Virtual Cluster Embeddings for SDNs

Akshay Gadre(CS12B034) R Ranjan(CS12B050)

1 Abstract

Virtual Cluster Embedding(VCE) is the problem of servicing a network tenant request requesting a virtual abstraction of nodes with link capacities. Rost et al.[1] have shown recently that the VCE problem is not NP-hard by presenting a polynomial-time algorithm to produce the flow optimal solution. We, as part of our project, have implemented the tweaked polynomial-time algorithm to service tenant requests with variable bandwidths for each server by abstracting out the network optimal solution using Floodlight controller as a functionality for the network operator. On top of that, we provide you with a "VirNet" toolkit where you can type in the physical topology and the virtual requests and the toolkit will give you the required Mininet topology with the respective customisable Floodlight controller application to implement the network optimal virtualization.

2 Introduction

Network Virtualization is the process of abstracting out a set of hardware and software network resources into a virtual network. Network Virtualization is used and implemented in various ways for Software Defined Networks(SDNs)[2]. FlowVisor[3] and OpenVirteX[4] are two famous packages supporting network virtualization for SDNs.

One of the important steps towards actually implementing virtualization is to map the virtual components on a physical topology. The concept of virtualization actually was developed so that, on the same network, multiple tenants can request a virtual network and if the resources requested by the

tenant can be allocated using prespecified schemes[5][6][7], then the resources will be allocated, otherwise, the request will be denied. Thus the scheme/algorithm for actually mapping/ abstracting out the physical infrastructure demanded is extremely critical towards optimal network virtualization.

Floodlight is open-source package including the Floodlight controller along with a set of applications built on top of the Floodlight controller. The Floodlight controller acts like a normal controller interfacing with the application layer via the north-bound interface(NBI) and and with the OpenFlow switches using the OpenFlow protocol via the south-bound or the control-data plane interface(CDPI). Floodlight uses REST(Representational State Transfer) API to interact with Java module applications to receive flow-rules and pushes them using the OpenFlow protocol to the underlying topology.

Our approach as explained in the following sections was to extract the network optimal solution for the given embedding request (in section 4), generating rules to embed the virtual topology in the physical topology (in section 5) and the architecture of "VirNet" toolkit (in section 6).

3 Concise Literature Review

Prior to Rost's work, the problem of embedding a request $\Omega(\mathcal{N}, \mathcal{B}, \mathcal{C})$ was thought to have been NP-hard. Many previous works[5][6][7] used namespace partioning and link-node infrastructure mappings to actually proposed exponential solutions for the same. Rost et al.[1] have debunked the myth by providing a polynomial-time algorithm to embed the request $\Omega(\mathcal{N}, \mathcal{B}, \mathcal{C})$ in the physical topology. Moreover, they further explained why the star topology is not the optimal topology for serving such requests and there may exist a non-star topology giving a lesser cost allocation. However, they showed that to achieve a non-star optimal embedding is not known to have a polynomial solution and shown it to be NP-hard.

4 Extracting the network solution

4.1 Assumptions

1. **Honest hosts assumption:** Every node would not send traffic at a bandwidth higher than what it requests for. We assume that the hosts

do not cheat the network by trying to extract extra bandwidth from the physical topology.

2. **Free lunch assumption:**We here assume that each host allows better performance for its working i.e. host does not bother receiving higher bandwidth than what it requested for.

4.2 Network Solution vs Flow Optimal Solution

The original algorithm proposed by Rost et al. generates a flow optimal solution for allocation of VMs in any underlying topology. It could be modified to make it a flow optimal solution for network embedding of a star topology onto any underlying physical topology (via the Min-cost Flow algorithm). But converting them to flow rules in order to maintain the request is infeasible as can be seen from the below example :

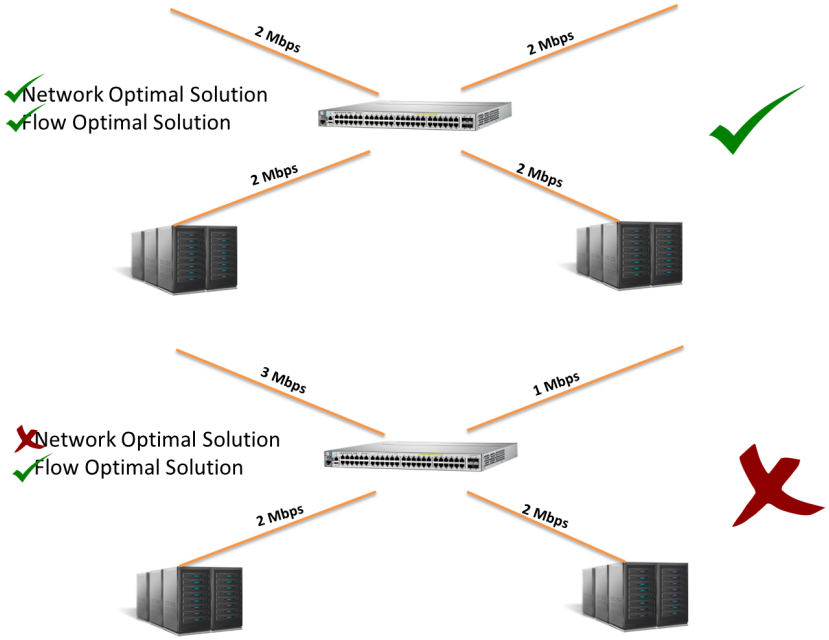

Figure 1: Network solution vs Flow Optimal solution

We instead use a shortest path algorithm based cost heuristic which tries to extract the network optimal solution. Creating flow table rules for the switches to embed a star topology is a computationally harder problem. So we resolve to a heuristic which outputs a solution only if it is feasible.

Correctness guarantee In other words, if the algorithm outputs a solution, it is guaranteed to cater to all the requests else it will suggest a solution that it can cater to in proportion of the requested constraints.

4.3 The algorithm

Algorithm 1 Network Optimal Embedding Extraction Algorithm

```
1:  procedure NEE
2:      leastCost = MAX
3:      final path = []
4:      for switch ∈ S  do
5:          paths := []
6:          failed := false
7:          for node ∈ sorted(nodes) do
8:              path = shortestPath(switch, node)
9:              if !feasible(path)  then
10:                 failed := true
11:                 break
12:             end if
13:             paths.append(path)
14:             removeAllocatedBandwidth(path)
15:             // Remove the allocated bandwidth for this particular node
16:             // for all the links along the path
17:         end for
18:         if  !failed and path.cost < leastCost then
19:             final path = path
20:             leastCost = cost
21:         end if
22:     end for
23: end procedure
```

5 Generating rules to embed the virtual topology in the physical topology

5.1 Design decisions

The first decisions to perform routing along the virtual network was to do it using source and destination ports. As we went along, we came across anomalies which forced us to force source routing till reaching the central switch and destination routing while coming out of it. So the final decision was to select the following features for installing the flow rules in Floodlight.

- **srcPort** - We use source port during the whole flow of the packet to route it.
- **destPort** - We use destination port during the whole flow of the packet to route it.
- **srcMAC** - We use the source ethernet MAC address to route the packet only till the central switch.
- **destMAC** - We use destination ethernet MAC address to route the packet from the central switch to the destination. The destination MAC address based rules are given more priority than the source-based ones.

5.2 Controller decisions

So we started this project with the plan to install the flow rules upon the virtual compilation stack of NetKAT. However due to multiple developmental issues we were unable to install the flow rules. Some of the chief problems to install the rules were:

- There was no proper installation instructions.
- Their own controller stack was incomplete. (An issue was raised from Ranjan's GitHub account which is still live)
- There were timeline constraints and better alternatives available.

After the debacle of NetKAT,we turned towards more familiar territories i.e. POX. We were able to install our flow rules and test the bandwidth

contraints. However due to some internal automatic routing mechanism, packets were being forwarded along paths restricted by our virtual embedding strategy. This, we tried to debug for about a week before shifting to a more easier to access and debug Floodlight controller.

Floodlight controller provides intuitive interfaces for our other modules to interface with and has easy debugging features including a GUI visualisation of the flow rules and the topology. This greatly favored us to choose this as the final cornerstone of the whole toolkit.

6 VirNet toolkit

6.1 The Architecture

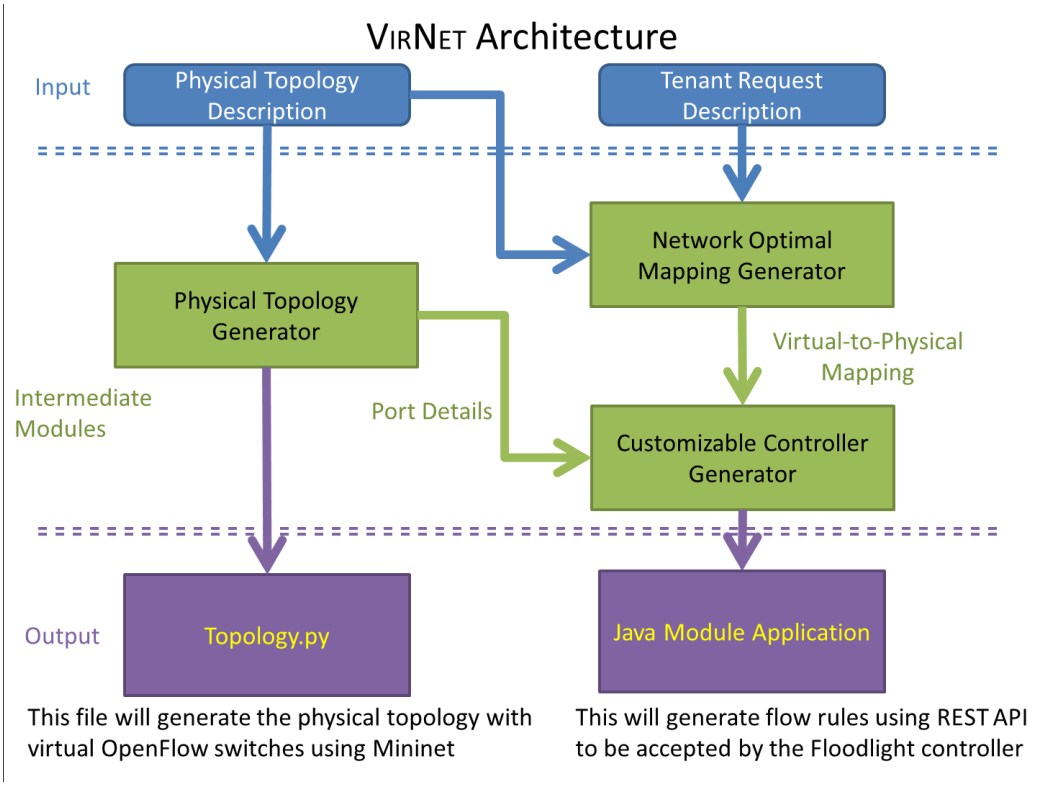

Figure 2: The VirNet Architecture

As you can see in the figure 2, the architecture takes the physical topology description and tenant request description as the input and generates the files to instantiate the physical topology using virtual OpenFlow switches using Mininet and to generate the flow rules to be used by the Floodlight controller to enforce the virtualization. The **physical topology description** contains the details about the no. of hosts, no. of switches, and the links with their parameters. Right now we have added functionality only for bandwidth for our project purposes but adding other parameters like delay, loss, max. queue size can be added easily.

A sample topology description file for a dumbbell topology will be like:

```
4 ← No. of Hosts
2 ← No. of Switches
5 ← No. of Links
s1,s2,2 ← Links in the format <Node 1,Node 2, BW>
s1,h1,1
s1,h2,1
s2,h3,1
s2,h4,1
```

The **tenant request description** contains the bandwidth requirements per host as requested by the tenant. Currently the tenant requirements are constrained to bandwidth but they can be extended to include more parameters.

A sample tenant request description file for the dumbbell topology can be like

```
4 ← No. of Hosts
h1,1 ← Tenant requirements in the format <Host 1, BW>
h2,1
h3,1
h4,1
```

We even plan to extend this to accept multiple virtualizations over the same network as well with every next request being served after the previous getting served with the remaining resources after the first one to create multiple virtualizations and serve multiple tenants or the same tenant multiple times.

The outputs of the architecture are the Mininet topology generation program and the Floodlight Controller .

The **Mininet topology generation program** is a python program which invokes Mininet's library to generate the physical topology as described

in the physical topology description.

The **Floodlight Controller** is a REST API described in the previous sections which installs OpenFlow rules as described in section 5 and provides access to the switch which the network tenant can use to control the network.

We describe the intermediate modules in the subsequent subsections.

6.2 The Physical Topology Generator

The **physical topology generator** is a Java class which receives the physical topology description, generates L2-level linkages, and outputs the port details and the physical topology generation class for python using Mininet.

One of the most important functions this module provides is the automatic allocation of ports to the links and hence generating the port details required by the customizable controller generator. It includes distinct allocation of ports for each link. This allocation has the freedom to take input the ports to be avoided so that they are not allocated.

After the allocation of the ports is completed, the physical topology generator adds links of specified bandwidths(in Mbps) between the specified nodes. After processing all the above it dispatches the constructed topology in the format required by Mininet.

6.3 The Network Optimal Mapping Generator

The **network optimal mapping generator** is a python module which receives the physical topology description, constructs the appropriate network graph and generates the virtual-to-physical mapping servicing the request generated by the tenant.

It receives the physical topology and encodes it into a NetworkX graph as per the description. Then the algorithm described above to generate the optimal virtual embedding is run on the graph to obtain the network optimal solution.

This solution is then converted into Physical-to-Virtual mapping which is sent to the customizable controller generator to be used to generate flow-rules.

6.4 The Customizable Controller Generator

The **customizable controller generator** receives the port details and the virtual-to-physical mapping from the above two modules and generates the required Java module application which acts like a REST API generating flow rules to be sent by the Floodlight controller to the respective switches.

7 Testing and Analysis

7.1 Example 1 - Dumbell Topology

The below is the physical topology:

```
4 ← No. of Hosts
2 ← No. of Switches
5 ← No. of Links
s1,s2,2 ← Links in the format <Node 1,Node 2, BW>
s1,h1,1
s1,h2,1
s2,h3,1
s2,h4,1
```

The request to VirNet is 1 Mbps for each of the hosts, namely h1, h2, h3 and h4.

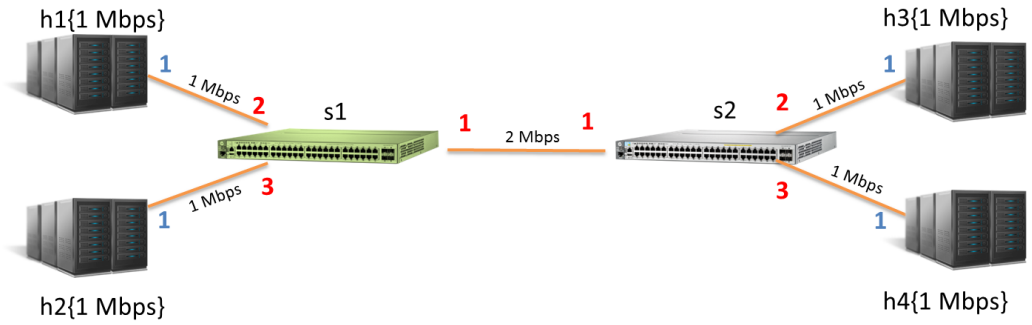

Figure 3: Example 1- Dumbell Topology

Here, the algorithm runs through all the switches i.e. s1 and s2 and finds whether it can service all the bandwidth requirements. First it tries with s1.

The order in which they are serviced is h1, h2, h3, h4 (could potentially be any order as all the hosts request for the same bandwidth).
The paths generated are as follows :

`h1` : `h1` → `s1`.
`h2` : `h2` → `s2`.
`h3` : `h3` → `s2` → `s1`.
`h4` : `h4` → `s2` → `s1`.

As the allocation is feasible in this case, the set of paths are given to the customizable controller generator which generates the following rules in the flow tables :

Cookie	Table	Priority	Match	Apply Actions	Write Actions	Clear Actions	Goto Group	Goto Meter	Write Metadata	Experimenter	Packets	Bytes	Age (s)	Timeout (s)
45035998588196899	0x0	3200	eth_dst=00:00:00:00:00:04 eth_type=0x0x800	actions:output=1	---	---	---	---	---	---	0	0	10	0
45035998588196902	0x0	3200	eth_dst=00:00:00:00:00:01 eth_type=0x0x800	actions:output=2	---	---	---	---	---	---	0	0	10	0
45035998588196903	0x0	3200	eth_dst=00:00:00:00:00:03 eth_type=0x0x800	actions:output=1	---	---	---	---	---	---	0	0	10	0
45035998588196906	0x0	3200	eth_dst=00:00:00:00:00:02 eth_type=0x0x800	actions:output=3	---	---	---	---	---	---	0	0	10	0
45035997649344784	0x0	20	eth_type=0x0x800	---	---	---	---	---	---	---	0	0	10	0

Figure 4: Example 1- Dumbell Topology S1 Flow Rules

Cookie	Table	Priority	Match	Apply Actions	Write Actions	Clear Actions	Goto Group	Goto Meter	Write Metadata	Experimenter	Packets	Bytes	Age (s)	Timeout (s)
45035998588196900	0x0	3200	in_port=1 eth_dst=00:00:00:00:00:04 eth_type=0x0x800	actions:output=3	---	---	---	---	---	---	0	0	37	0
45035998588196904	0x0	3200	in_port=1 eth_dst=00:00:00:00:00:03 eth_type=0x0x800	actions:output=2	---	---	---	---	---	---	0	0	36	0
45035998588196901	0x0	760	eth_src=00:00:00:00:00:04 eth_type=0x0x800	actions:output=1	---	---	---	---	---	---	0	0	37	0
45035998588196905	0x0	760	eth_src=00:00:00:00:00:03 eth_type=0x0x800	actions:output=1	---	---	---	---	---	---	0	0	36	0
45035997649344783	0x0	20	eth_type=0x0x800	---	---	---	---	---	---	---	0	0	37	0

Figure 5: Example 1- Dumbbell Topology S2 Flow Rules

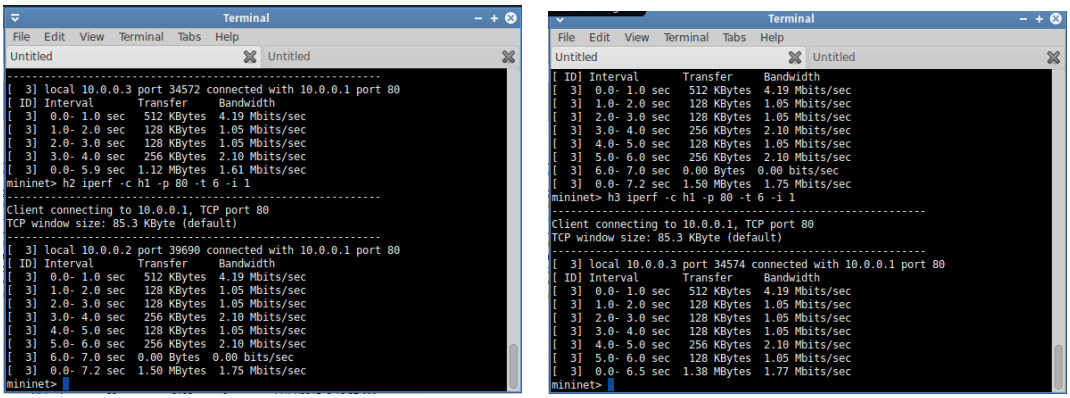

Figure 6: Example 1- Snapshots of running iperf at h1 and checking performance from h2 and h3 to get the expected bandiwdth

7.2 Example 2 - Distributed Topology

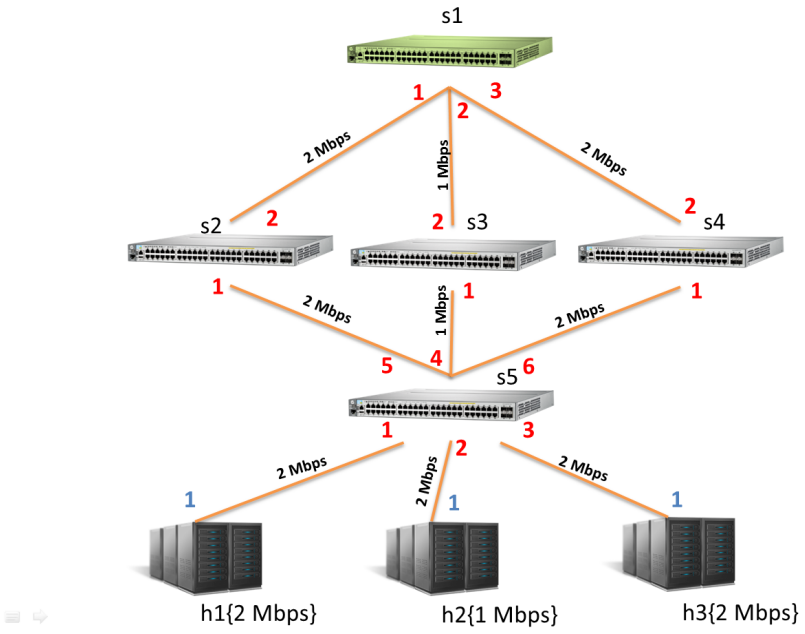

Figure 7: Example 2- Distributed Topology

In this topology request, we are asking for heterogeneous bandwidths.We first try to service the request having s1 as the central switch.

The paths generated are :

`h1` : `h1` $\rightarrow$ `s5` $\rightarrow$ `s2` $\rightarrow$ `s1`.
`h2` : `h2` $\rightarrow$ `s5` $\rightarrow$ `s3` $\rightarrow$ `s1`.
`h3` : `h3` $\rightarrow$ `s5` $\rightarrow$ `s4` $\rightarrow$ `s1`.

The allocation is feasible and the rules generated are as follows :

Cookie	Table	Priority	Match	Apply Actions	Write Actions	Clear Actions	Goto Group	Goto Meter	Write Metadata	Experimenter	Packets	Bytes	Age (s)	Timeout (s)
45035998588196902	0x0	3200	eth_dst=00:00:00:00:00:01 eth_type=0x0x800	actions:output=1	---	---	---	---	---	---	0	0	11	0
45035999300199478	0x0	3200	eth_dst=00:00:00:00:00:03 eth_type=0x0x800	actions:output=3	---	---	---	---	---	---	0	0	11	0
45035999300199483	0x0	3200	eth_dst=00:00:00:00:00:02 eth_type=0x0x800	actions:output=2	---	---	---	---	---	---	0	0	11	0
45035997649344783	0x0	20	eth_type=0x0x800	---	---	---	---	---	---	---	0	0	11	0
0	0x0	0		actions:output=controller	---	---	---	---	---	---	38	2989	19	0

Figure 8: Example 2- Distributed Topology S1 Flow Rules

Cookie	Table	Priority	Match	Apply Actions	Write Actions	Clear Actions	Goto Group	Goto Meter	Write Metadata	Experimenter	Packets	Bytes	Age (s)	Timeout (s)
45035998588196903	0x0	3200	in_port=2 eth_dst=00:00:00:00:00:01 eth_type=0x0x800	actions:output=1	---	---	---	---	---	---	0	0	75	0
45035998588196906	0x0	760	in_port=1 eth_src=00:00:00:00:00:01 eth_type=0x0x800	actions:output=2	---	---	---	---	---	---	0	0	75	0
45035997649344787	0x0	20	eth_type=0x0x800	---	---	---	---	---	---	---	0	0	75	0

Figure 9: Example 2- Distributed Topology S2 Flow Rules

Cookie	Table	Priority	Match	Apply Actions	Write Actions	Clear Actions	Goto Group	Goto Meter	Write Metadata	Experimenter	Packets	Bytes	Age (s)	Timeout (s)
45035999300199484	0x0	3200	in_port=2 eth_dst=00:00:00:00:00:02 eth_type=0x0x800	actions:output=1	---	---	---	---	---	---	0	0	35	0
45035999300199487	0x0	760	in_port=1 eth_src=00:00:00:00:00:02 eth_type=0x0x800	actions:output=2	---	---	---	---	---	---	0	0	35	0
45035997649344786	0x0	20	eth_type=0x0x800	---	---	---	---	---	---	---	0	0	36	0

Figure 10: Example 2- Distributed Topology S3 Flow Rules

Cookie	Table	Priority	Match	Apply Actions	Write Actions	Clear Actions	Goto Group	Goto Meter	Write Metadata	Experimenter	Packets	Bytes	Age (s)	Timeout (s)
45035999300199479	0x0	3200	in_port=2 eth_dst=00:00:00:00:00:03 eth_type=0x0x800	actions:output=1	---	---	---	---	---	---	0	0	91	0
45035999300199482	0x0	760	in_port=1 eth_src=00:00:00:00:00:03 eth_type=0x0x800	actions:output=2	---	---	---	---	---	---	0	0	91	0
45035997649344785	0x0	20	eth_type=0x0x800	---	---	---	---	---	---	---	0	0	92	0

Figure 11: Example 2- Distributed Topology S4 Flow Rules

Cookie	Table	Priority	Match	Apply Actions	Write Actions	Clear Actions	Goto Group	Goto Meter	Write Metadata	Experimenter	Packets	Bytes	Age (s)	Timeout (s)
45035998588196904	0x0	3200	in_port=5 eth_dst=00:00:00:00:00:01 eth_type=0x0x800	actions:output=1	---	---	---	---	---	---	0	0	7	0
45035999300199480	0x0	3200	in_port=6 eth_dst=00:00:00:00:00:03 eth_type=0x0x800	actions:output=3	---	---	---	---	---	---	0	0	7	0
45035999300199485	0x0	3200	in_port=4 eth_dst=00:00:00:00:00:02 eth_type=0x0x800	actions:output=2	---	---	---	---	---	---	0	0	7	0
45035998588196905	0x0	760	eth_src=00:00:00:00:00:01 eth_type=0x0x800	actions:output=5	---	---	---	---	---	---	0	0	7	0
45035999300199481	0x0	760	eth_src=00:00:00:00:00:03 eth_type=0x0x800	actions:output=6	---	---	---	---	---	---	0	0	7	0
45035999300199486	0x0	760	eth_src=00:00:00:00:00:02 eth_type=0x0x800	actions:output=4	---	---	---	---	---	---	0	0	7	0
45035997649344784	0x0	20	eth_type=0x0x800	---	---	---	---	---	---	---	0	0	8	0

Figure 12: Example 2- Distributed Topology S5 Flow Rules

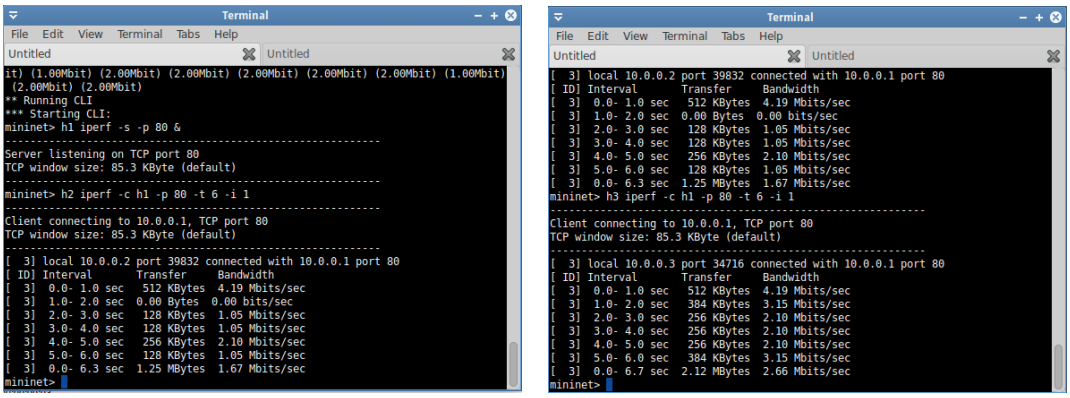

Figure 13: Example 2- Snapshots of running iperf at h1 and checking performance from h2 and h3 to get the expected bandwidth

8 Future Work

This is a dynamic project which means multiple new modules can be added with new exciting properties without changing the "VirNet" architecture. Some of the future ideas can be:

1. **Sequential Virtualization:** We can add the feature to support servicing multiple tenant requests sequentially over the same physical topology by changing the network optimal mapping generator and adding more parameters to guide the flow like VLAN Id.

2. **Algorithm Testbed:** This can act like a algorithm testing toolkit where the required details regarding the physical topology are passed by the physical topology generator to the algorithm module which will feed the rules as a REST API to the Floodlight controller.

3. **Supporting a variety of input format:** We can develop different physical topology generators and algorithm-dependent mapping generators to support different input formats of the physical topology description and the input to that algorithm.

4. **Supporting a variety of controllers:** We can develop multiple customizable controller generators each producing the controller module of respective controller like POX, Pyretic, etc.

9 Acknowledgements

We would like to thank Prof Marco Canini of UCL Italy for always helping us with all the doubts we had related to Frenetic VM and guiding us when all else had failed. We would also like to thank the Frenetic team who have been vociferous and enthusiastic in response to all our queries and bugs.

References

[1] Matthias Rost, Carlo Fuerst, and Stefan Schmid, "Beyond the stars: Revisiting virtual cluster embeddings", *SIGCOMM CCR*, July 2015.

[2] N. McKeown, Software defined networking, in *OFC 2013* Plenary

[3] Sherwood, R., Gibb, G., Yap, K.-K., Appenzeller, G., Casado, M., McKeown, N., and Parulkar, G, "FlowVisor: A Network Virtualization Layer", Tech. Rep. *OPENFLOW-TR-2009-1, OpenFlow*, October 2009.

[4] A. Al-Shabibi, M. De Leenheer, M. Gerola, A. Koshibe, G. Parulkar, E. Salvadori, and B. Snow, OpenVirteX: Make your virtual SDNs programmable in *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking, ser. HotSDN 14*, New York, NY, USA: ACM, 2014, pp. 2530.

[5] Yufeng Xin , Ilia Baldine , Anirban Mandal , Chris Heermann , Jeff Chase , Aydan Yumerefendi, "Embedding virtual topologies in networked clouds", *Proceedings of the 6th International Conference on Future Internet Technologies*, June 13-15, 2011, Seoul, Republic of Korea [doi¿10.1145/2002396.2002403]

[6] Mosharaf Chowdhury , Muntasir Raihan Rahman , Raouf Boutaba, "ViNEYard: virtual network embedding algorithms with coordinated node and link mapping", *IEEE/ACM Transactions on Networking (TON)*, v.20 n.1, p.206-219, February 2012 [doi¿10.1109/TNET.2011.2159308]

[7] Chrysa Papagianni , Aris Leivadeas , Symeon Papavassiliou , Vasilis Maglaris , Cristina Cervello-Pastor , Alvaro Monje, "On the Optimal Allocation of Virtual Resources in Cloud Computing Networks", *IEEE Transactions on Computers*, v.62 n.6, p.1060-1071, June 2013 [doi¿10.1109/TC.2013.31]