switch and switches are handed off to them once they make a connection with the FlowVisor. For every switch, internally, FlowVisor creates an FVSlicer instance per slice. This accounts for Switch x Slice number of FVSlicer instances.

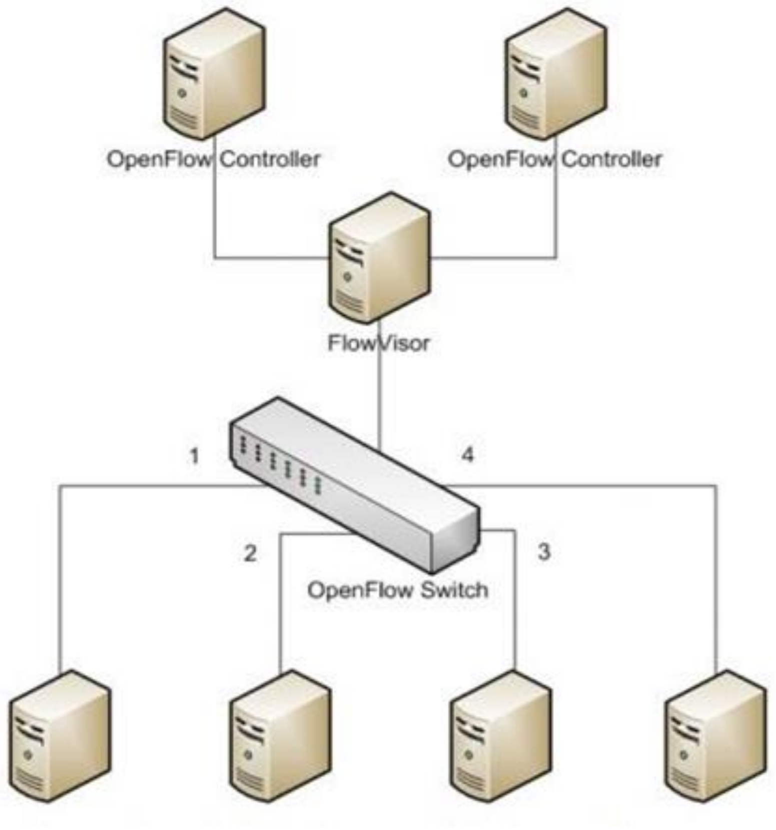


Figure 4. Setup of FlowVisor in SDN network.

2.4 SDN Controller

An SDN controller is the central decision making device within the network. It uses the OpenFlow protocol to exchange network state and statistical information that is crucial for its decision making ability. The controller provides network services that aid in network management. In addition to this, many different services within large complex networks can be multiplexed to achieve to dynamically alter the behavior of the network. Few popular controller are listed below-

OpenDayLight: It is an open source controller that is widely employed in production networks today. It is Java based and provides many network features such as Multi-tenancy that are currently not available in other controllers.

NOX: It was the first OpenFlow controller that was developed by Nicira networks. It is completely based on C++ and its development is now undertaken by the research community handling NOX.

POX: It is a Python based SDN controller that was created after NOX. It high level APIs allow for advanced implementations of network control software and enhanced prototyping. Within this project, POX controller is run on various SDN network slices as the default controller.

2.5 sFlow-RT

It is an SDN controller that provides real-time monitoring of the network statistics. Through its analytical approach, it receives a continuous stream of statistics from the network and converts them into measurable network metrics that is further accessible using the REST flow APIs. The network applications can be developed on any language that provides support for REST/HTTP. These applications alert the sFlow-RT controller about any change in network traffic through open protocols like Hybrid OpenFlow, JSON RPC, etc.

In this project, the sFlow-RT controller is used to monitor the performance of different network nodes as well as the overall traffic pattern through the collection of network parameters at every virtual node. These recording as then analyzed and plotted to demonstrate the improvement in the quality of service.

Chapter – 3

3.1 Network Topology

To generate the network topology, below steps must be followed-

- Create a Mininet VM session and install OpenFlow, Open vSwitch, FlowVisor, POX controller and sFlow-RT controller within this Mininet.
- Run the python script that contains the network topology using the Mininet libraries.

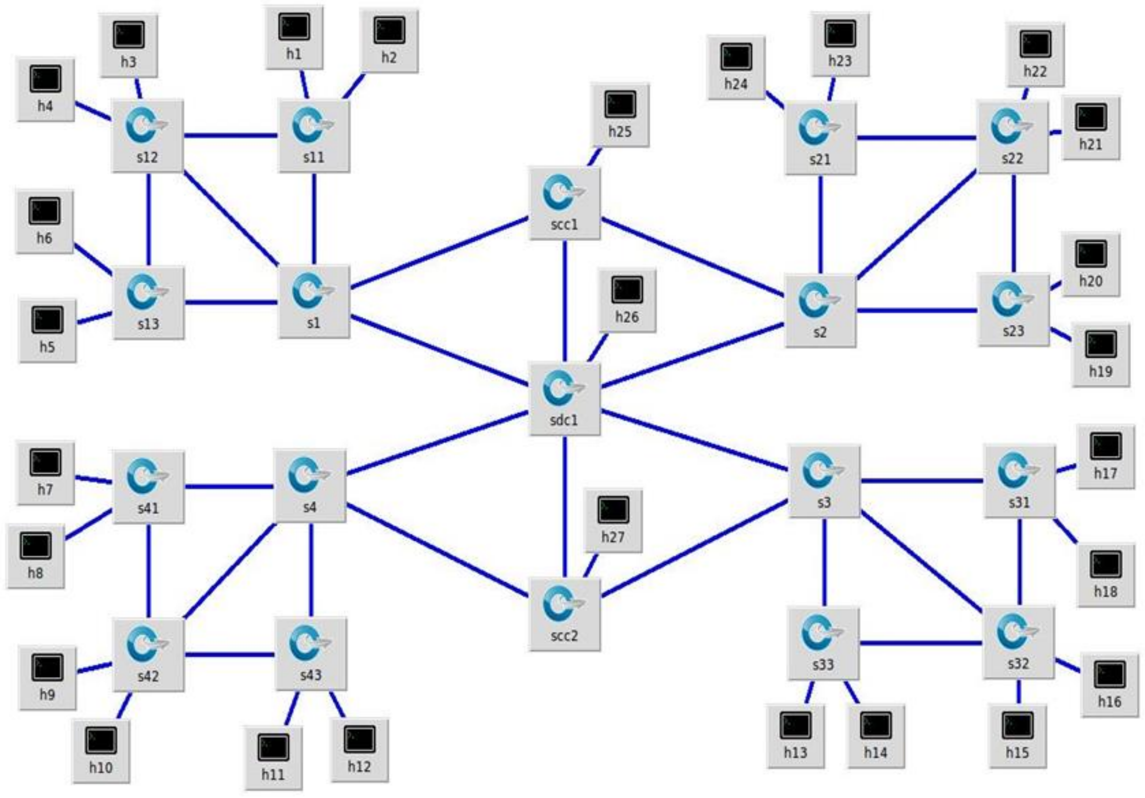
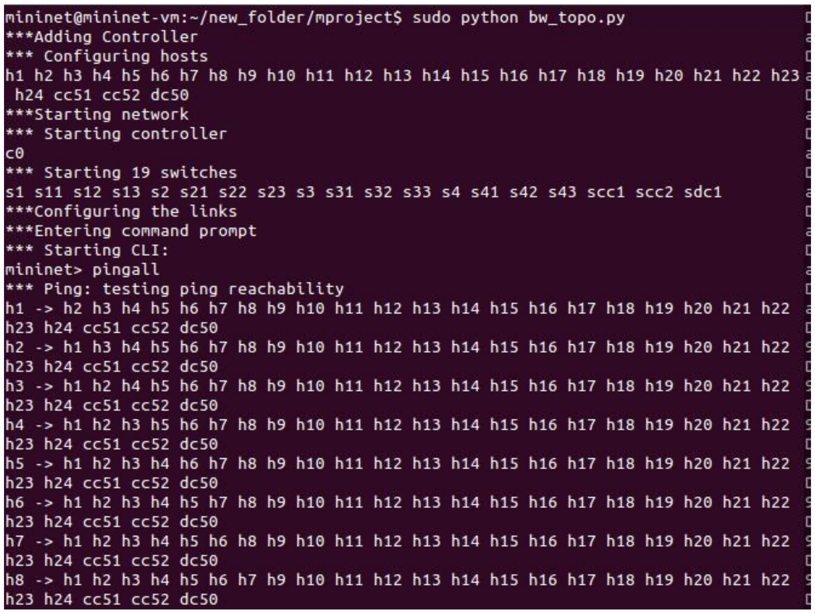


Figure 5. Service Provider Network Topology

- Open vSwitch is used to configure all virtual ports on every virtual switch with multiple bandwidth ranges.
- Once the network is created, Mininet enters into the CLI interactive session where the network parameters can be checked. To ensure that all the nodes are reachable and

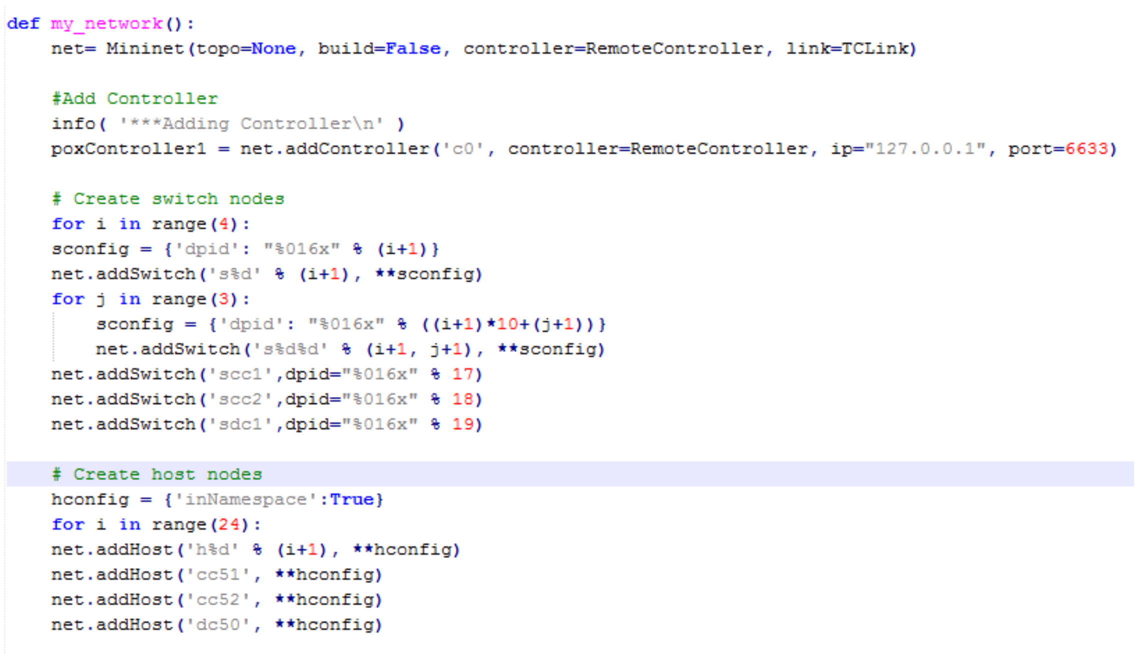
functioning properly, ‘*pingall’* command was successfully issues and tested. The result can be seen below.



```
mininet@mininet-vm:~/new_folder/mproject$ sudo python bw_topo.py
***Adding Controller
*** Configuring hosts
h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h21 h22 h23
 h24 cc51 cc52 dc50
***Starting network
*** Starting controller
c0
*** Starting 19 switches
s1 s11 s12 s13 s2 s21 s22 s23 s3 s31 s32 s33 s4 s41 s42 s43 scc1 scc2 sdc1
***Configuring the links
***Entering command prompt
*** Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h21 h22
h23 h24 cc51 cc52 dc50
h2 -> h1 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h21 h22
h23 h24 cc51 cc52 dc50
h3 -> h1 h2 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h21 h22
h23 h24 cc51 cc52 dc50
h4 -> h1 h2 h3 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h21 h22
h23 h24 cc51 cc52 dc50
h5 -> h1 h2 h3 h4 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h21 h22
h23 h24 cc51 cc52 dc50
h6 -> h1 h2 h3 h4 h5 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h21 h22
h23 h24 cc51 cc52 dc50
h7 -> h1 h2 h3 h4 h5 h6 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h21 h22
h23 h24 cc51 cc52 dc50
h8 -> h1 h2 h3 h4 h5 h6 h7 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h21 h22
h23 h24 cc51 cc52 dc50
```

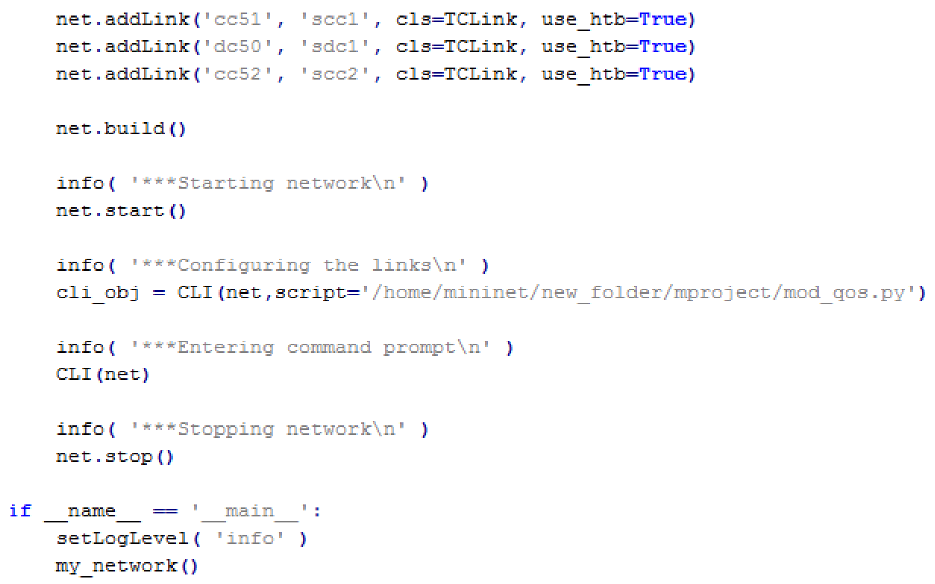
Figure 6. Mininet Network.

- The python script generating this code is provided below-



```
def my_network():
    net= Mininet(topo=None, build=False, controller=RemoteController, link=TCLink)

    #Add Controller
    info( '***Adding Controller\n' )
    poxController1 = net.addController('c0', controller=RemoteController, ip="127.0.0.1", port=6633)

    # Create switch nodes
    for i in range(4):
    sconfig = {'dpid': "%016x" % (i+1)}
    net.addSwitch('s%d' % (i+1), **sconfig)
    for j in range(3):
        sconfig = {'dpid': "%016x" % ((i+1)*10+(j+1))}
        net.addSwitch('s%d%d' % (i+1, j+1), **sconfig)
    net.addSwitch('scc1',dpid="%016x" % 17)
    net.addSwitch('scc2',dpid="%016x" % 18)
    net.addSwitch('sdc1',dpid="%016x" % 19)

    # Create host nodes
    hconfig = {'inNamespace':True}
    for i in range(24):
    net.addHost('h%d' % (i+1), **hconfig)
    net.addHost('cc51', **hconfig)
    net.addHost('cc52', **hconfig)
    net.addHost('dc50', **hconfig)
```

Figure 7. Code to create virtual Switches and Hosts.



```
    net.addLink('cc51', 'scc1', cls=TCLink, use_htb=True)
    net.addLink('dc50', 'sdc1', cls=TCLink, use_htb=True)
    net.addLink('cc52', 'scc2', cls=TCLink, use_htb=True)

    net.build()

    info( '***Starting network\n' )
    net.start()

    info( '***Configuring the links\n' )
    cli_obj = CLI(net,script='/home/mininet/new_folder/mproject/mod_qos.py')

    info( '***Entering command prompt\n' )
    CLI(net)

    info( '***Stopping network\n' )
    net.stop()

if __name__ == '__main__':
    setLogLevel( 'info' )
    my_network()
```

Figure 8. Code to create virtual links and build Topology

- As seen from the above code, the custom topology connects to FlowVisor Controller that is listening at 127.0.0.1:6633 location within the Mininet VM.

3.2 Virtualized Service Provider Network

In this project, network virtualization is achieved through slicing the network using a specialized SDN controller called FlowVisor. It is an SDN controller that has the capabilities to slice a network. Each slice is associated with a set of policies, called flow space. The network is directly connected to the FlowVisor. When a new packet arrives at the network, instead of sending the OpenFlow message to a general purpose SDN controller like Pox or OpenDayLight, here it is sent to the FlowVisor.

The FlowVisor is now responsible for handling these message packets. It checks its flow-space and based on the policies that match this particular packet the OpenFlow information is forwarding to the corresponding slice. Each slice now has a separate general purpose SDN controller waiting to handle this message.

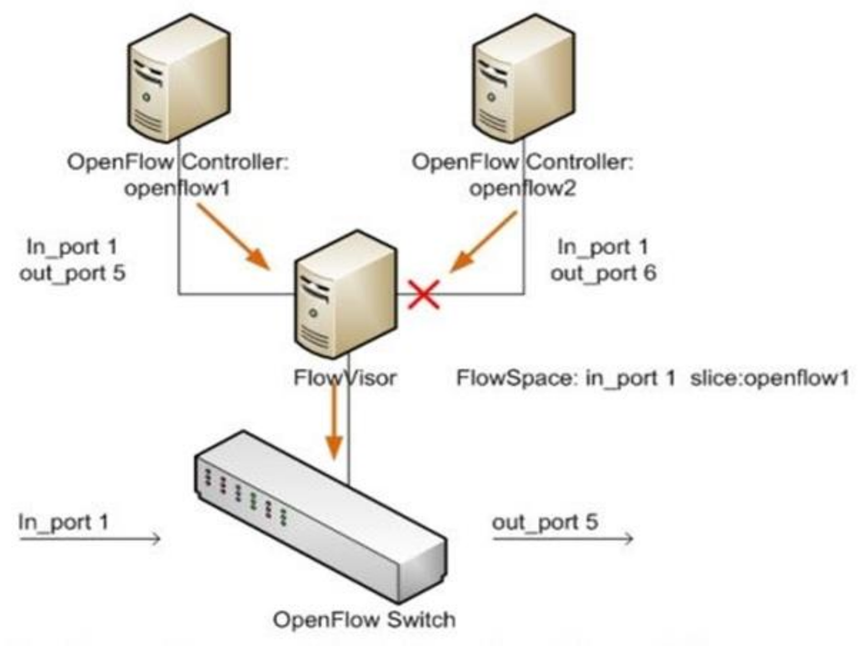

OpenFlow Controller: openflow1
OpenFlow Controller: openflow2
In_port 1
out_port 5
In_port 1
out_port 6
FlowVisor
FlowSpace: in_port 1 slice:openflow1
In_port 1
out_port 5
OpenFlow Switch

Figure 9. Policy based Slicing in FlowVisor.

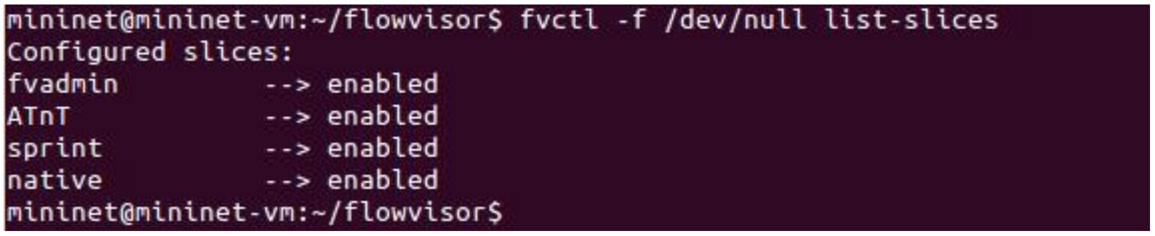
The network generated by the Mininet and Open vSwitch connects to the FlowVisor. At the same time, SDN controllers for each slice also communicate with the same FlowVisor.

FlowVisor Slicing

- The FlowVisor generates different slices using the command-

  `mininet@mininet-vm:~/ fvctl –f /dev/null add-slice ATnT tcp:localhost:10001 admin@ATnT`

- For this project three slices were created and successfully tested.

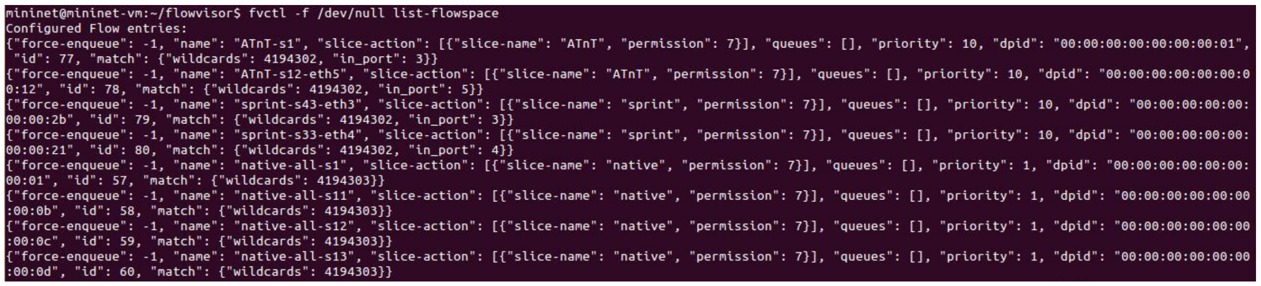


```
mininet@mininet-vm:~/flowvisor$ fvctl -f /dev/null list-slices
Configured slices:
fvadmin         --> enabled
ATnT            --> enabled
sprint          --> enabled
native          --> enabled
mininet@mininet-vm:~/flowvisor$
```

Figure 10. Slices defined in FlowVisor

- By default, all the switches and hosts are placed in slice ‘native’. This is considered as the default slice to be provided with minimum guaranteed quality of service for on-demand customers.
- The topology is then partitioned between two slices, ‘ATnT’ and ‘sprint’. Each slice is a virtual network that can support its own SDN controller. To demonstrate this two POX controller are connected to these two slices at ports 10001 and 20001 ports of localhost.
- Each slice is defined by a set of flow-space policies that defines the actions FlowVisor takes on receiving packets from each network slice.

```
mininet@mininet:~/ fvctl –f /dev/null add-flowspace ATnT-s1
00:00:00:00:00:00:00:01 10 in_port=3 ATnT=7
```



```
mininet@mininet-vm:~/flowvisor$ fvctl -f /dev/null list-flowspace
Configured Flow entries:
{"force-enqueue": -1, "name": "ATnT-s1", "slice-action": [{"slice-name": "ATnT", "permission": 7}], "queues": [], "priority": 10, "dpid": "00:00:00:00:00:00:00:01",
 "id": 77, "match": {"wildcards": 4194302, "in_port": 3}}
{"force-enqueue": -1, "name": "ATnT-s12-eth5", "slice-action": [{"slice-name": "ATnT", "permission": 7}], "queues": [], "priority": 10, "dpid": "00:00:00:00:00:00:0
0:12", "id": 78, "match": {"wildcards": 4194302, "in_port": 5}}
{"force-enqueue": -1, "name": "sprint-s43-eth3", "slice-action": [{"slice-name": "sprint", "permission": 7}], "queues": [], "priority": 10, "dpid": "00:00:00:00:00:
00:00:2b", "id": 79, "match": {"wildcards": 4194302, "in_port": 3}}
{"force-enqueue": -1, "name": "sprint-s33-eth4", "slice-action": [{"slice-name": "sprint", "permission": 7}], "queues": [], "priority": 10, "dpid": "00:00:00:00:00:
00:00:21", "id": 80, "match": {"wildcards": 4194302, "in_port": 4}}
{"force-enqueue": -1, "name": "native-all-s1", "slice-action": [{"slice-name": "native", "permission": 7}], "queues": [], "priority": 1, "dpid": "00:00:00:00:00:00:
00:01", "id": 57, "match": {"wildcards": 4194303}}
{"force-enqueue": -1, "name": "native-all-s11", "slice-action": [{"slice-name": "native", "permission": 7}], "queues": [], "priority": 1, "dpid": "00:00:00:00:00:00
:00:0b", "id": 58, "match": {"wildcards": 4194303}}
{"force-enqueue": -1, "name": "native-all-s12", "slice-action": [{"slice-name": "native", "permission": 7}], "queues": [], "priority": 1, "dpid": "00:00:00:00:00:00
:00:0c", "id": 59, "match": {"wildcards": 4194303}}
{"force-enqueue": -1, "name": "native-all-s13", "slice-action": [{"slice-name": "native", "permission": 7}], "queues": [], "priority": 1, "dpid": "00:00:00:00:00:00
:00:0d", "id": 60, "match": {"wildcards": 4194303}}
```

Figure 11. Flow-spaces in FlowVisor.

Flow-Space Policies

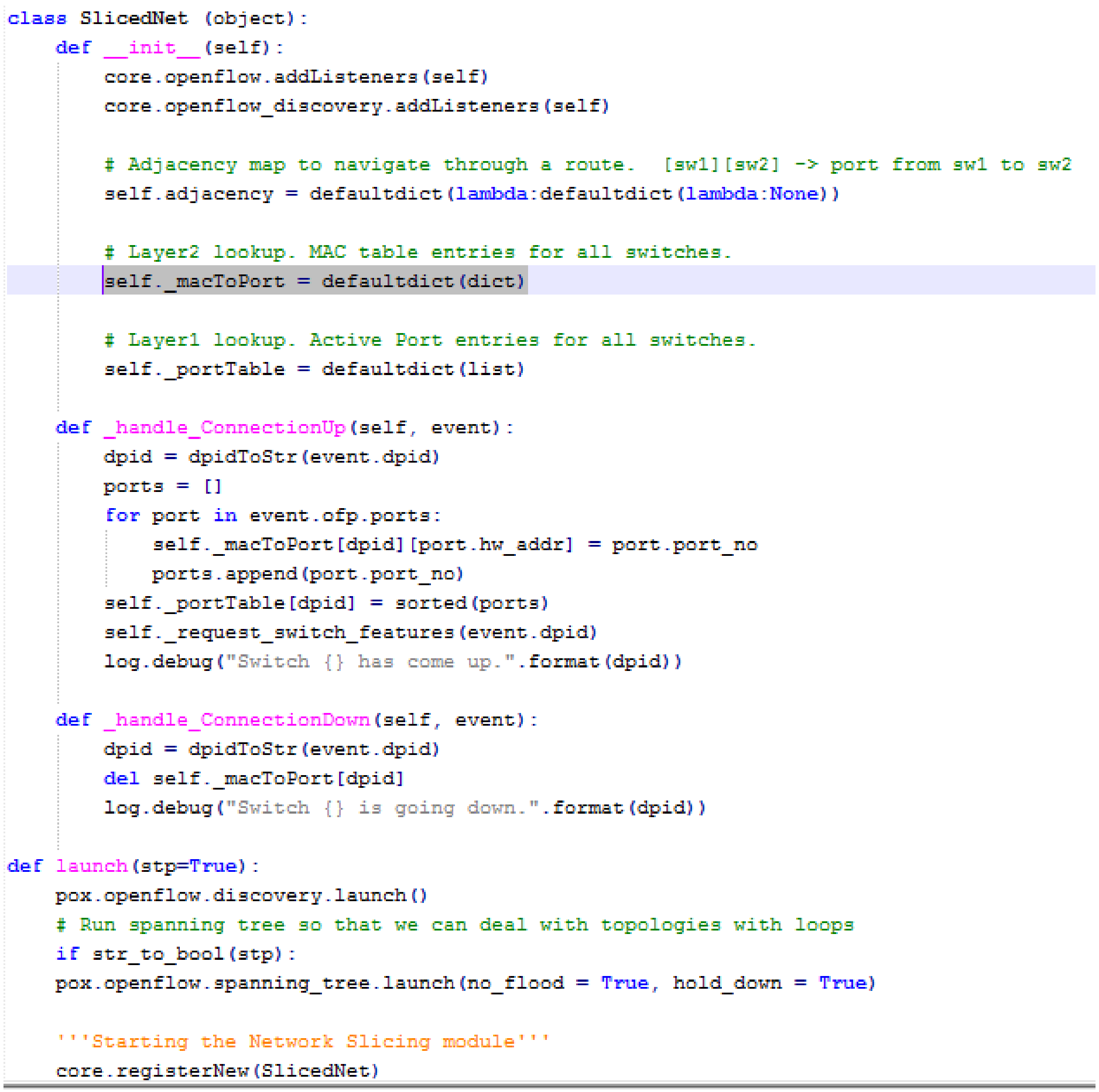
- The flow space is a set of policies associated with a network slice that governs the decision making process of the FlowVisor.
- Every policy is directed to at least one slice thus ensuring that the packets that match this policy is directed to corresponding SDN controller.
- Further, pre-defined permissions can be set on every policy for each slice. Slices can either read, write or do both to a flow-space policy. They can also delegate their actions to another slice for a particular policy.

- Each policy is defined on a single datapath (network devices) thus providing fine granularity and multiple scenarios to decision making of FlowVisor.
- It is possible to have multiple policies in different flow spaces that match a single packet. To resolve this issue, every policy is provided with a priority value. Policies with larger priority values are chosen when compared to lower value ones.

With this fine grain policy making facility, FlowVisor can efficiently slice the network and its isolation mechanism adds security to it. These are the reasons why FlowVisor was chosen as a Network Virtualization tool for this project.

Connectivity within slices

- To ensure that the SDN controllers running on top of the slices are able to properly view their own slice topology, this project provides a simple application to be run on these controller.
- Upon starting this application, it registers all the data paths that are part of its own slice along with its ports and their corresponding MAC address.
- It then generates a MAC Table that provides mapping between the datapath ports that are part of the slice and their corresponding MAC entries. This is crucial because the POX SDN controller running on each slice must be able to recognize that virtual network that it is catering to.
- FlowVisor is only responsible to segregate the OpenFlow messages that are being communicated between the SDN controller and the data paths. It does not provide any topological information to the above SDN controllers.

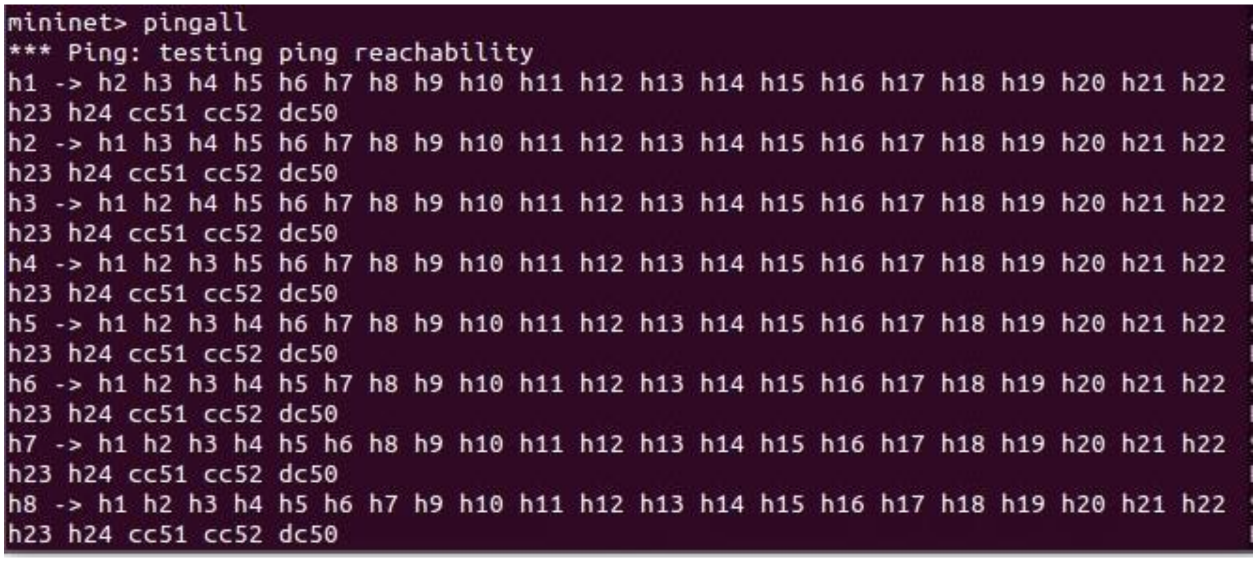


```
class SlicedNet (object):
    def __init__(self):
        core.openflow.addListeners(self)
        core.openflow_discovery.addListeners(self)

        # Adjacency map to navigate through a route.  [sw1][sw2] -> port from sw1 to sw2
        self.adjacency = defaultdict(lambda:defaultdict(lambda:None))

        # Layer2 lookup. MAC table entries for all switches.
        self._macToPort = defaultdict(dict)

        # Layer1 lookup. Active Port entries for all switches.
        self._portTable = defaultdict(list)

    def _handle_ConnectionUp(self, event):
        dpid = dpidToStr(event.dpid)
        ports = []
        for port in event.ofp.ports:
            self._macToPort[dpid][port.hw_addr] = port.port_no
            ports.append(port.port_no)
        self._portTable[dpid] = sorted(ports)
        self._request_switch_features(event.dpid)
        log.debug("Switch {} has come up.".format(dpid))

    def _handle_ConnectionDown(self, event):
        dpid = dpidToStr(event.dpid)
        del self._macToPort[dpid]
        log.debug("Switch {} is going down.".format(dpid))

def launch(stp=True):
    pox.openflow.discovery.launch()
    # Run spanning tree so that we can deal with topologies with loops
    if str_to_bool(stp):
    pox.openflow.spanning_tree.launch(no_flood = True, hold_down = True)

    '''Starting the Network Slicing module'''
    core.registerNew(SlicedNet)
```

Figure 12. SDN application to generate the topology within slices.

- The above code shows the new application, called SlicedNet that must be running on SDN controllers to effectively generate a topology of the network. It also highlights the slice specific MAC table that this application provides to other application running on this controller for ease of operation.

- Other applications can now access the slice topology information provided by the SliceNet application to forward or route packets within the slice.
- At the end of slicing, the connectivity is each slice is tested by successfully receiving the PING replies from all the hosts within that slice. Inter-slice connectivity is not possible due to FlowVisor isolation.



```
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h21 h22
h23 h24 cc51 cc52 dc50
h2 -> h1 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h21 h22
h23 h24 cc51 cc52 dc50
h3 -> h1 h2 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h21 h22
h23 h24 cc51 cc52 dc50
h4 -> h1 h2 h3 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h21 h22
h23 h24 cc51 cc52 dc50
h5 -> h1 h2 h3 h4 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h21 h22
h23 h24 cc51 cc52 dc50
h6 -> h1 h2 h3 h4 h5 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h21 h22
h23 h24 cc51 cc52 dc50
h7 -> h1 h2 h3 h4 h5 h6 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h21 h22
h23 h24 cc51 cc52 dc50
h8 -> h1 h2 h3 h4 h5 h6 h7 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h21 h22
h23 h24 cc51 cc52 dc50
```

Figure 13. Successful PING replies in each slice.

3.3 Network Applications

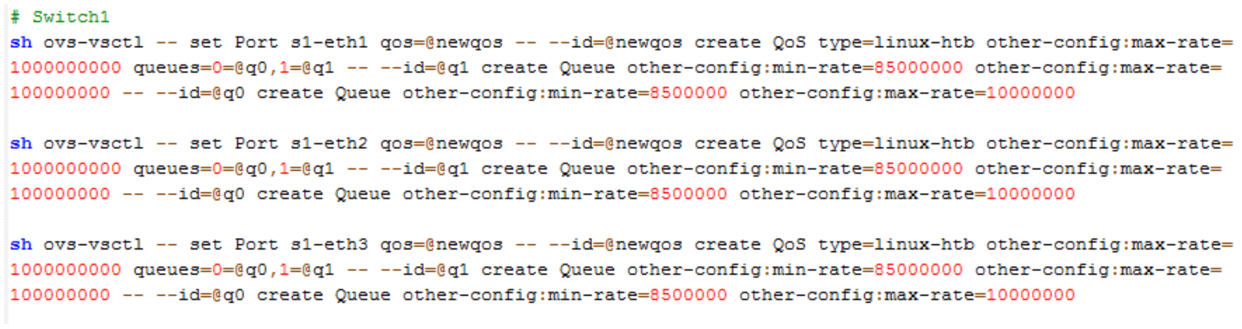
In this project, each slice created by the FlowVisor is a virtual network capable of handling different network applications. To demonstrate this, we run a Content Delivery Service on one slice and simple shortest path first (SPF) on the other slice.

Further, to improve the Quality of Service in each slice, underlying infrastructure network is provided with links that can handle multiple bandwidth requirements. These virtual networks can simultaneously operate on these links and hence can take the full advantage of bandwidth available. Based on the type of service (TOS) of the traffic, each slice can decide on the type of bandwidth service that needs to be allocated to them.

Quality of Service

In a topology with virtual networks, traffic from all these networks travel through the underlying physical links. To improve the Quality of Service that reflects in multiple networks, the bandwidth characteristics of the links are modified in this project. **Hierarchical Token Bucket (HTB)** is enabled on every link to the incoming and the outgoing traffic. HTB is defined with multiple queues handling traffic at a higher bandwidth range of 80-100 Mbps and lower bandwidth range of 8.5-10 Mbps.

Due to the network virtualization, both the slices defined in this project can simultaneously access these queues in the physical links and yet ensure perfect isolation between the traffic. In a virtual environment, this allows the service providers to customize their bandwidth requirement as per their traffic and not be worried about the traffic passing through the provider’s network. Also, it prevents unnecessary bandwidth based service level agreements between the providers that could lead to inter-operability issues.



```
# Switch1
sh ovs-vsctl -- set Port s1-eth1 qos=@newqos -- --id=@newqos create QoS type=linux-htb other-config:max-rate=
1000000000 queues=0=@q0,1=@q1 -- --id=@q1 create Queue other-config:min-rate=85000000 other-config:max-rate=
100000000 -- --id=@q0 create Queue other-config:min-rate=8500000 other-config:max-rate=10000000

sh ovs-vsctl -- set Port s1-eth2 qos=@newqos -- --id=@newqos create QoS type=linux-htb other-config:max-rate=
1000000000 queues=0=@q0,1=@q1 -- --id=@q1 create Queue other-config:min-rate=85000000 other-config:max-rate=
100000000 -- --id=@q0 create Queue other-config:min-rate=8500000 other-config:max-rate=10000000

sh ovs-vsctl -- set Port s1-eth3 qos=@newqos -- --id=@newqos create QoS type=linux-htb other-config:max-rate=
1000000000 queues=0=@q0,1=@q1 -- --id=@q1 create Queue other-config:min-rate=85000000 other-config:max-rate=
100000000 -- --id=@q0 create Queue other-config:min-rate=8500000 other-config:max-rate=10000000
```

Figure 14. HTB based Bandwidth scheduling on the virtual links.

Content Delivery Network

Content Delivery Network is a large distributed system of proxy servers that delivers the Web content to users based on their geographical location, origin of the web page or the location of content delivery server. It is effective in rapid delivery of the content between websites that have a global reach to the websites with high traffic. The closer the CDN server is to the users, faster will be the content delivery.

When a user requests a Web content, the server that is closest to the user responds to the request. In CDN, the contents of the page are copied by the server at geographical different locations, thus caching the contents of the page. Upon receiving the request from the user, the original request handling server will redirect the request to the CDN server that is closer in location to deliver the cached content of the web information. If that information is not previously cached, then the CDN server communicates with the original request handling server to resolve this issue.

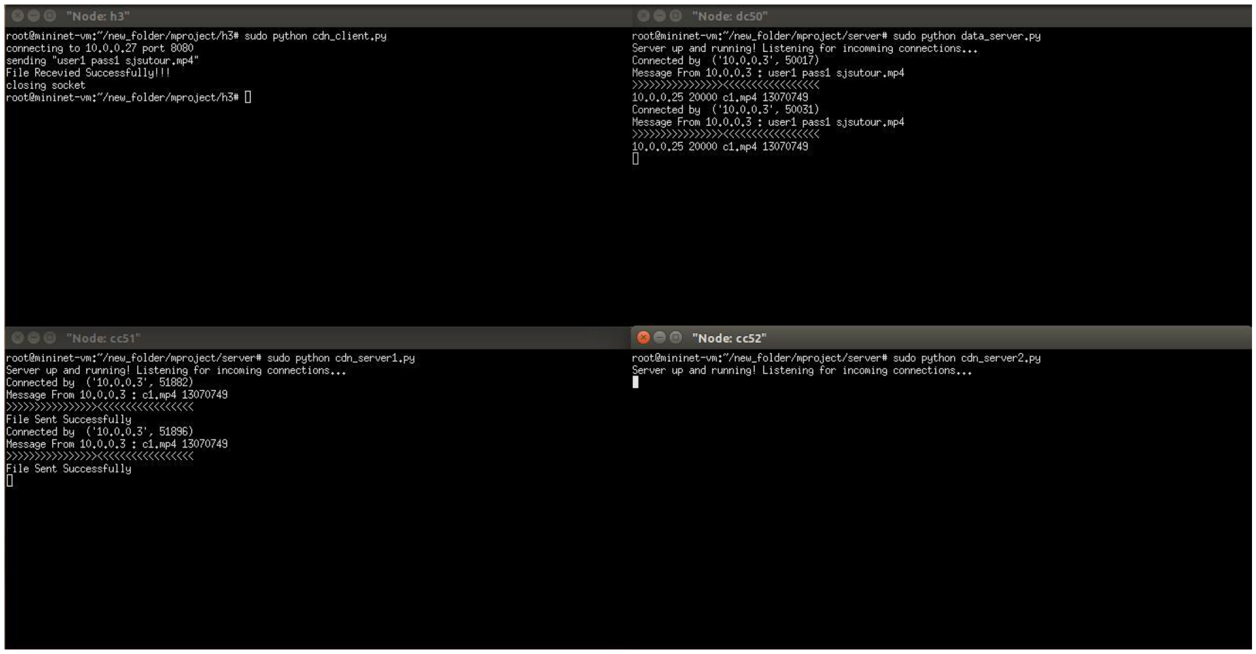
There are different types of Content Delivery Networks-

- **Commercial CDN.** These CDN function in a way similar to the original concept of CDN described above. They organize and provide access to large web content on a large complex networks of proxy servers handling many thousands of requests.
- **Peer-to-Peer CDN.** In this type of CDN service, the client consume as well as provide resources to other clients. Thus, unlike simple client-server approach, this CDN approach can perform much better with more client adding to it. This is one of the major advantages of Peer-to-Peer CDN service requiring very low initial setup expenditure of the original CDN provider.

- **Private CDN.** If the owners of the content are not satisfied with commercial CDN service, they can opt of private CDNs. It consists of multiple POPs that are always serving the content only to the owner. These POPs can be application delivery controllers or proxy servers. They can also range from simple two servers to a network of proxy servers carrying petabytes of content.

To demonstrate the functioning of the CDN within a slice of the network, multiple hosts were selected to act as CDN data server and CDN content servers. The CDN data server is responsible for responding to user requests. Upon receiving the user request, the CDN data server analyzes the request and forwards it to the CDN content servers that are at geographically closer locations to transfers the cached information. In this project, **dc50** is a CDN data server while **cc51** and **cc52** are CDN content delivery servers catering to two different geographical range.

To enhance the quality of service of the Content Delivery Network, the initial user requests between the users and the CDN data server are provided with a low guaranteed bandwidth of 10Mbps. While the subsequent web content transfer between the user and the CDN content delivery servers are provided with the high bandwidth of 100Mbps. These bandwidth caps can be changed dynamically thus greatly improving the speed of web content delivery within the network.



```
root@mininet-vm:~/new_folder/mproject/h3# sudo python cdn_client.py
connecting to 10.0.0.27 port 8080
sending "user1 pass1 sjsutour.mp4"
File Recevied Successfully!!!
closing socket
root@mininet-vm:~/new_folder/mproject/h3# 
```

```
root@mininet-vm:~/new_folder/mproject/server# sudo python data_server.py
Server up and running! Listening for incomming connections...
Connected by  ('10.0.0.3', 50017)
Message From 10.0.0.3 : user1 pass1 sjsutour.mp4
>>>>>>>>>>>>>>><<<<<<<<<<<<<<<<
10.0.0.25 20000 c1.mp4 13070749
Connected by  ('10.0.0.3', 50031)
Message From 10.0.0.3 : user1 pass1 sjsutour.mp4
>>>>>>>>>>>>>>><<<<<<<<<<<<<<<<
10.0.0.25 20000 c1.mp4 13070749
```

```
root@mininet-vm:~/new_folder/mproject/server# sudo python cdn_server1.py
Server up and running! Listening for incoming connections...
Connected by  ('10.0.0.3', 51882)
Message From 10.0.0.3 : c1.mp4 13070749
>>>>>>>>>>>>>>><<<<<<<<<<<<<<<<
File Sent Successfully
Connected by  ('10.0.0.3', 51896)
Message From 10.0.0.3 : c1.mp4 13070749
>>>>>>>>>>>>>>><<<<<<<<<<<<<<<<
File Sent Successfully
```

```
root@mininet-vm:~/new_folder/mproject/server# sudo python cdn_server2.py
Server up and running! Listening for incoming connections...
```

Figure 15. Content Delivery Network.

Here a user (Host h3) requests for a video file from the CDN data server (Host dc50). Upon receiving the request, the data server transfer the information request to the CDN content server (Host cc51) to transfer the cached video file. Both CDN content servers, cc51 and cc52, are UP and listening for connections. Host cc51 then successfully transfers the video file to the requesting user, h3 as seen from the figure above. The received video file is then played using the VLC player within the user system as seen below.

Chapter – 4

4.1 Results

The results of this project implementation is evaluated in two level are detailed below-

4.1.1 Network Slicing and Performance

In this project, two slices were created over a single emulated network topology of an infrastructure provider. FlowVisor was employed to achieve policy-based network slicing.

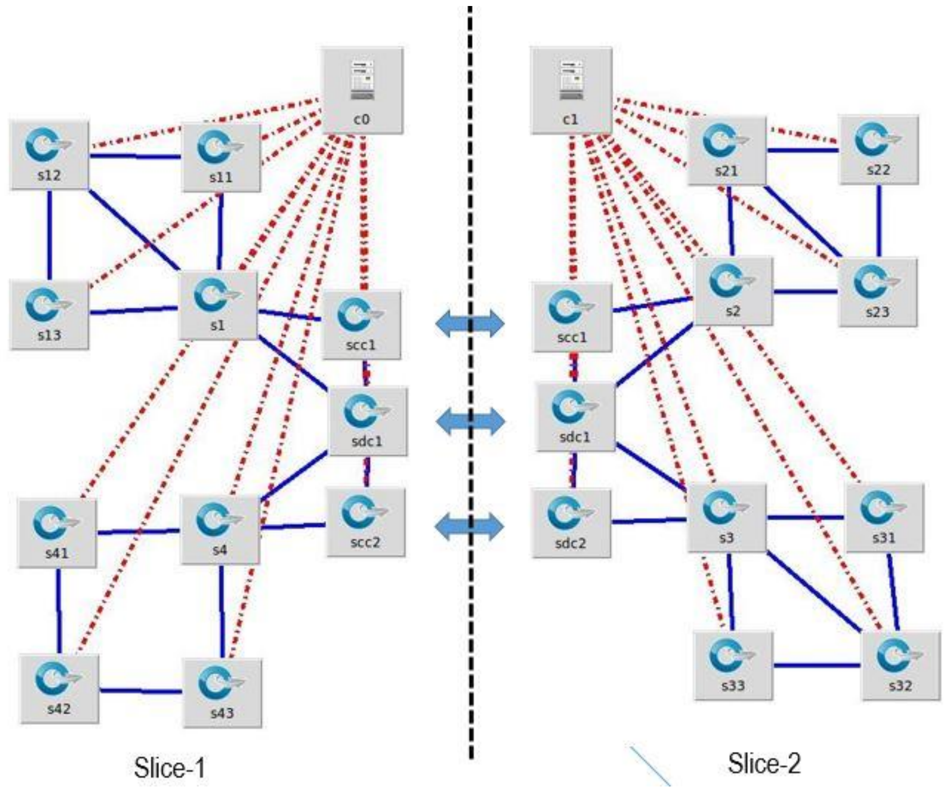
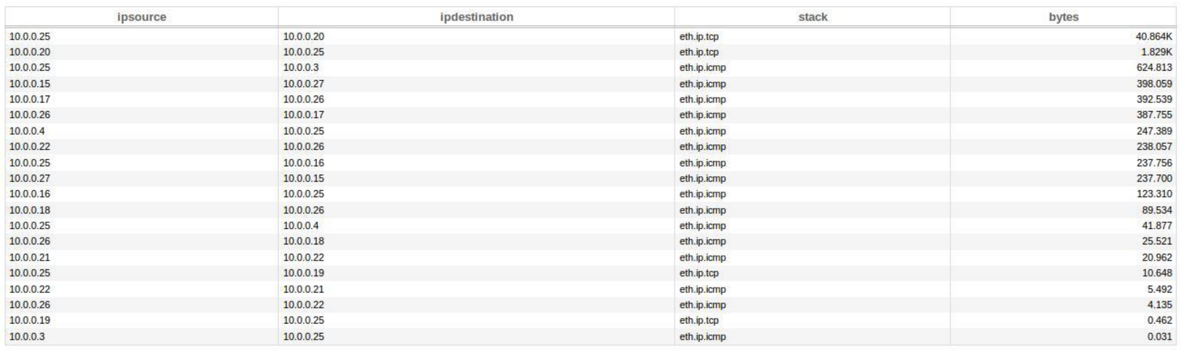


Figure 16. Network slicing from FlowVisor.

As seen from the above figure both controllers have their own slice of topology in which they have the freedom to run customized applications. Here the central switches scc1, sdc1 and scc2 are part of both the slices for inter-communicability. Successful PING responses were ensured to check the connectivity within each slice.

- To check the performance of virtual networks in each slice two separate network application with different use cases were chosen.
- On one slice, a Content Delivery Service was implemented. On the other slice, a simple shortest path first algorithm was run.
- To check the capabilities of the slices during high traffic conditions, load on each slice was increased. sFlow-RT was initiated to collect all the network parameters and these metrics were plotted.
- By analyzing the amount of bandwidth utilized by different slices, it becomes easier to predict and regulate behavior of the network in SDN and thereby optimizing the network performance.



ipsource	ipdestination	stack	bytes
10.0.0.25	10.0.0.20	eth.ip.tcp	40.864K
10.0.0.20	10.0.0.25	eth.ip.tcp	1.829K
10.0.0.25	10.0.0.3	eth.ip.icmp	624.813
10.0.0.15	10.0.0.27	eth.ip.icmp	398.059
10.0.0.17	10.0.0.26	eth.ip.icmp	392.539
10.0.0.26	10.0.0.17	eth.ip.icmp	387.755
10.0.0.4	10.0.0.25	eth.ip.icmp	247.389
10.0.0.22	10.0.0.26	eth.ip.icmp	238.057
10.0.0.25	10.0.0.16	eth.ip.icmp	237.756
10.0.0.27	10.0.0.15	eth.ip.icmp	237.700
10.0.0.16	10.0.0.25	eth.ip.icmp	123.310
10.0.0.18	10.0.0.26	eth.ip.icmp	89.534
10.0.0.25	10.0.0.4	eth.ip.icmp	41.877
10.0.0.26	10.0.0.18	eth.ip.icmp	25.521
10.0.0.21	10.0.0.22	eth.ip.icmp	20.962
10.0.0.25	10.0.0.19	eth.ip.tcp	10.648
10.0.0.22	10.0.0.21	eth.ip.icmp	5.492
10.0.0.26	10.0.0.22	eth.ip.icmp	4.135
10.0.0.19	10.0.0.25	eth.ip.tcp	0.462
10.0.0.3	10.0.0.25	eth.ip.icmp	0.031

Figure 17. Packets from different slices captured in sFlow-RT

- HTB based queuing was initiated to improve QoS on the CDN service running on one slice. Low bandwidth was allocated during initial requests while high bandwidth was allocated during the final content transfer.

  At the same time, in the second slice, to demonstrate on demand high bandwidth facility, few hosts were provisioned with high bandwidth while others were continues in the default low bandwidth class of service. Iperf traffic was generated between these two types of hosts, alternatively, to capture the network parameters.

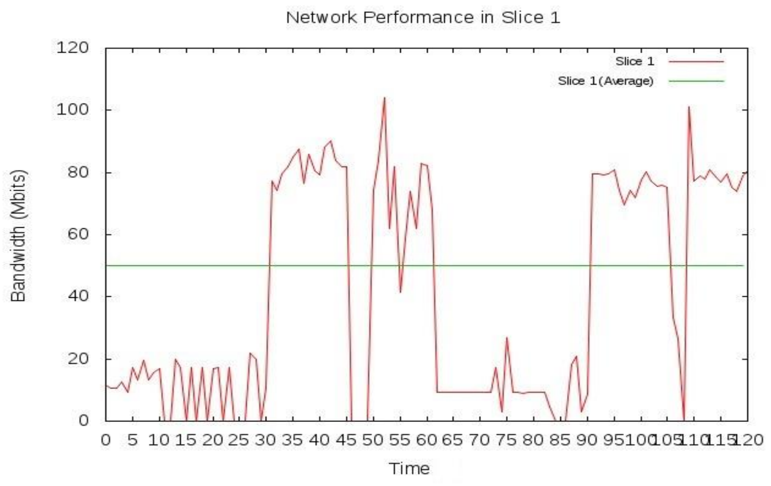


Figure 18. CDN traffic in network slice 1.

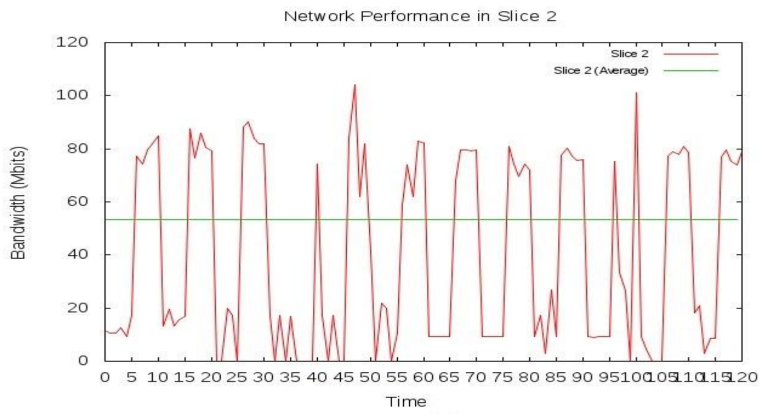


Figure 19. Iperf capture for On-Demand traffic in network slice 2.

- The packet count and the utilization at each port was also recorded and analyzed using sFlow-RT.

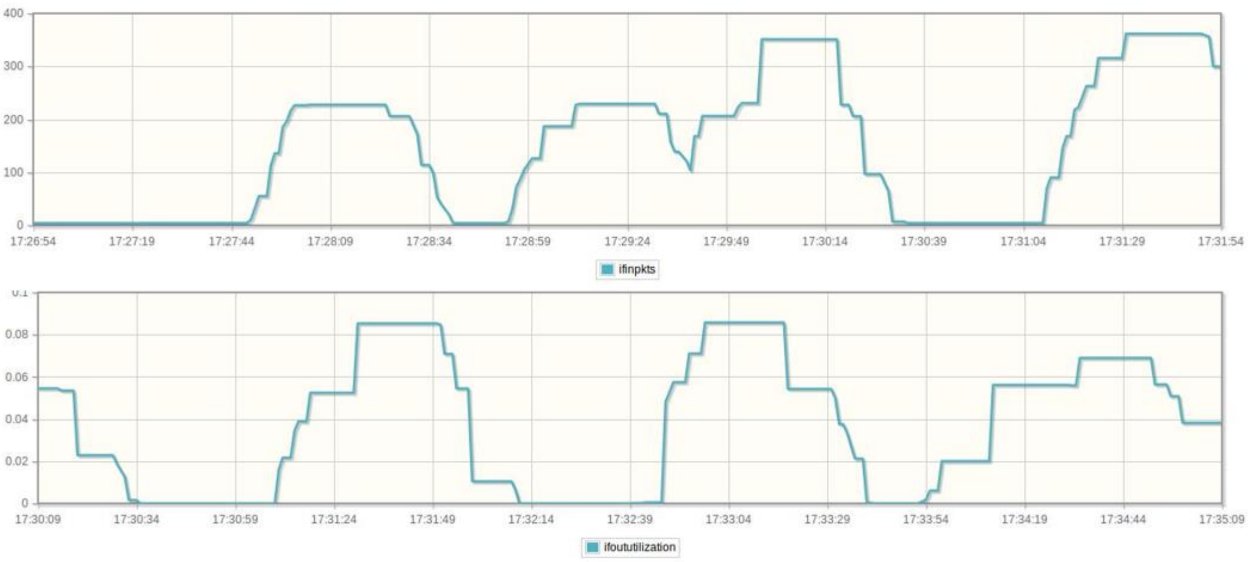
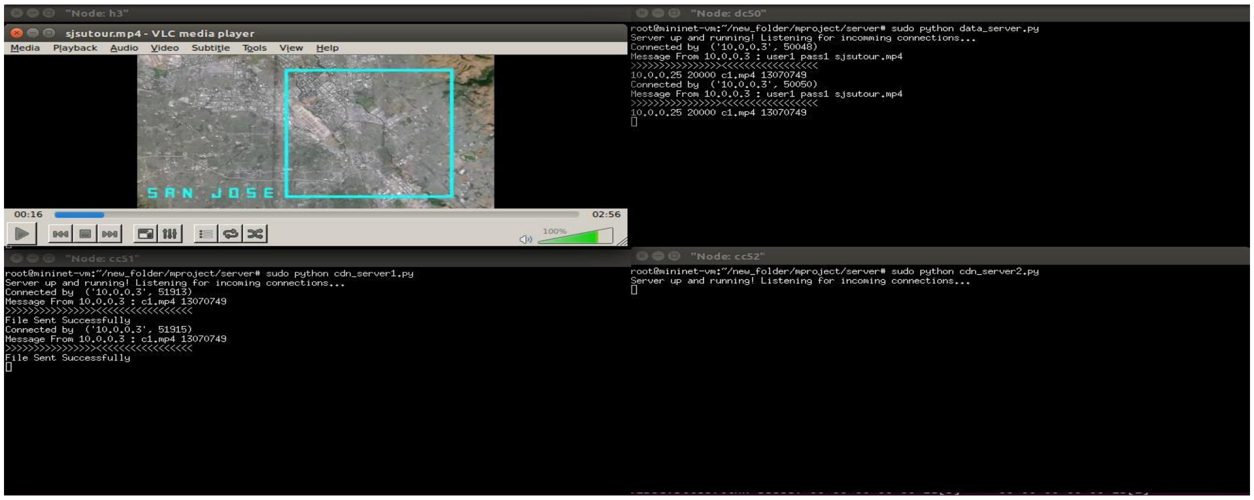


Figure 20. Port Statistics at data-path s11 in slice 1.

4.1.2 <u>CDN</u>

- In this implementation, single CDN data server and two CDN content servers were created.
- Other hosts in this slice query for a video file from the CDN data server using minimum guaranteed low bandwidth.
- This server then suitably redirects the request to CDN content server based on geographical location of the host.
- The content server then transfers the video file through high bandwidth of 80-100 Mbps.


"Node: h3"
sjsutour.mp4 - VLC media player
Media Playback Audio Video Subtitle Tools View Help
SAN JOSE
00:16
02:56
100%
"Node: dc50"
"Node: cc51"
"Node: cc52"

```
root@mininet-vm:~/new_folder/mproject/server# sudo python data_server.py
Server up and running! Listening for incomming connections...
Connected by  ('10.0.0.3', 50048)
Message From 10.0.0.3 : user1 pass1 sjsutour.mp4
>>>>>>>>>>>>>>>><<<<<<<<<<<<<<<<
10.0.0.25 20000 c1.mp4 13070749
Connected by  ('10.0.0.3', 50050)
Message From 10.0.0.3 : user1 pass1 sjsutour.mp4
>>>>>>>>>>>>>>>><<<<<<<<<<<<<<<<
10.0.0.25 20000 c1.mp4 13070749
```

```
root@mininet-vm:~/new_folder/mproject/server# sudo python cdn_server1.py
Server up and running! Listening for incoming connections...
Connected by  ('10.0.0.3', 51913)
Message From 10.0.0.3 : c1.mp4 13070749
>>>>>>>>>>>>>>>><<<<<<<<<<<<<<<<
File Sent Successfully
Connected by  ('10.0.0.3', 51915)
Message From 10.0.0.3 : c1.mp4 13070749
>>>>>>>>>>>>>>>><<<<<<<<<<<<<<<<
File Sent Successfully
```

```
root@mininet-vm:~/new_folder/mproject/server# sudo python cdn_server2.py
Server up and running! Listening for incoming connections...
```

Figure 21. Transferred Video content being played in Host h3.

Conclusion

The primary goal of service providers is to efficiently provide specialized services to their customers at competitive rates. While enterprises and end customers are constantly demanding for different services simultaneously, the current service provider infrastructure does not allow for complete optimizations leading to inefficiency. As an alternative, using SDN along with FlowVisor can provide a flexible approach in rapidly deploying required services and thereby making the network easily manageable. This flexibility in allocating network resources, improves overall network efficiency and allows the service providers to meet the demands of their customers instantaneously. Finally, the designed system gives multiple carriers some control over how their traffic is handled inside the core network and the freedom to run custom (or) experimental applications.

References

[1] N. Feamster, J. Rexford, and E. Zegura, “The Road to SDN: An Intellectual History of Programmable Networks,” SIGCOMM Comput Commun Rev, vol. 44, no. 2, pp. 87–98, Apr. 2014.

[2] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, “OpenFlow: Enabling Innovation in Campus Networks,” SIGCOMM Comput Commun Rev, vol. 38, no. 2, pp. 69–74, Mar. 2008.

[3] Chowdhury, N. M., and Raouf Boutaba. "Network virtualization: state of the art and research challenges." Communications Magazine, IEEE 47.7 (2009): 20-26.

[4] Kreutz, D.; Ramos, F.M.V.; Esteves Verissimo, P.; Esteve Rothenberg, C.; Azodolmolky, S.; Uhlig, S., "Software-Defined Networking: A Comprehensive Survey," Proceedings of the IEEE , vol.103, no.1, pp.14,76, Jan. 2015.

[5] Sherwood, Rob, et al. "Flowvisor: A network virtualization layer." OpenFlow Switch Consortium, Tech. Rep (2009).

[6] M.R. Nascimento et al., “Virtual Routers as a Service: The RouteFlow Approach Leveraging Software-Defined Networks,” Proc. Conf. Future Internet Technologies (CFI), ACM, 2011, pp. 34–37.

[7] N. M. M. K. Chowdhury and R. Boutaba, “A survey of network virtualization,” Computer Networks, vol. 54, no. 5, pp. 862–876, Apr. 2010.

[8] M. F. Bari, R. Boutaba, R. Esteves, L. Z. Granville, M. Podlesny, M. G. Rabbani, Q. Zhang, and M. F. Zhani, “Data center network virtualization: A survey,” Communications Surveys & Tutorials, IEEE, vol. 15, no. 2, pp. 909–928, 2013.N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow:

enabling innovation in campus networks. ACM SIGCOMM Computer Communication Review, 38(2):69–74, April 2008.

[9] N. Gude et al., “NOX: Towards an Operating System for Networks,” ACM SIGCOMM Computer Communication Rev., vol. 38, no. 3, 2008, pp. 105–110.

[10] N. Chowdhury, M. Rahman, and R. Boutaba, “Virtual Network Embedding with Coordinated Node and Link Mapping,” Proc. 28th IEEE Conf. Computer Communications (INFOCOM 09), IEEE, 2009, pp. 783–791.

[11] Kreutz, Diego, et al. "Software-defined networking: A comprehensive survey. *“Proceedings of the IEEE* 103.1 (2015): 14-76.

[12] H. Kim and N. Feamster, ‘‘Improving network management with software defined networking,’’ IEEE Commun. Mag., vol. 51, no. 2, pp. 114–119, Feb. 2013.

[13] M. Casado, N. Foster, and A. Guha, ‘‘Abstractions for software-defined networks,’’ ACM Commun., vol. 57, no. 10, pp. 86–95, Sep. 2014

[14] K. Calvert, S. Bhattacharjee, E. Zegura, and J. Sterbenz, ‘‘Directions in active networks,’’ IEEE Commun.Mag., vol. 36, no. 10, pp. 72–78, Oct. 1998.

[15] Y.-D. Lin, D. Pitt, D. Hausheer, E. Johnson, and Y.-B. Lin, “Software-Defined Networking: Standardization for Cloud Computing’s Second Wave,” Computer, no. 11, pp. 19–21, 2014

[16] Jarschel, M.; Zinner, T.; Hossfeld, T.; Tran-Gia, P.; Kellerer, W., "Interfaces, attributes, and use cases: A compass for SDN," *Communications Magazine, IEEE* , vol.52, no.6, pp.210,217, June 2014

[17] Bari, M.F.; Boutaba, R.; Esteves, R.; Granville, L.Z.; Podlesny, M.; Rabbani, M.G.; Qi Zhang; Zhani, M.F., "Data Center Network Virtualization: A Survey," *Communications Surveys & Tutorials, IEEE* , vol.15, no.2, pp.909,928, Second Quarter 2013

[18] Drutskoy, Dmitry; Keller, Eric; Rexford, Jennifer, "Scalable Network Virtualization in Software-Defined Networks," Internet Computing, IEEE , vol.17, no.2, pp.20,27, March-April 2013

[19] A. Al-Shabibi et al., ‘‘OpenVirteX: Make your virtual SDNs programmable,’’ in Proc. 3rd Workshop Hot Topics Softw. Defined Netw., 2014, pp. 25–30.

[20] D. A. Drutskoy, ‘‘Software-defined network virtualization with FlowN,’’ Ph.D. dissertation, Dept. Comput. Sci., Princeton Univ., Princeton, NJ, USA, Jun. 2012.

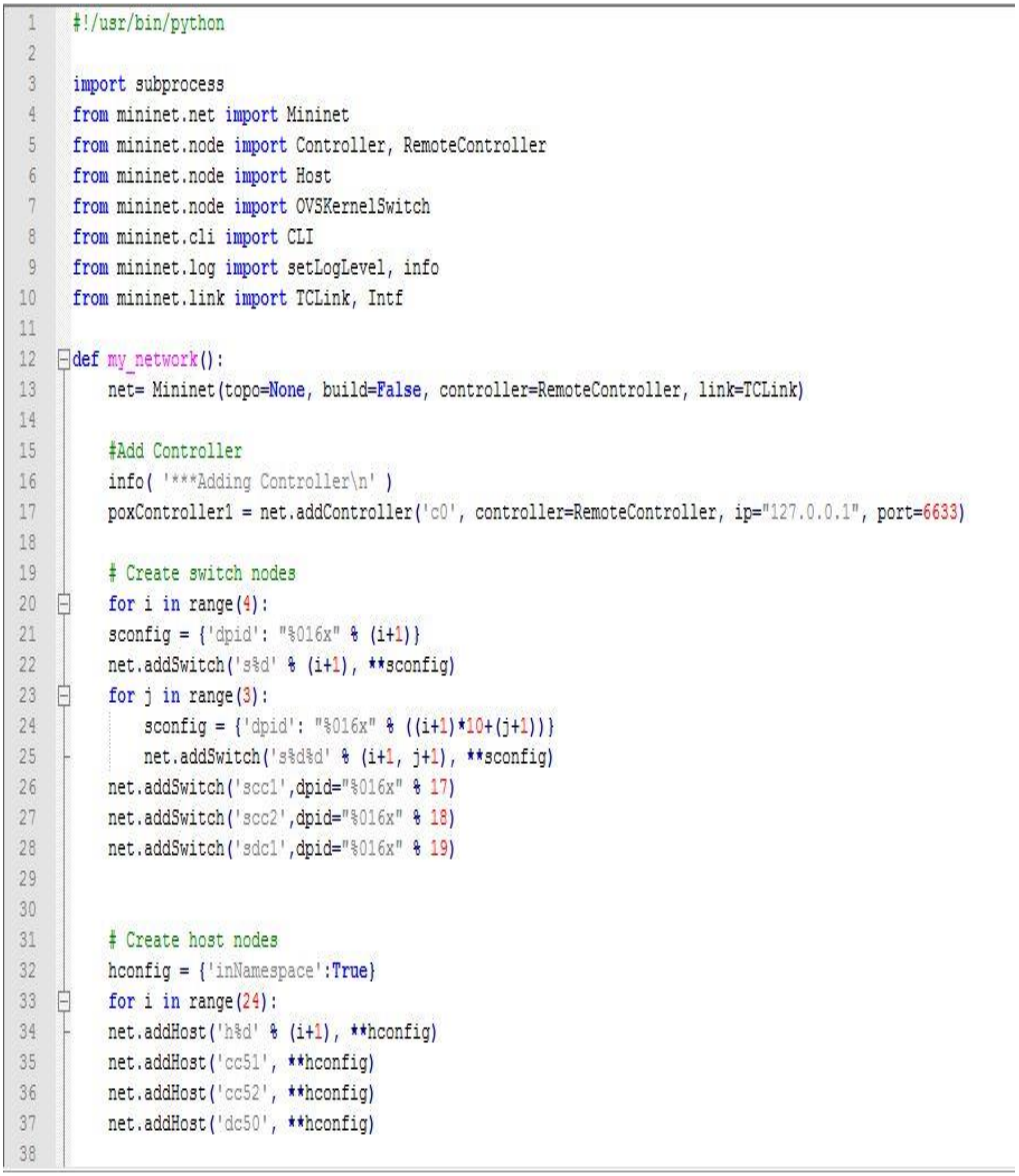
[21] I. Stoica, H. Zhang, and T. S. E. Ng, “A hierarchical fair service curve algorithm for link-sharing, real-time and priority services,” *ACM SIGCOMM Comput. Commun. Rev.*, vol. 27, no. 4, pp. 249–262, Oct. 1997.

[22] M. F. Bari, R. Boutaba, R. Esteves, L. Z. Granville, M. Podlesny, M. G. Rabbani, Q. Zhang, and M. F. Zhani, “Data Center Network Virtualization: A Survey,” *IEEE Commun. Surv. Tutor.*, vol. 15, no. 2, pp. 909–928, 2013.

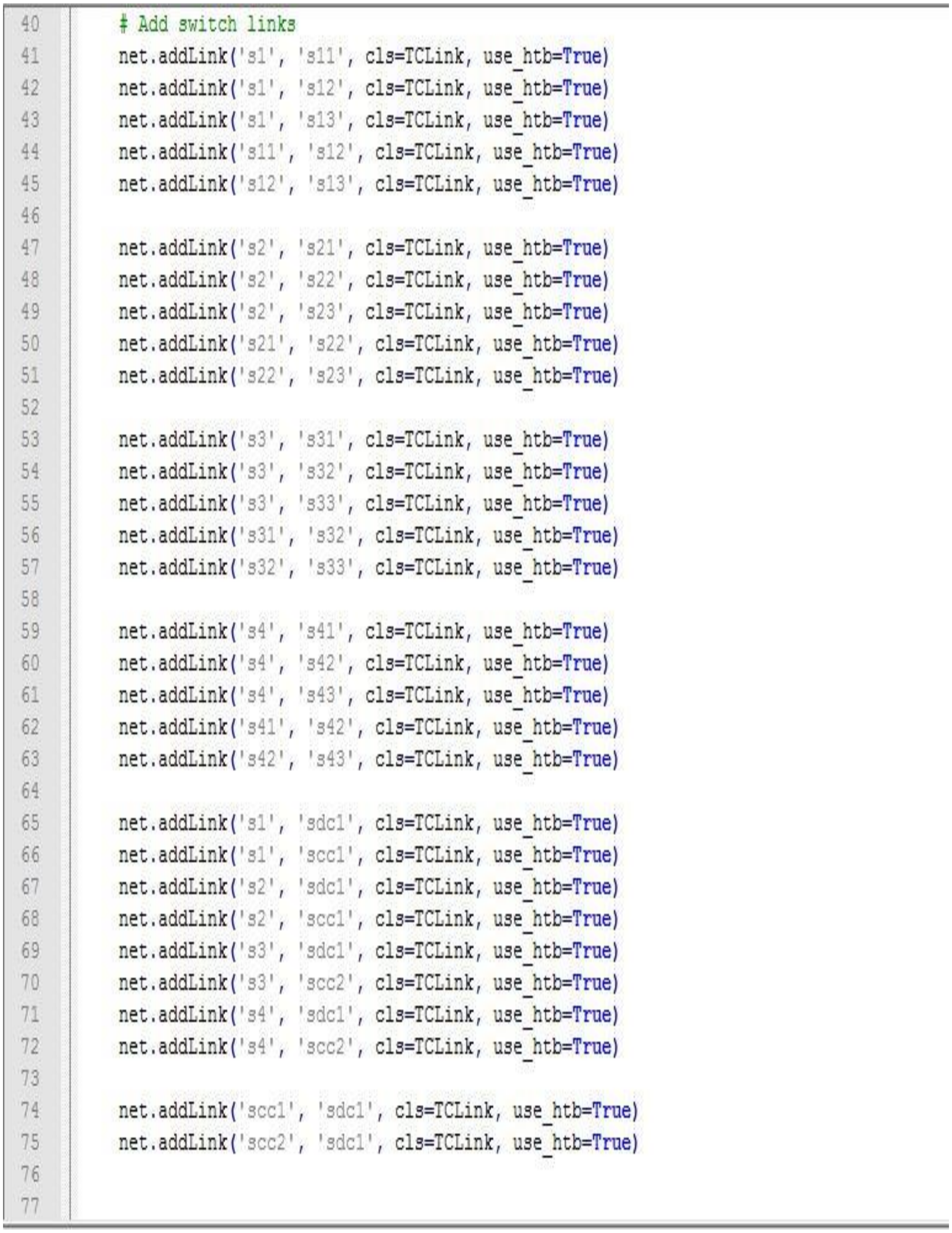
[23] An instant virtual network on your laptop, Mininet,Feb.2015

[24] Rodrigues Prete, L.; Schweitzer, C.M.; Shinoda, A.A.; Santos de Oliveira, R.L., "Simulation in an SDN network scenario using the POX Controller," *Communications and Computing (COLCOM), 2014 IEEE Colombian Conference on* , vol., no., pp.1,6, 4-6 June 2014

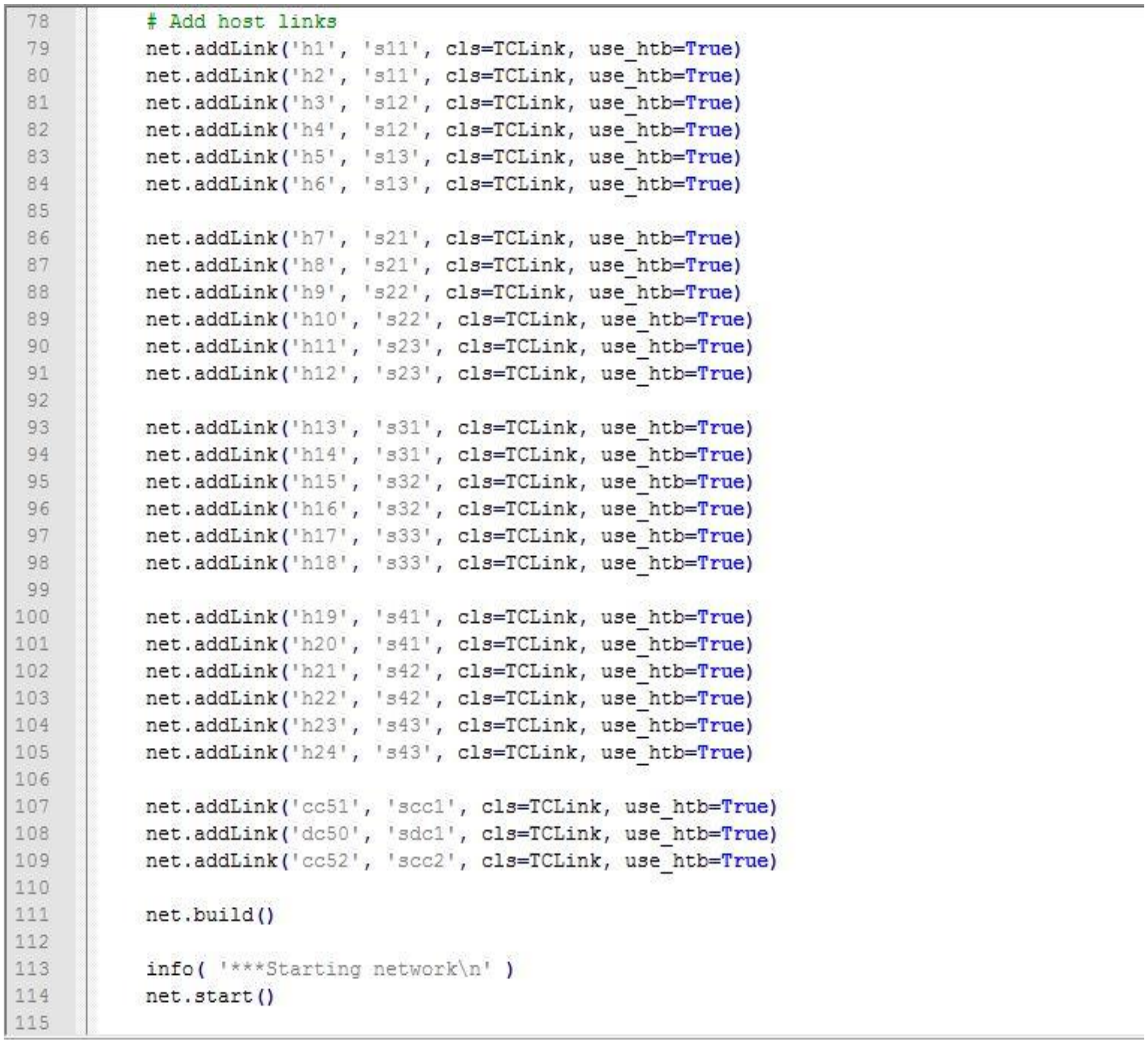
APPENDIX A – Custom Topology Design



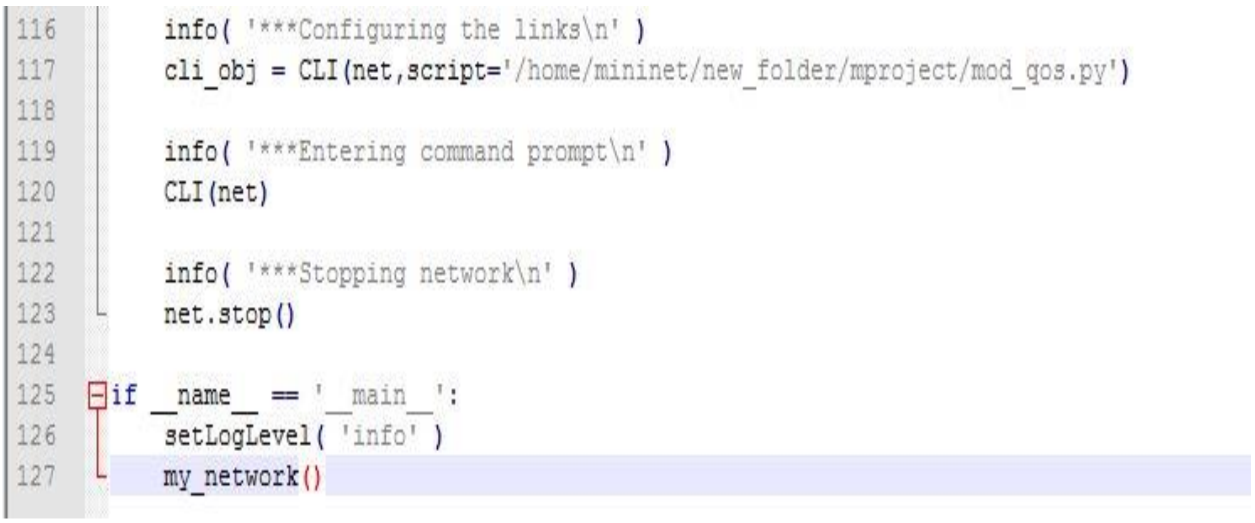
```
#!/usr/bin/python

import subprocess
from mininet.net import Mininet
from mininet.node import Controller, RemoteController
from mininet.node import Host
from mininet.node import OVSKernelSwitch
from mininet.cli import CLI
from mininet.log import setLogLevel, info
from mininet.link import TCLink, Intf

def my_network():
    net= Mininet(topo=None, build=False, controller=RemoteController, link=TCLink)

    #Add Controller
    info( '***Adding Controller\n' )
    poxController1 = net.addController('c0', controller=RemoteController, ip="127.0.0.1", port=6633)

    # Create switch nodes
    for i in range(4):
    sconfig = {'dpid': "%016x" % (i+1)}
    net.addSwitch('s%d' % (i+1), **sconfig)
    for j in range(3):
        sconfig = {'dpid': "%016x" % ((i+1)*10+(j+1))}
        net.addSwitch('s%d%d' % (i+1, j+1), **sconfig)
    net.addSwitch('scc1',dpid="%016x" % 17)
    net.addSwitch('scc2',dpid="%016x" % 18)
    net.addSwitch('sdc1',dpid="%016x" % 19)


    # Create host nodes
    hconfig = {'inNamespace':True}
    for i in range(24):
    net.addHost('h%d' % (i+1), **hconfig)
    net.addHost('cc51', **hconfig)
    net.addHost('cc52', **hconfig)
    net.addHost('dc50', **hconfig)

```



```
    # Add switch links
    net.addLink('s1', 's11', cls=TCLink, use_htb=True)
    net.addLink('s1', 's12', cls=TCLink, use_htb=True)
    net.addLink('s1', 's13', cls=TCLink, use_htb=True)
    net.addLink('s11', 's12', cls=TCLink, use_htb=True)
    net.addLink('s12', 's13', cls=TCLink, use_htb=True)

    net.addLink('s2', 's21', cls=TCLink, use_htb=True)
    net.addLink('s2', 's22', cls=TCLink, use_htb=True)
    net.addLink('s2', 's23', cls=TCLink, use_htb=True)
    net.addLink('s21', 's22', cls=TCLink, use_htb=True)
    net.addLink('s22', 's23', cls=TCLink, use_htb=True)

    net.addLink('s3', 's31', cls=TCLink, use_htb=True)
    net.addLink('s3', 's32', cls=TCLink, use_htb=True)
    net.addLink('s3', 's33', cls=TCLink, use_htb=True)
    net.addLink('s31', 's32', cls=TCLink, use_htb=True)
    net.addLink('s32', 's33', cls=TCLink, use_htb=True)

    net.addLink('s4', 's41', cls=TCLink, use_htb=True)
    net.addLink('s4', 's42', cls=TCLink, use_htb=True)
    net.addLink('s4', 's43', cls=TCLink, use_htb=True)
    net.addLink('s41', 's42', cls=TCLink, use_htb=True)
    net.addLink('s42', 's43', cls=TCLink, use_htb=True)

    net.addLink('s1', 'sdc1', cls=TCLink, use_htb=True)
    net.addLink('s1', 'scc1', cls=TCLink, use_htb=True)
    net.addLink('s2', 'sdc1', cls=TCLink, use_htb=True)
    net.addLink('s2', 'scc1', cls=TCLink, use_htb=True)
    net.addLink('s3', 'sdc1', cls=TCLink, use_htb=True)
    net.addLink('s3', 'scc2', cls=TCLink, use_htb=True)
    net.addLink('s4', 'sdc1', cls=TCLink, use_htb=True)
    net.addLink('s4', 'scc2', cls=TCLink, use_htb=True)

    net.addLink('scc1', 'sdc1', cls=TCLink, use_htb=True)
    net.addLink('scc2', 'sdc1', cls=TCLink, use_htb=True)
```

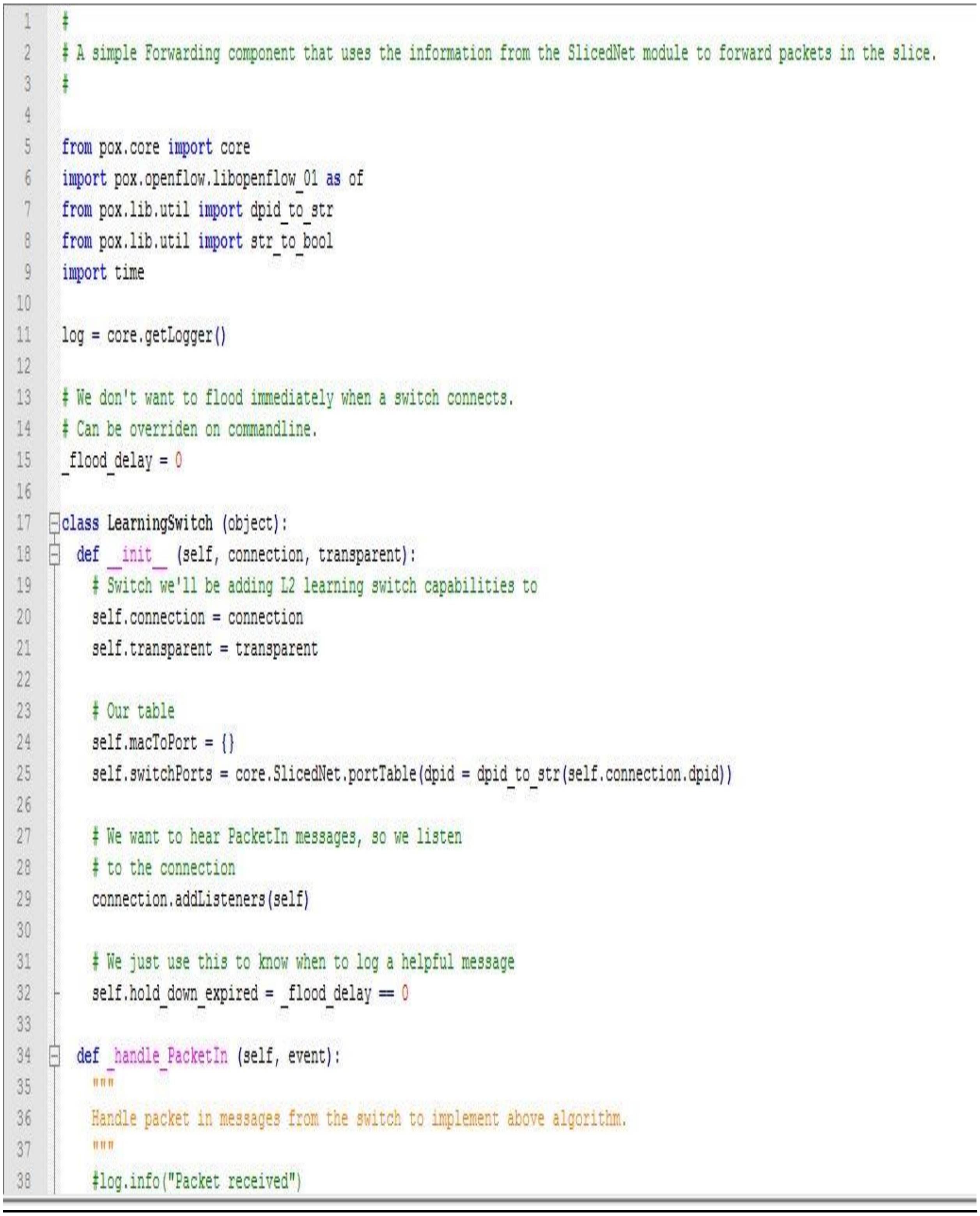


```
    # Add host links
    net.addLink('h1', 's11', cls=TCLink, use_htb=True)
    net.addLink('h2', 's11', cls=TCLink, use_htb=True)
    net.addLink('h3', 's12', cls=TCLink, use_htb=True)
    net.addLink('h4', 's12', cls=TCLink, use_htb=True)
    net.addLink('h5', 's13', cls=TCLink, use_htb=True)
    net.addLink('h6', 's13', cls=TCLink, use_htb=True)

    net.addLink('h7', 's21', cls=TCLink, use_htb=True)
    net.addLink('h8', 's21', cls=TCLink, use_htb=True)
    net.addLink('h9', 's22', cls=TCLink, use_htb=True)
    net.addLink('h10', 's22', cls=TCLink, use_htb=True)
    net.addLink('h11', 's23', cls=TCLink, use_htb=True)
    net.addLink('h12', 's23', cls=TCLink, use_htb=True)

    net.addLink('h13', 's31', cls=TCLink, use_htb=True)
    net.addLink('h14', 's31', cls=TCLink, use_htb=True)
    net.addLink('h15', 's32', cls=TCLink, use_htb=True)
    net.addLink('h16', 's32', cls=TCLink, use_htb=True)
    net.addLink('h17', 's33', cls=TCLink, use_htb=True)
    net.addLink('h18', 's33', cls=TCLink, use_htb=True)

    net.addLink('h19', 's41', cls=TCLink, use_htb=True)
    net.addLink('h20', 's41', cls=TCLink, use_htb=True)
    net.addLink('h21', 's42', cls=TCLink, use_htb=True)
    net.addLink('h22', 's42', cls=TCLink, use_htb=True)
    net.addLink('h23', 's43', cls=TCLink, use_htb=True)
    net.addLink('h24', 's43', cls=TCLink, use_htb=True)

    net.addLink('cc51', 'scc1', cls=TCLink, use_htb=True)
    net.addLink('dc50', 'sdc1', cls=TCLink, use_htb=True)
    net.addLink('cc52', 'scc2', cls=TCLink, use_htb=True)

    net.build()

    info( '***Starting network\n' )
    net.start()

```

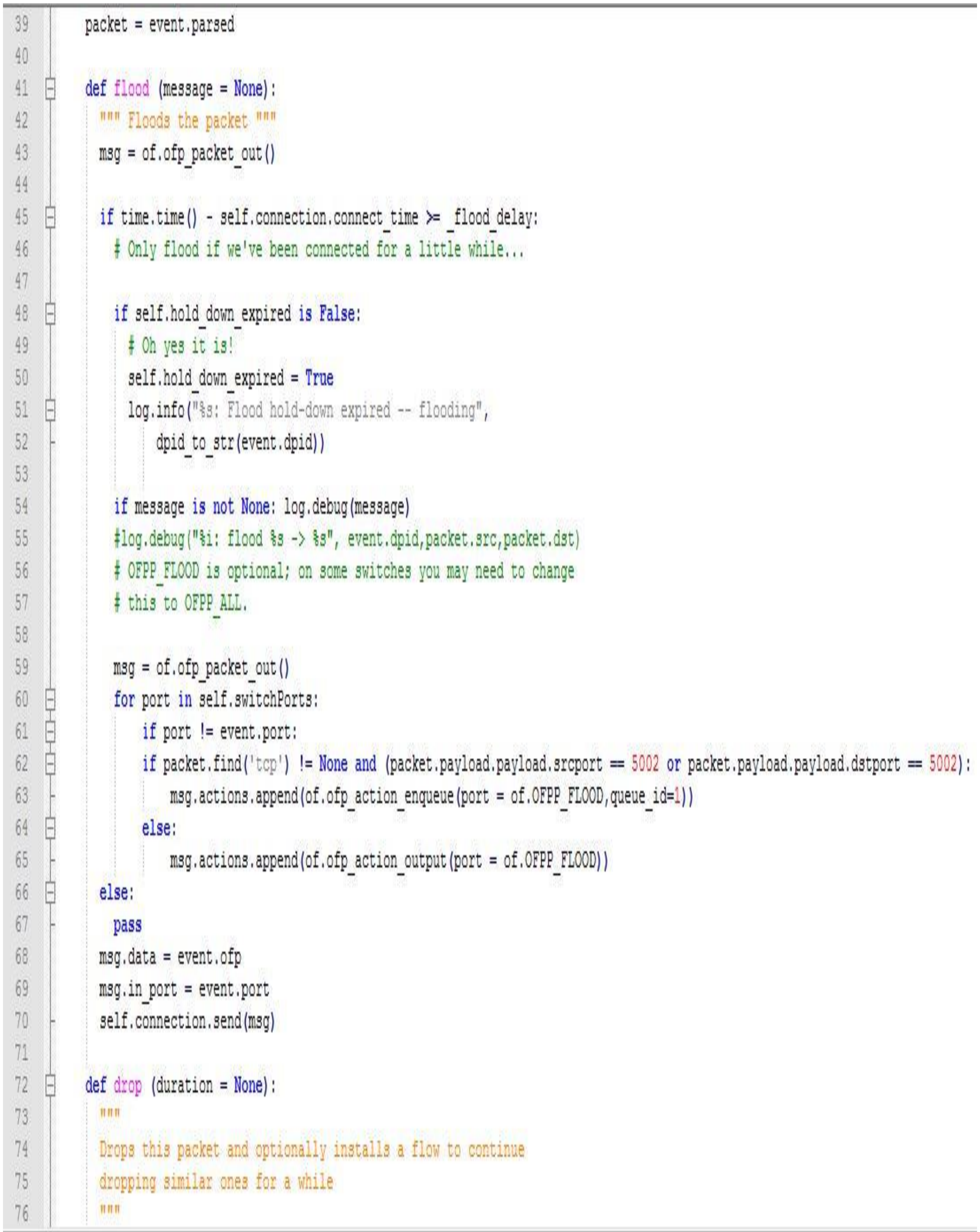


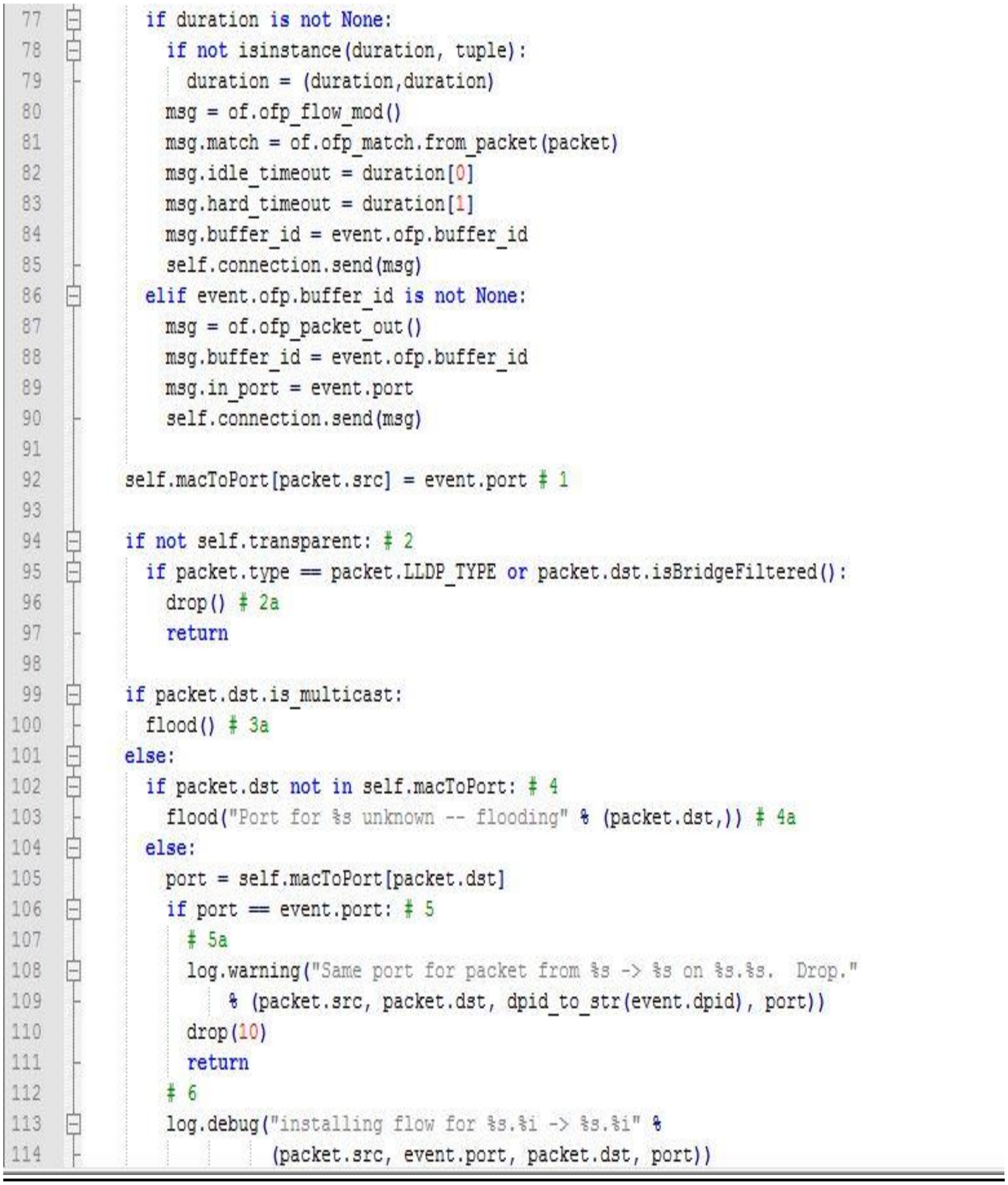
```
    info( '***Configuring the links\n' )
    cli_obj = CLI(net,script='/home/mininet/new_folder/mproject/mod_qos.py')

    info( '***Entering command prompt\n' )
    CLI(net)

    info( '***Stopping network\n' )
    net.stop()

if __name__ == '__main__':
    setLogLevel( 'info' )
    my_network()
```

APPENDIX B – layer 2 learning bridge Code

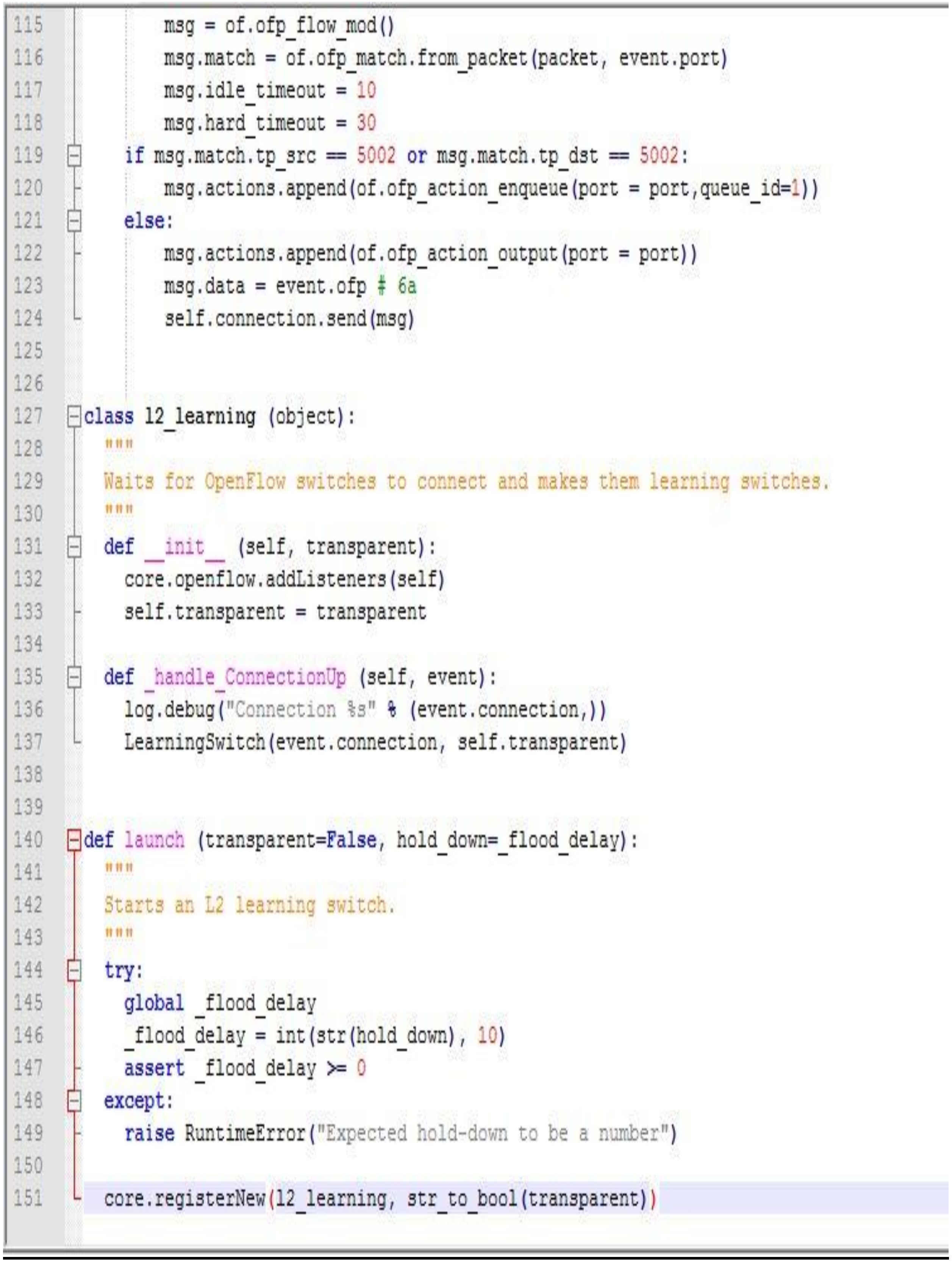


```
#
# A simple Forwarding component that uses the information from the SlicedNet module to forward packets in the slice.
#

from pox.core import core
import pox.openflow.libopenflow_01 as of
from pox.lib.util import dpid_to_str
from pox.lib.util import str_to_bool
import time

log = core.getLogger()

# We don't want to flood immediately when a switch connects.
# Can be overriden on commandline.
_flood_delay = 0

class LearningSwitch (object):
  def __init__ (self, connection, transparent):
    # Switch we'll be adding L2 learning switch capabilities to
    self.connection = connection
    self.transparent = transparent

    # Our table
    self.macToPort = {}
    self.switchPorts = core.SlicedNet.portTable(dpid = dpid_to_str(self.connection.dpid))

    # We want to hear PacketIn messages, so we listen
    # to the connection
    connection.addListeners(self)

    # We just use this to know when to log a helpful message
    self.hold_down_expired = _flood_delay == 0

  def _handle_PacketIn (self, event):
    """
    Handle packet in messages from the switch to implement above algorithm.
    """
    #log.info("Packet received")
```

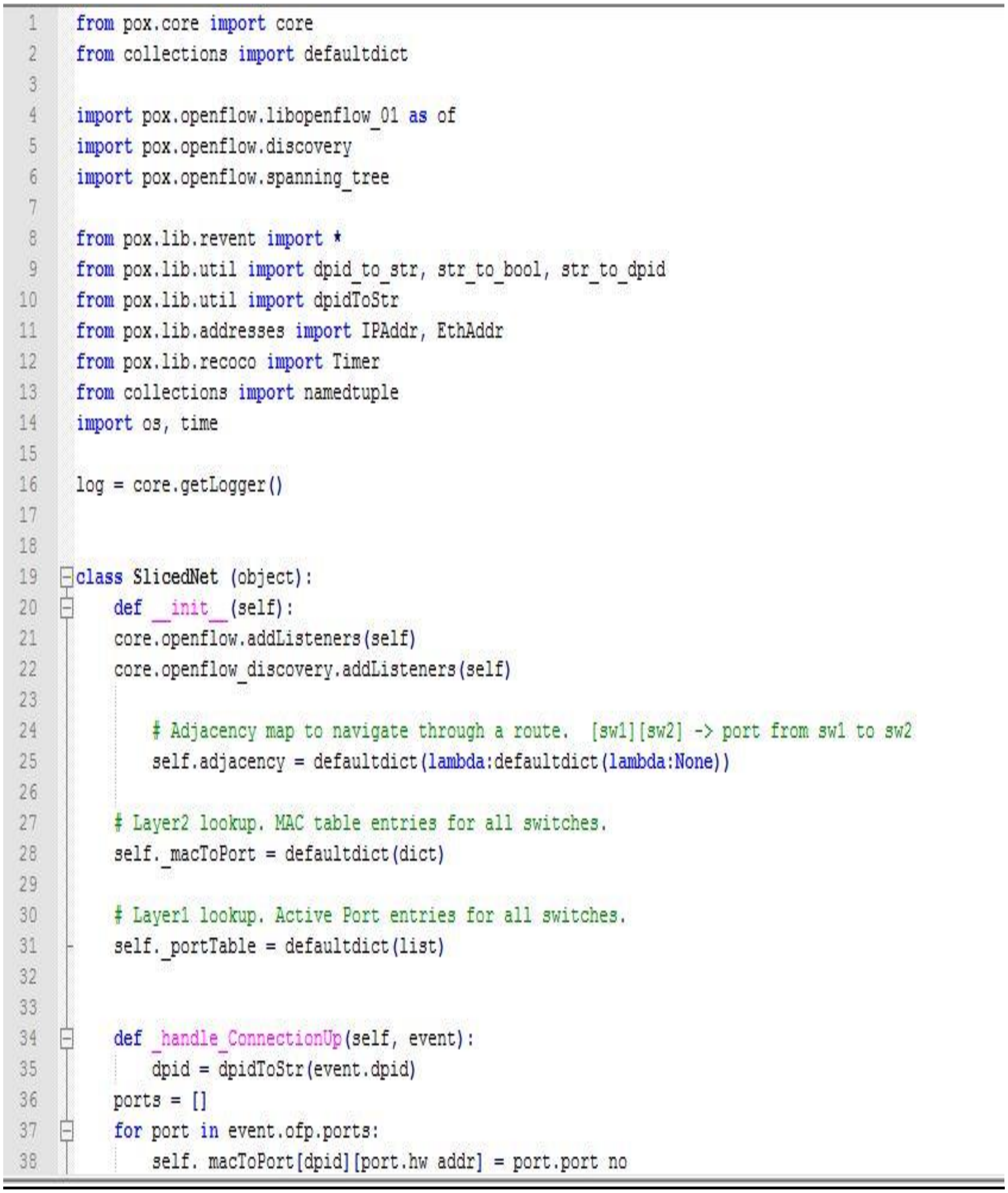

```
    packet = event.parsed

    def flood (message = None):
      """ Floods the packet """
      msg = of.ofp_packet_out()

      if time.time() - self.connection.connect_time >= _flood_delay:
        # Only flood if we've been connected for a little while...

        if self.hold_down_expired is False:
          # Oh yes it is!
          self.hold_down_expired = True
          log.info("%s: Flood hold-down expired -- flooding",
              dpid_to_str(event.dpid))

        if message is not None: log.debug(message)
        #log.debug("%i: flood %s -> %s", event.dpid,packet.src,packet.dst)
        # OFPP_FLOOD is optional; on some switches you may need to change
        # this to OFPP_ALL.

        msg = of.ofp_packet_out()
        for port in self.switchPorts:
            if port != event.port:
            if packet.find('tcp') != None and (packet.payload.payload.srcport == 5002 or packet.payload.payload.dstport == 5002):
                msg.actions.append(of.ofp_action_enqueue(port = of.OFPP_FLOOD,queue_id=1))
            else:
                msg.actions.append(of.ofp_action_output(port = of.OFPP_FLOOD))
      else:
        pass
      msg.data = event.ofp
      msg.in_port = event.port
      self.connection.send(msg)

    def drop (duration = None):
      """
      Drops this packet and optionally installs a flow to continue
      dropping similar ones for a while
      """
```



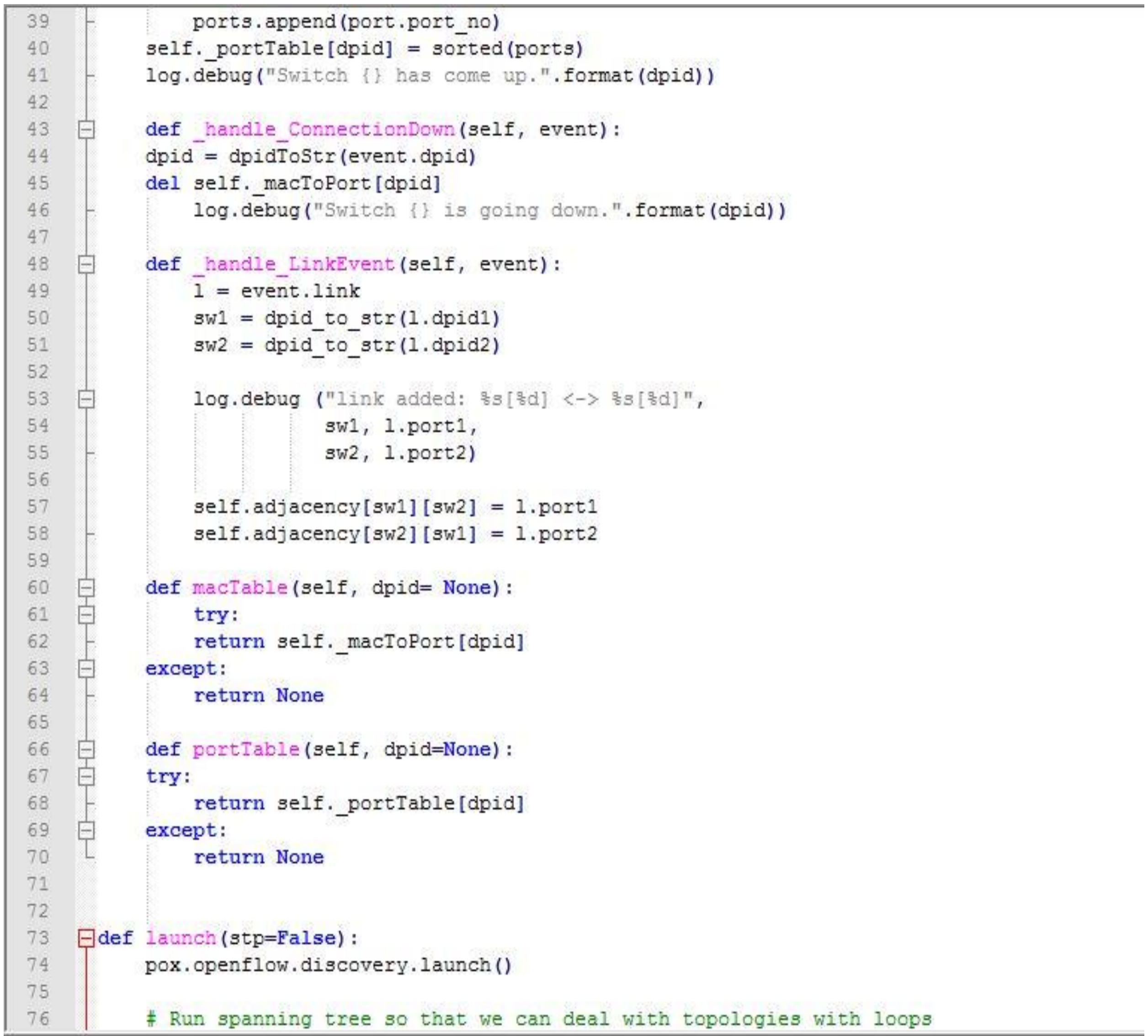
```
      if duration is not None:
        if not isinstance(duration, tuple):
          duration = (duration,duration)
        msg = of.ofp_flow_mod()
        msg.match = of.ofp_match.from_packet(packet)
        msg.idle_timeout = duration[0]
        msg.hard_timeout = duration[1]
        msg.buffer_id = event.ofp.buffer_id
        self.connection.send(msg)
      elif event.ofp.buffer_id is not None:
        msg = of.ofp_packet_out()
        msg.buffer_id = event.ofp.buffer_id
        msg.in_port = event.port
        self.connection.send(msg)

    self.macToPort[packet.src] = event.port # 1

    if not self.transparent: # 2
      if packet.type == packet.LLDP_TYPE or packet.dst.isBridgeFiltered():
        drop() # 2a
        return

    if packet.dst.is_multicast:
      flood() # 3a
    else:
      if packet.dst not in self.macToPort: # 4
        flood("Port for %s unknown -- flooding" % (packet.dst,)) # 4a
      else:
        port = self.macToPort[packet.dst]
        if port == event.port: # 5
          # 5a
          log.warning("Same port for packet from %s -> %s on %s.%s.  Drop."
              % (packet.src, packet.dst, dpid_to_str(event.dpid), port))
          drop(10)
          return
        # 6
        log.debug("installing flow for %s.%i -> %s.%i" %
                  (packet.src, event.port, packet.dst, port))
```


```
        msg = of.ofp_flow_mod()
        msg.match = of.ofp_match.from_packet(packet, event.port)
        msg.idle_timeout = 10
        msg.hard_timeout = 30
    if msg.match.tp_src == 5002 or msg.match.tp_dst == 5002:
        msg.actions.append(of.ofp_action_enqueue(port = port,queue_id=1))
    else:
        msg.actions.append(of.ofp_action_output(port = port))
        msg.data = event.ofp # 6a
        self.connection.send(msg)


class l2_learning (object):
  """
  Waits for OpenFlow switches to connect and makes them learning switches.
  """
  def __init__ (self, transparent):
    core.openflow.addListeners(self)
    self.transparent = transparent

  def _handle_ConnectionUp (self, event):
    log.debug("Connection %s" % (event.connection,))
    LearningSwitch(event.connection, self.transparent)


def launch (transparent=False, hold_down=_flood_delay):
  """
  Starts an L2 learning switch.
  """
  try:
    global _flood_delay
    _flood_delay = int(str(hold_down), 10)
    assert _flood_delay >= 0
  except:
    raise RuntimeError("Expected hold-down to be a number")

  core.registerNew(l2_learning, str_to_bool(transparent))
```

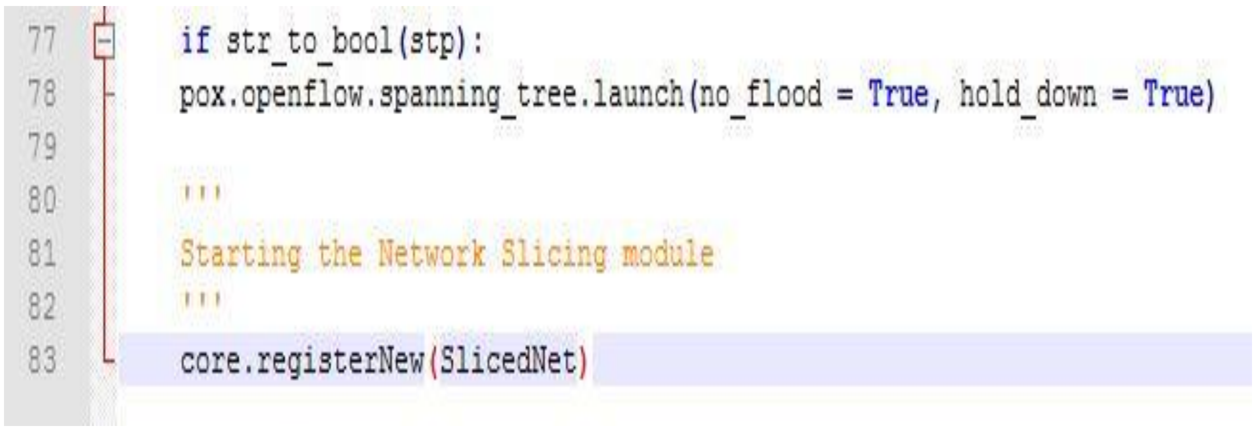
APPENDIX C – Slicing code and spanning tree



```
from pox.core import core
from collections import defaultdict

import pox.openflow.libopenflow_01 as of
import pox.openflow.discovery
import pox.openflow.spanning_tree

from pox.lib.revent import *
from pox.lib.util import dpid_to_str, str_to_bool, str_to_dpid
from pox.lib.util import dpidToStr
from pox.lib.addresses import IPAddr, EthAddr
from pox.lib.recoco import Timer
from collections import namedtuple
import os, time

log = core.getLogger()


class SlicedNet (object):
    def __init__(self):
    core.openflow.addListeners(self)
    core.openflow_discovery.addListeners(self)

        # Adjacency map to navigate through a route.  [sw1][sw2] -> port from sw1 to sw2
        self.adjacency = defaultdict(lambda:defaultdict(lambda:None))

    # Layer2 lookup. MAC table entries for all switches.
    self._macToPort = defaultdict(dict)

    # Layer1 lookup. Active Port entries for all switches.
    self._portTable = defaultdict(list)


    def _handle_ConnectionUp(self, event):
        dpid = dpidToStr(event.dpid)
    ports = []
    for port in event.ofp.ports:
        self._macToPort[dpid][port.hw_addr] = port.port_no
```



```
            ports.append(port.port_no)
        self._portTable[dpid] = sorted(ports)
        log.debug("Switch {} has come up.".format(dpid))

    def _handle_ConnectionDown(self, event):
        dpid = dpidToStr(event.dpid)
        del self._macToPort[dpid]
            log.debug("Switch {} is going down.".format(dpid))

    def _handle_LinkEvent(self, event):
        l = event.link
        sw1 = dpid_to_str(l.dpid1)
        sw2 = dpid_to_str(l.dpid2)

        log.debug ("link added: %s[%d] <-> %s[%d]",
                   sw1, l.port1,
                   sw2, l.port2)

        self.adjacency[sw1][sw2] = l.port1
        self.adjacency[sw2][sw1] = l.port2

    def macTable(self, dpid= None):
        try:
        return self._macToPort[dpid]
    except:
        return None

    def portTable(self, dpid=None):
    try:
        return self._portTable[dpid]
    except:
        return None


def launch(stp=False):
    pox.openflow.discovery.launch()

    # Run spanning tree so that we can deal with topologies with loops
```



```
    if str_to_bool(stp):
    pox.openflow.spanning_tree.launch(no_flood = True, hold_down = True)

    '''
    Starting the Network Slicing module
    '''
    core.registerNew(SlicedNet)
```