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 [**pox**] 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 [**mininet**] 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 conclusions and future work in Chapter 7.

Chapter 2

Problem Description

Chapter 3

Related Work

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*

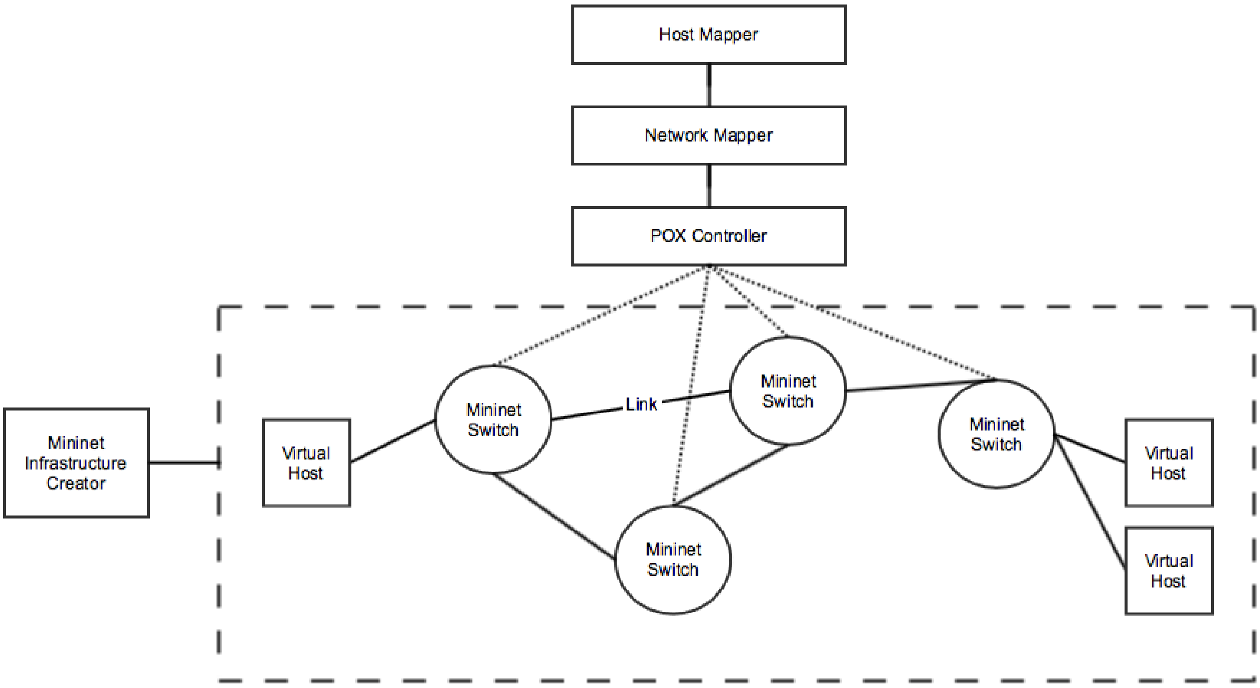

Figure 4.1: POXVine Architecture

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)

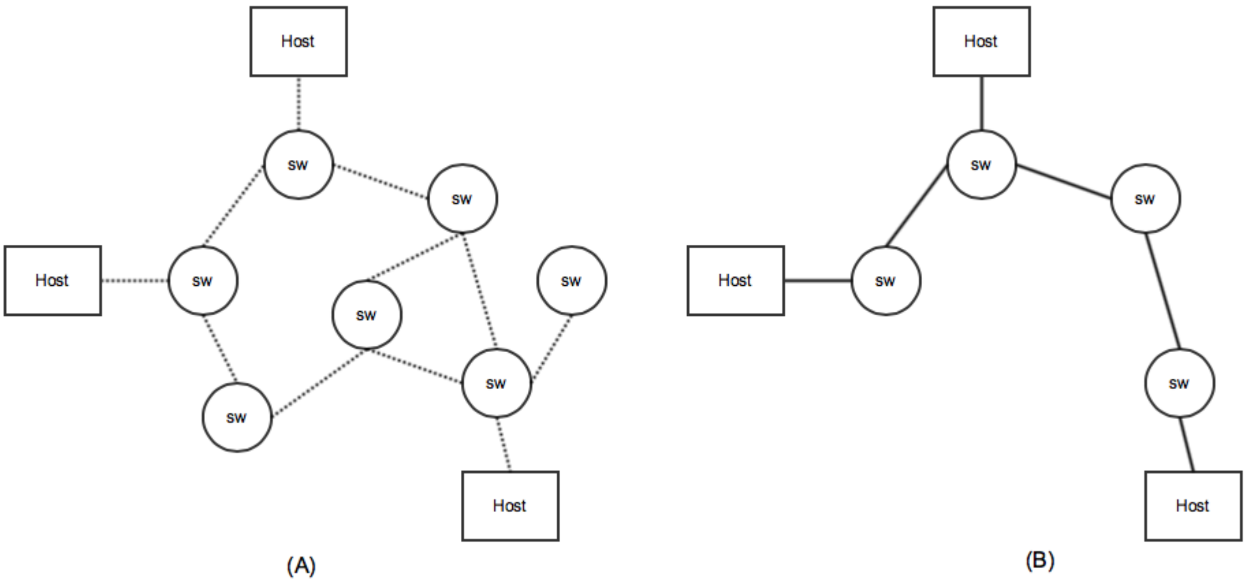

Figure 4.2: Example of the Network Graph Heuristic. (A) shows the physical network topology. Suppose if the virtual hosts are mapped to the hosts as shown in (A). The network graph consisting of shortest paths to all the hosts are shown in (B)

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 *s1* and *s2*, 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

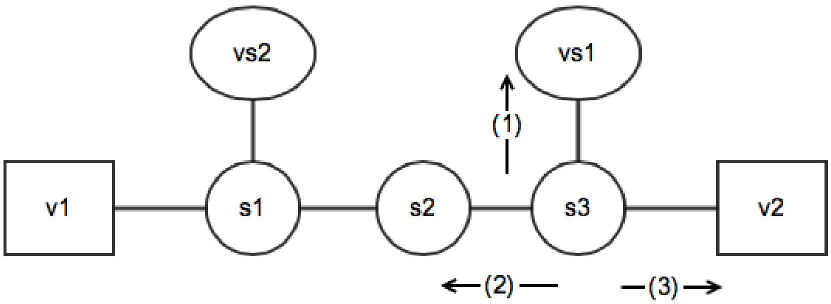

Figure 4.3: Mapping of *v1, v2, vs1 and vs2*. (1),(2) and (3) depict the three different rules a packet from $v1$ to $v2$ needs at switch $s3$.

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 [**simple**]. 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 $s3$.

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

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 [**simple**], 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.

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

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

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

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) ≠ 0 do
        sw = swlist1.pop()
        if capacity ( sw→hostlist ) ≥ 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

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).

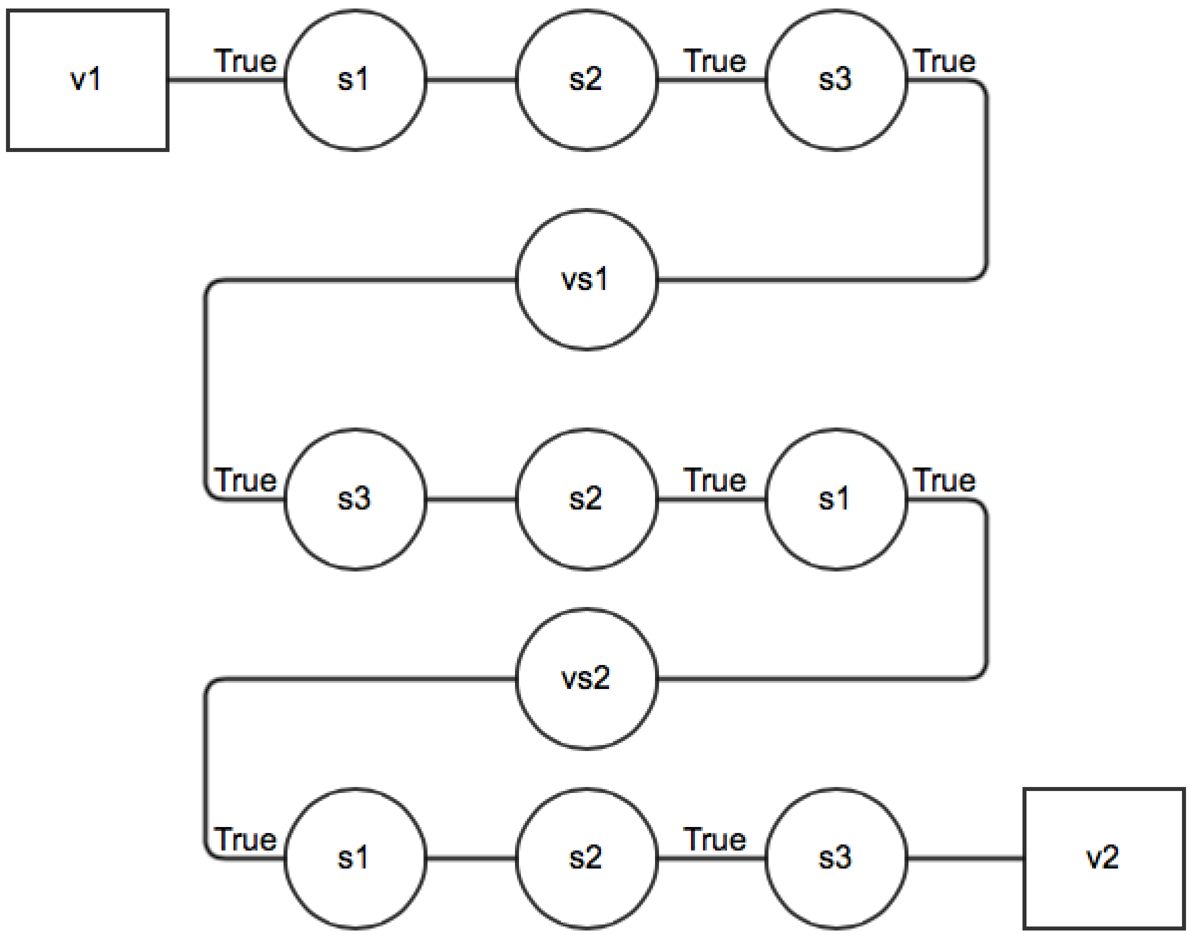

Figure 5.1: Setting Up Route Tags

Chapter 6

Conclusion and Future Work