POXVine : Multi-Tenant Virtual Network Emulator

B.Tech. Project 2nd Stage Report

Submitted in partial fulfilment of the requirements for the degree of

Bachelor of Technology

Student:
Kausik Subramanian
Roll No: 110050003

Guide:
Purushottam Kulkarni

Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Mumbai 400076, India

Abstract

Abstract The advent of cloud computing and software defined networks allows tenants to run huge and complex network topologies on shared infrastructure. We built a multi-tenant virtual network emulation application *POXVine* using the POX controller which controls a network over Mininet. In this report, we demonstrate the design and implementation of POXVine.

Acknowledgement

I wish to express my sincere gratitude and indebtedness to my guide, Purushottam Kulkarni and Umesh Bellur for their constant support and guidance throughout the project. I am also indebted to Martin Casado, whose talk on Software defined Networks at IIT Bombay inspired me to pursue research in the field.

Kausik Subramanian
B.Tech. IV
CSE, IIT Bombay

Contents

Chapter 1

Introduction

1.1 Software Defined Networks

Software defined Networks is an emerging architecture which provides a framework to manage network services through the abstraction of lower level functionality. In SDNs, the *control plane* which makes the decision of where to send traffic is decoupled from the *data plane*, which performs the actual forwarding of traffic. The advantages of SDN over traditional network architectures are listed below.

- *Central State* : The entire state of the network and name bindings exist in a central location, called the controller. All inputs from the network are passed to the controller, which decides the policy needs to be implemented.

- *Decoupled Control and Forwarding* : In SDNs, the control plane is separated from the data plane. The controller performs the route computation and push the forwarding rules to the switches. The switches perform the forwarding of packets.

- *Software Controller* : In SDNs, the controller is implemented in software, so can be modified to implement any kind of policy. The switches perform basic forwarding and expose a common API for the controller to talk to them. Because the control plane is in software, changes in network protocols and services are easier to implement without a major hardware overhaul.

1.2 POX

POX [1] is a single-threaded Python-based controller. It is widely used for fast prototyping of network applications in research. At its core, its a platform for the rapid development and prototyping of network control software. Convenient to setup up for research experiments makes POX an excellent SDN controller to play with and develop.

1.3 Network emulation using Mininet

Mininet [2] is a network emulator which creates a network of virtual hosts, switches, controllers, and links. Mininet hosts run standard Linux network software, and its switches support OpenFlow for highly flexible custom routing and Software-Defined Networking. Mininet provides a simple and inexpensive network testbed for developing OpenFlow applications, with the need of an actual physical OpenFlow enabled topology. Mininet has been accelerating research on Software Defined Networking.

1.4 Organization of the Report

The rest of the report is organized as follows: In Chapter 2, we discuss the problem description and inspiration and use cases of POXVine. We discuss related work in Chapter 3. In Chapter 4, we describe the design of POXVine system and Implementation in Chapter 5. We present some experiments using POXVine in Chapter 6 and present future work in Chapter 7.

Chapter 2

Problem Description

With the advent of software-defined networks, data-centers allow *tenants* its own network topology and control over the flow of its traffic. The mapping of virtual to physical topologies can be done with respect to various parameters, allowing efficient resource utilisation of the data-center.
Another use of network virtualization is that tenants can do a test-run of a network before deploying it in real life. Thus, a network testbed can be extremely useful. Developing a network testbed using SDNs due to its standard interface between controller applications and switch forwarding tables, which give great flexibility over the network. However, there is a lack of a physical SDN, thus we decided to use Mininet as the physical software defined network infrastructure to create virtual networks. With these motive, we developed *POXVine*: POX Virtual Network Emulator (POX-VI-N-E). POXVine takes input the topology specifications of the physical and virtual topologies, and produces a emulated network on Mininet. We describe the exact topology specifications in the next section.

2.0.1 Topology Specification

Hosts

For specifying the configuration of hosts, POXVine uses RAM size (in gigabytes) of host for deciding the placement of vrtual hosts on each host, i.e a virtual host with RAM size less than remaining capacity of a physical host can be placed on the host. We can safely disregard disk storage for VM placement (can use Network Storage). We specify the IP address of the host in the configuration. The last field in the configuration is the switch this host is connected in the topology(physical or virtual). One assumption is that a host is only connected to a single switch in the network.

```
<server-name>  <RAM-size>  <ip-address>  <switch-name>
```

Switches

POXVine uses Mininet to emulate OpenFlow-enabled switches. To simulate real network constraints, one of the important resources in software-defined networks is flow table size, especially in multi-tenant networks with thousands of hosts. Certain switches in the network can become a bottleneck due to excess of rules installed at the switch.

```
<switch-name>  <flow-table-size>
```

Links

POXVIne uses Mininet to emulate the links connecting the hosts and switches. Each link is full-duplex (communication both ways) and the configuration specifies both the end-points of the link. Also, another important link parameter is *bandwidth*. POXVine could be used to map virtual tenants providing *minimum guarantees*. We can also have host mappings which try to minimise the bandwidth utilisation of the tenant. Since we are modeling small area networks (datacenters), we do not consider latency.

```
<entity-one-name>  <entity-two-name>  <link-bandwidth>
```

2.0.2 Output

The POXVine system takes the topology configurations as input, and provides a emulated virtual network mapped on the physical network using Mininet. POXVine preserves the *virtual network abstraction* for each tenant network and also, we can analyse the physical network on which the virtual network is mapped.

Chapter 3

Related Work

One of our biggest motivations comes from Flowvisor [3], which slices the network hardware by placing a layer between the control plane and the data plane. Using Flowvisor, we can split the network traffic into slices, and each slice can have its control logic. Using this, we can build a network testbed that is embedded in the physical network.
Flowvisor allows network researchers to test new protocols on production networks. It provides isolation between slices, so changes in one slice's control logic will not affect the other slices. Figure 3.1 illustrates the Flowvisor architecture.

Another work in this area is FlowN [4], which deals with scalable Network Virtualization in software-defined networks. FlowN allows every tenant to run its own controller. The controllers are virtualized, so that the tenants' traffic are handled by the respective controller. For scalability purposes, FlowN uses databases to store the *virtual-to-physical topology mappings*, thus capitalizing the years of research to achieve durable state and a highly scalable system.

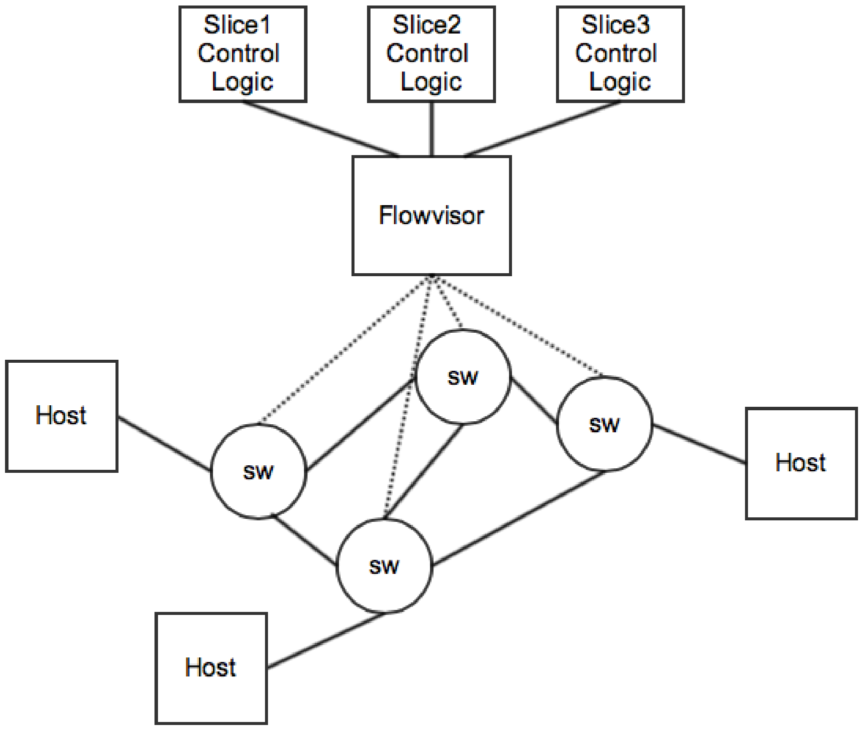

Figure 3.1: Flowvisor Architecture

Chapter 4

System Design

POXVine consists of three main components.

- The *host mapper* is responsible to map the virtual network entities (hosts and switches) onto the physical topology. This mapping can be done based on different heuristics, so POXVine allows you to customize the host mapper. We have developed a host mapper *MinSwitchMapper*, which tries to minimize the number of *physical switches* which contain rules to the virtual topology.

- The *network Mapper* is an application built over the *POX* controller which uses the *virtual-to-physical* mappings to add the required routing OpenFlow rules on the mininet switches, so that the virtual hosts can talk to one other. Another important design consideration is that the *virtual network abstraction* must be preserved, that is, if a packet is to flow across a route in the virtual topology, on the physical topology, it must traverse the virtual network entities in the same order.

- The *Mininet* infrastructure is used to emulate the physical network topology and the virtual hosts which are connected to the emulated physical switches (according to the *virtual-to-physical*) mappings).

I explain the individual components in the coming sections.

4.1 MinSwitchMapper

The host mapper module is responsible for finding the *virtual-to-physical* mappings of the virtual network entities, i.e on which physical hosts, the virtual hosts and switches are placed. This mapping can be done according to various considerations, like *maximizing number of virtual hosts, minimizing*

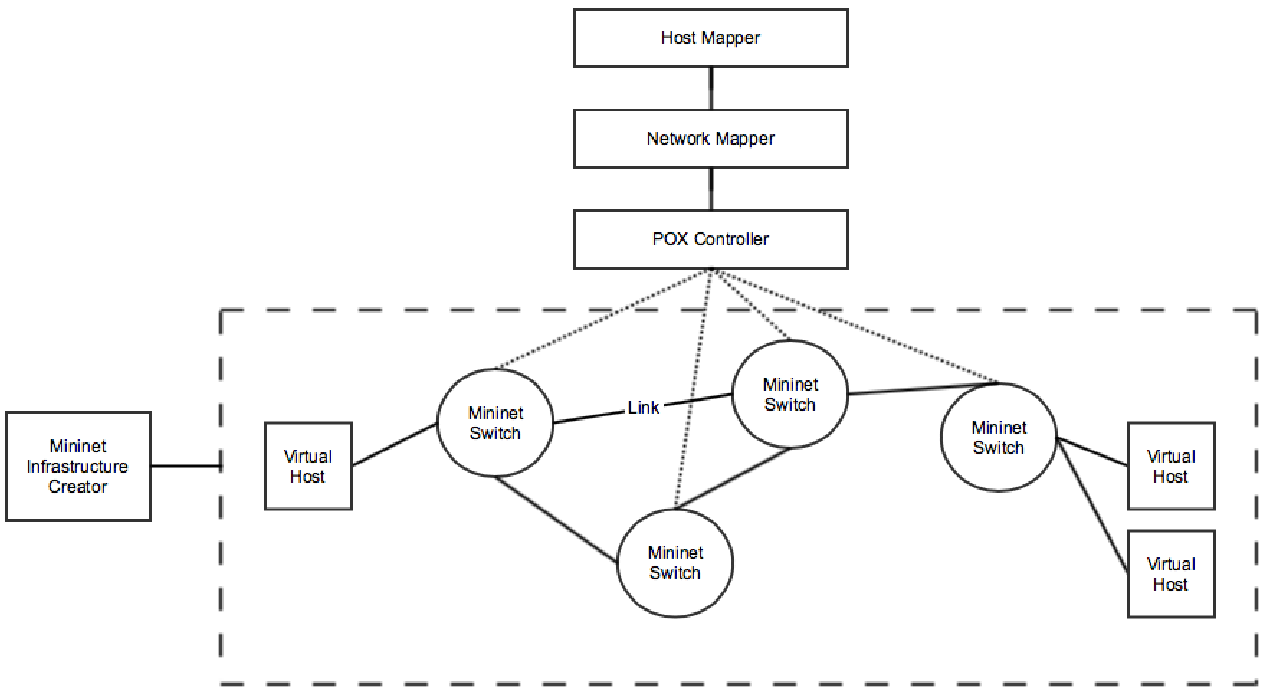

Figure 4.1: POXVine Architecture

the number of switches mapped to a virtual topology, greedy host allocation, providing bandwidth guarantees etc.

All the virtual network entities are mapped to physical hosts, which are connected by the physical network topology. Consider the network graph which is formed by using only the switches and links that are required to connect all the physical hosts (we use the shortest path between two hosts in the network graph). Figure 2.2 demonstrates an example of such a graph.

We have developed *MinSwitchMapper*, which minimises the diameter of the network graph connecting the hosts of the virtual topology, The basis for this heuristic is that the traffic of this tenant's hosts are confined to the *smallest portion* in the physical topology. This also minimises the number of switches where rules regarding this virtual topology is installed, thus increasing the number of tenants we can accomodate in POXVine (provided the physical host capacity is not insufficient)

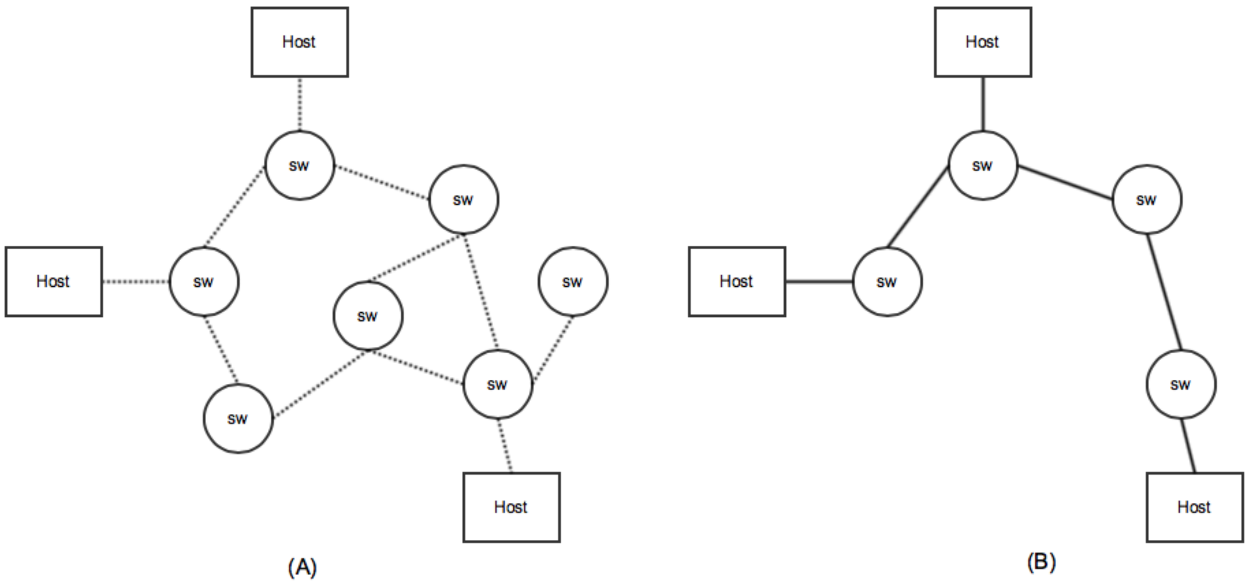

Figure 4.2: Example of the Network Graph Heuristic. (A) shows the physical network topology. Suppose if the virtual hosts are mapped to the hosts as shown in (A). The network graph consisting of shortest paths to all the hosts are shown in (B)

4.2 NetworkMapper

The *NetworkMapper* module is an application built on top of the POX controller. The role of the NetworkMapper is to use the *virtual-to-physical mappings* generated by the host mapper module and add the required routing rules on the mininet switches for the virtual hosts of a tenant. One important design decision is that the NetworkMapper preserves the *virtual network abstraction*. Let us suppose there are two virtual hosts *v1* and *v2*, connected by a path of two switches *vs1* and *vs2*, i.e $v1 \rightarrow vs1 \rightarrow vs2 \rightarrow v2$. Irrespective of the mapping, the Network Mapper must add the rules such that traffic from $v1 \rightarrow v2$ must traverse through $vs1$, $vs2$ and $v2$ in that order.

4.2.1 Route Tagging

Consider the two virtual hosts *v1* and *v2*, connected by the following path in the virtual topology.

$$v1 \rightarrow vs1 \rightarrow vs2 \rightarrow v2$$

Consider the mapping as shown in Figure 2.3. The path taken by a packet from $v1$ to $v2$ must go to $vs1$, then $vs2$ then $v2$. At switch s3, we need three

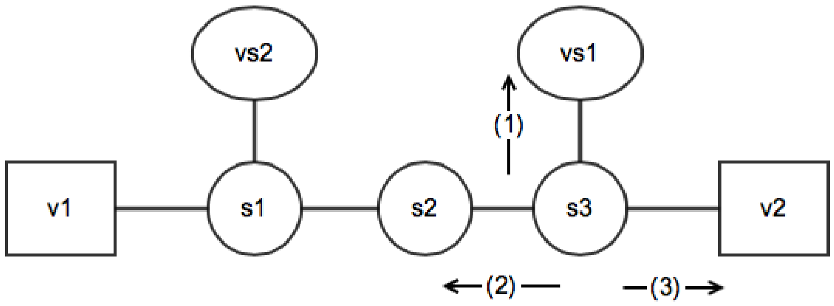

Figure 4.3: Mapping of *v1, v2, vs1 and vs2.* (1),(2) and (3) depict the three different rules a packet from $v1$ to $v2$ needs at switch $s3$.

different rules for this packet.

1. The packet from $v1$ reaches $s3$ for the first time. The packet is sent out to $vs1$.

2. The packet is received from $vs1$. The packet is sent to out to $s2$ and will be subsequently sent to $vs2$

3. The packet is received after traversing $vs2$. The packet is sent out to $v2$.

Therefore, the rules added at $s3$ cannot differentiate these three kind of flows using just the IP headers. For this, we incorporate *RouteTags* in the packet header [5]. In our case, the VLAN ID header field is used to store both the *tenant ID* and the *RouteTag*. Using the RouteTag, we can differentiate which part of the route the packet is in. Listing an example of the rules on switch $s3$.

$$\begin{gathered}
\text{Rule 1} \rightarrow \\
\textit{Match}: \text{IP Src}{=}v1 \mid \text{IP Dst}{=}v2 \mid \text{RouteTag}{=}1 \\
\textit{Action}: \text{Output}{=}vs1 \mid \text{RouteTag}{=}2 \\
\text{Rule 2} \rightarrow \\
\textit{Match}: \text{IP Src}{=}\ v1 \mid \text{IP Dst}{=}\ v2 \mid \text{RouteTag} = 2 \\
\textit{Action}: \text{Output}{=}s2 \mid \text{RouteTag} = 3 \\
\text{Rule 3} \rightarrow \\
\textit{Match}: \text{IP Src}{=}\ v1 \mid \text{IP Dst}{=}\ v2 \mid \text{RouteTag} = 3 \\
\textit{Action}: \text{Output}{=}v2
\end{gathered}$$

Thus, we identify straight paths in the network route and assign a Route Tag for each of them, thus the switch can distinguish which part of the network route the packet is in. We will look at Route Tag calculation in the next chapter.

4.2.2 Switch Tunnelling

As seen in the previous example, we know that $s3$ needs to have three different rules for the different Route Tag packets. Let us consider switch $s2$. Adding three different rules for $s2$ is wasteful, as for $s2$, the route tag is of no importance. It just sends packets from $s1$ to $s3$ and $s3$ to $s1$. Inspired from [5], we establish switch tunnels in the network. Thus, if we divide the physical network route into *segments* (where the start and end of each segment is connected to a virtual entity), then the switches in the middle of the segment do not need fine grained rules, they need to route the packet to the end-switch of the segment.

Thus, the NetworkMapper adds routing rules for every other switch on each switch. At the start of each *segment*, the rules added modify the source MAC address (unused field for the POXVine system) to indicate the end-switch of the segment. The switches in the middle will just route the packet to that switch. Revisiting the example in Figure 2.3, switch $s1$ will have the following rule.

$$\begin{gathered} Match : \text{IP Src}=v1 \mid \text{IP Dst}=v2 \\ \textit{Action} : \text{Output}=s2 \mid \text{RouteTag}=1 \mid \text{MAC Src}=s3 \end{gathered}$$

Switch $s2$ will have switch tunnel rules to switch $s1$ and $s3$.

$$\begin{gathered} \text{Rule 1} \rightarrow \\ Match : \text{MAC Src}=s3 \\ \textit{Action} : \text{Output}=s3's\ Port\ Number \\ \text{Rule 2} \rightarrow \\ Match : \text{MAC Src}=s1 \\ \textit{Action} : \text{Output}=s1's\ Port\ Number \end{gathered}$$

4.3 Mininet Infrastructure Creator

POXVine uses Mininet to create a *emulated* physical/virtual topology as per the specifications. From the topology configurations and the virtual-to-physical mappings, a hybrid topology configuration is created for the

Mininet Infrastructure Creator, which comprises of all the physical topology's switches and links, and according to the virtual-to-physical mappings, the virtual hosts and switches connected to the corresponding physical switches. We do not create the physical hosts in Mininet. Future versions of POXVine can run the virtual hosts and switches on real 'physical' hosts.

Chapter 5

Implementation

In this chapter, we describe POXVine's implementation in detail.

5.1 MinSwitchMapper

The POXVine System can use any host mapper heuristic to map the virtual hosts. For *modularization*, each host mapper takes input the physical Topology and virtual Topology object and stores the mapping in text files for other modules of the system to use. This also allows us to use different host mappers simultaneously for different tenants.

MinSwitchMapper minimises the diameter of the network graph connecting the hosts of the virtual topology as seen in Chapter 5. Algorithm 1 is used to find the virtual-to-physical mapping which minimises the diameter. The intuition behind the algorithm is that the switches connected to hosts initially carry the capacity of the host. Every round, we *percolate* the capacity to its neighbours, so, the first round we get a switch with capacity exceeding the required capacity, we can terminate the algorithm. A host list obtained in a *later round* will have greater distance among hosts as we percolate to farther hosts each round, increasing the distance, and thus the diameter of the network graph formed by the virtual hosts.

Once the algorithm provides a list of physical hosts with sufficient capacity, we try to map the largest virtual host on the largest physical host, and if we can, repeat the step again till all the virtual hosts are mapped. If such a mapping is not possible, then we reiterate Algorithm 1 to find a different host list to map the virtual hosts.

```
input  : Physical Topology P, Virtual Host List vhosts
output: Virtual-to-physical mappings of each host in vhost
Heuristic: Minimise the constructed network graph.

Initialization:
swlist1 ← []
swlist2 ← []
round ← 0
for host in P.phosts do
    host→switch→hostlist = [host]
    swlist1.append (host→switch)
end
Start the Percolation Rounds.
while MappingNotFound do
    while len (swlist1) $\neq$ 0 do
        sw = swlist1.pop()
        if capacity ( sw→hostlist ) $\geq$ capacity (vhosts) then
            MappingNotFound = False
            mapHosts ( sw→hostlist )
        end
        nlist = neighbours (sw)
        for n in nlist do
            Add all the hosts in sw→hostlist to n→hostlist without
            duplicates.
            If any change in n→hostlist, add n to swlist2
        end
    end
    swlist1 ← swlist2
    swlist2 ← []
    round ← round + 1
end
```
Algorithm 1: Minimum Network Diameter Host Mapping Algorithm

5.2 NetworkMapping

The NetworkMapping class uses the *virtual-to-physical* mappings provided by the host mapper, and calculates all the network routes on the physical topology required for the virtual hosts to talk to each other. Consider the topology in Figure 4.3. The NetworkMapping object will first find the route from $v1$ to $v2$ in the virtual topology, that is $v1 \rightarrow vs1 \rightarrow vs2 \rightarrow v2$.

The Physical Topology object has a method `getCompleteRoute()` which translates the virtual route to the corresponding physical topology route by finding the physical route for each hop in the virtual topology. Thus, from the virtual route, we obtain the following physical route : $v1 \rightarrow s1 \rightarrow s2 \rightarrow s3 \rightarrow vs1 \rightarrow s3 \rightarrow s2 \rightarrow s1 \rightarrow vs2 \rightarrow s1 \rightarrow s2 \rightarrow s3 \rightarrow v2$.

Pseudo Code for Network Route Generation

```
for each pair of virtual hosts{h1, h2}) :
  virtualRoute = virtualTopology.getRoute(h1, h2)
  physicalRoute =
    physicalTopology.getCompleteRoute(virtualRoute)
  physicalRoute.setRouteTags()
  add physicalRoute to networkPaths
end for
```

Note that we need to add rules from $v1$ to $v2$ and the other way as well, $v2$ to $v1$. After generating all the routes for a virtual topology, this is passed to the NetworkMapper module, which translates them into OpenFlow rules to install at the respective switches.

5.3 NetworkMapper

5.3.1 Route Tagging

The NetworkMapper is a POX application. Using the NetworkMapping object, it obtains all the routes it needs to install OpenFlow rules for at the mininet switches. As discussed in the previous chapter, we need *Route Tags* to distinguish between different segments in the network route. The Route class has a function `SetRouteTags()` which differentiates the segments and marks those switches in the route where there is a need to match and increment the route tags (start and end of segments). We illustrate an example in Figure 5.1.

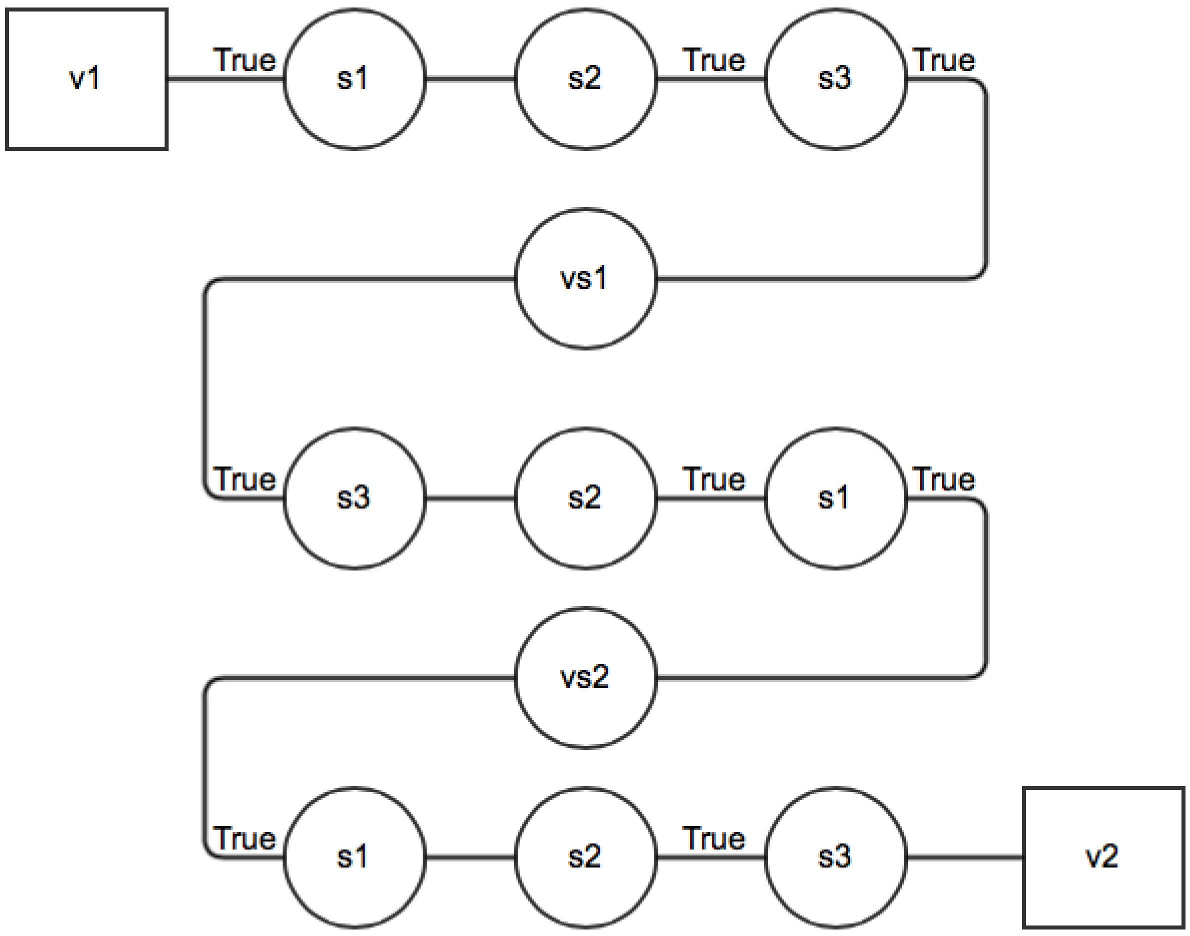

Figure 5.1: Setting Up Route Tags

The technique of finding these route tags is, in a network, the start and end of a segment will always be a *virtual entity.* Thus, the switch preceeding and succeeding the virtual entity(will be the same in our case) will need to check and increment the Route Tag. This is done by the function `SetRouteTags()`.

5.3.2 Proactive Rule Installation

When a packet reaches a switch which does not have any rule for the packet, a *PacketIn* message is sent to the POX Controller which can install the rule for the corresponding packet at the switch. However, this *reactive* rule installation will have increased latencies for the first flows. POXVine, after receiving the first packet, *proactively* installs all the network rules across all the switches (which will take some time), but any new flow will not be suffer from increased latency due to addition of OpenFlow rules by the controller. However, one problem with *proactive* rule installation is that we need to add rules for communication for all hosts to all other hosts for all tenants, which can impose resource constraints on the switch flow table sizes, thus hindering scalability. With *reactive* rule installation, you add only rules for hosts which are communicating with one other.

5.3.3 Discovery

The POX distribution comes with many useful applications. One of them is *Discovery* module, which helps detect the links in the topology. Using this, the NetworkMapper can create the *port map* for each switch(the mapping of switch port and neighbour). This is then used by to install the required rules (therefore, POXVine should not start the installation of rules before all the link discoveries have been received by the controller.) Another use of the Discovery module is that in case of link failures, the NetworkMapper must set up alternate routes to maintain connectivity.

5.3.4 Notes

- ARP packets are tricky to handle (in general, broadcast is in such a system). The virtual hosts do not need a ARP protocol for communication, however, to adhere to legacy protocols, ARP packets are flooded across the network.
- To check IP headers at the switch, we need the packet to match this.

```
msg.match.dl_type = ethernet.IP_TYPE
```

- The *VLAN ID* header field is used to store the tenant ID and RouteTag. The 12=bit VLAN ID has tenant ID as the most significant 6 bits and route tag as the least significant 6 bits.
- To send a packet back the input port (in case of loops), the controller needs to add a pecial rule and the outport should be *OFFP_IN_PORT*.

Chapter 6

Evaluation

6.1 Correctness

For correctness, we create two virtual topologies : Vt1($h1, h2, h3$) and Vt2($h4, h5, h6$). Hosts in Vt1 and Vt2 must be able to talk to each other. And, since tenants are *isolated* from each other, hosts of Vt1 should not be able to talk to Vt2 and vice-versa. The results of a `pingall` operation in Mininet gives the following results, thus verifying correctness.

```
> pingall
h1 -> h2 h3 X X X
h2 -> h1 h3 X X X
h3 -> h1 h2 X X X
h4 -> X X X h5 h6
h5 -> X X X h4 h6
h6 -> X X X h4 h5
```

6.2 Virtual Topology Mapping

We create a `TopologyCreator` class to create ring and tree topologies of custom length and depth respectively. To analyse the time complexity of mapping the virtual topology onto the physical topology, we create a tree topology of depth 10 as our physical topology and map virtual topologies of varying size and measure the time taken for the mapping. As we can observe in Figure 6.1 and 6.2, the time taken to calculate the mapping is constant across different virtual topologies(and less than 0.1 of a second). This follows from the algorithm as well, as the time complexity depends on the size of the physical topology.

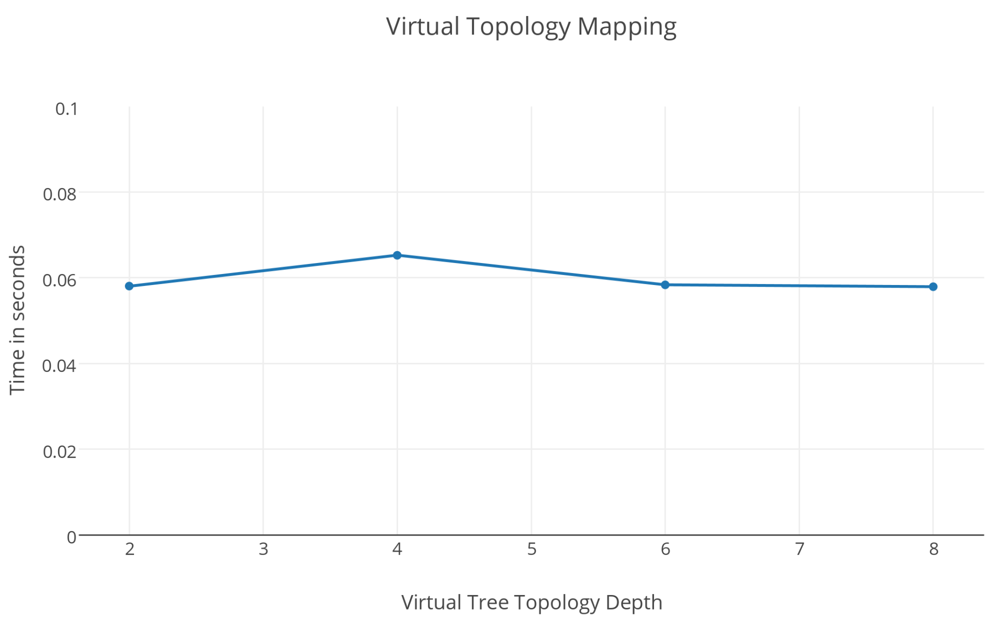

Figure 6.1: Virtual Topology Mapping for varying tree topologies

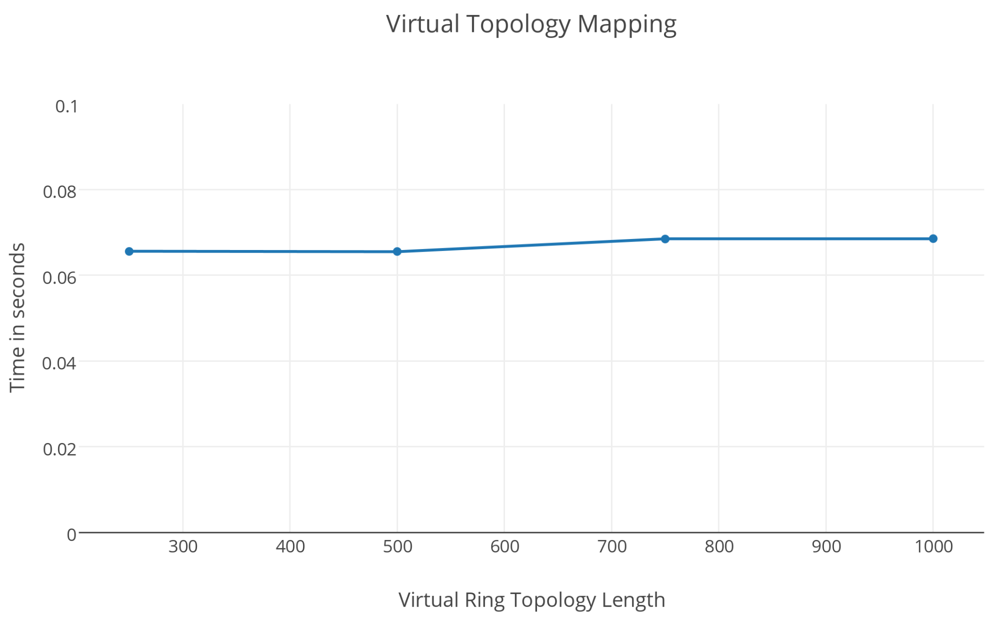

Figure 6.2: Virtual Topology Mapping for varying ring topologies

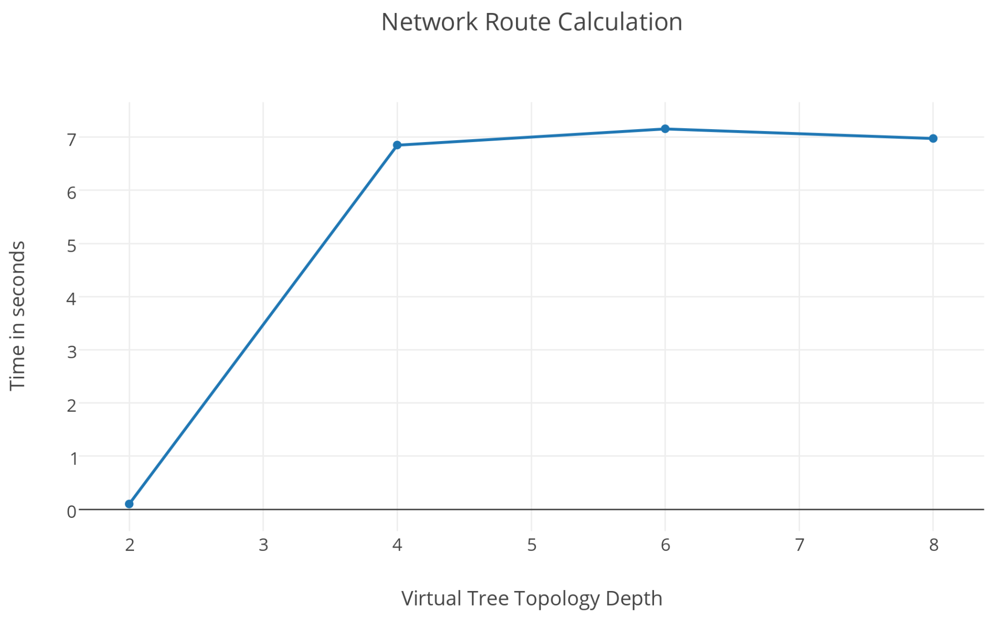

Figure 6.3: Network Route Calculation for varying tree topologies

6.3 Network Route Calculation

The next step is to calculate the *complete* network routes between every pair of hosts in the virtual topology. We can infer that this time will be dependent on the sizes of both the virtual topology and physical topologies and the mapping heuristic used. Since we try to minimise the diameter of the mapped virtual network graph, we would obtain better results than other mapping schemes. The physical topology is a tree topology of depth 10 and we vary the mapped virtual topology sizes. Figures 6.3 and 6.4 plot the time taken to calculate the network routes for the varying virtual topologies.

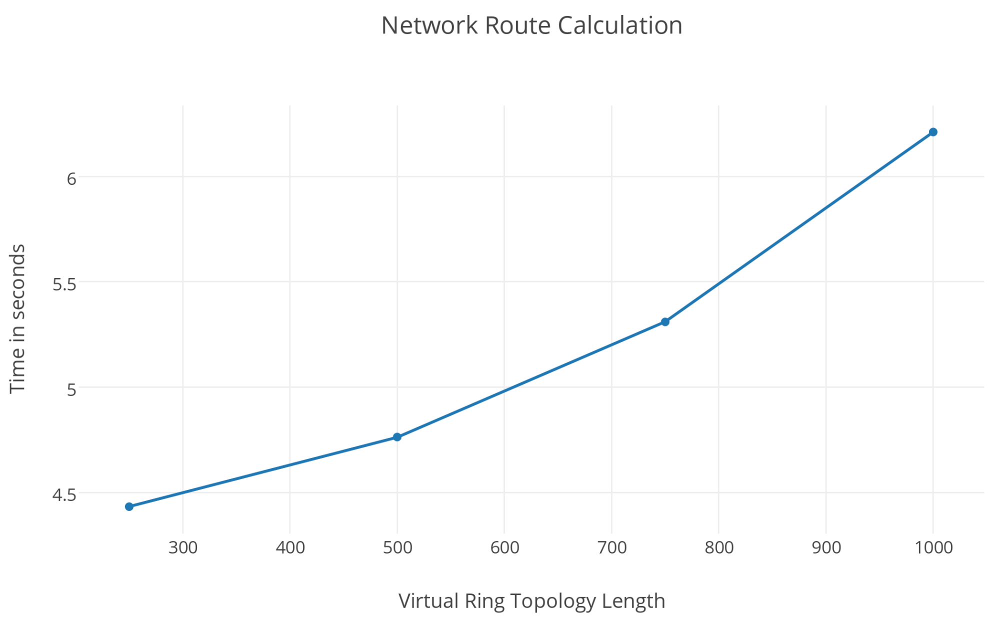

Figure 6.4: Network Route Calculation for varying ring topologies

Chapter 7

Future Work

- *Providing minimum bandwidth guarantees to tenants.* One of the most important requirement for multi-tenant cloud services is providing certain bandwidth guarantees. We can analyse different bandwidth allocation schemes, and tenants can emulate the network and estimate their performance with respect to the provided bandwidth guarantees. Providing *bandwidth guarantees*([6], [7]) is one of the hottest problems in the cloud.

- *ARP Packet Handling.* The ARP packets are not needed for communication between virtual hosts. For now, we have applied a *ad-hoc* approach of flooding packets across the network, which will lead to wasteful congestion of the network. The approach is to build a ARP handler as a POX application which sends the *ARP reply* to the *ARP responses.*

- *Handling Link and Switch Failures* In real life networks, links and switches keep on failing. Thus, the NetworkMapper must be dynamic to these failures and calculate and install new forwarding rules to maintain the *virtual network abstraction.*

- *Running actual hosts and routers for virtual networks.* Presently, we emulate the virtual network using Mininet's hosts and Open VSwitches for routers. An extension is to be able to plug in software elements (like VMs and software routers) to the Mininet backbone. This can help tenants test network configurations for correctness.

- *Network Testing as a Cloud Service.* Many enterprises want to test out new networks before deployment for correctness, resource requirements, and so on. Developing POXVine as a cloud service can be extremely useful for enterprises who need not be concerned with operations related to network testing. Also, by designing a multi-tenant emulator, multiple tenants can share POXVine's resources, thus, we need not provision resources for each tenant (the benefit of cloud computing).

Bibliography

[1] *POX.* URL: `http://www.noxrepo.org/pox/about-pox/`.

[2] *Mininet.* URL: `http://mininet.org`.

[3] Rob Sherwood et al. "Can the Production Network Be the Testbed?" In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation.* OSDI'10. Vancouver, BC, Canada: USENIX Association, 2010, pp. 1–6. URL: `http://dl.acm.org/citation.cfm?id=1924943.1924969`.

[4] Dmitry Drutskoy, Eric Keller, and Jennifer Rexford. "Scalable Network Virtualization in Software-Defined Networks". In: *IEEE Internet Computing* 17.2 (Mar. 2013), pp. 20–27. ISSN: 1089-7801. DOI: `10.1109/MIC.2012.144`. URL: `http://dx.doi.org/10.1109/MIC.2012.144`.

[5] Zafar Ayyub Qazi et al. "SIMPLE-fying Middlebox Policy Enforcement Using SDN". In: *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM.* SIGCOMM '13. Hong Kong, China: ACM, 2013, pp. 27–38. ISBN: 978-1-4503-2056-6. DOI: `10.1145/2486001.2486022`. URL: `http://doi.acm.org/10.1145/2486001.2486022`.

[6] Chuanxiong Guo et al. "SecondNet: A Data Center Network Virtualization Architecture with Bandwidth Guarantees". In: *Proceedings of the 6th International COnference.* Co-NEXT '10. Philadelphia, Pennsylvania: ACM, 2010, 15:1–15:12. ISBN: 978-1-4503-0448-1. DOI: `10.1145/1921168.1921188`. URL: `http://doi.acm.org/10.1145/1921168.1921188`.

[7] Hitesh Ballani et al. "Towards Predictable Datacenter Networks". In: *Proceedings of the ACM SIGCOMM 2011 Conference.* SIGCOMM '11. Toronto, Ontario, Canada: ACM, 2011, pp. 242–253. ISBN: 978-1-4503-0797-0. DOI: `10.1145/2018436.2018465`. URL: `http://doi.acm.org/10.1145/2018436.2018465`.