GENI-VIRO TESTING AND EVALUATION REPORT

Department of Computer Science and Engineering
University of Minnesota
4-192 EECS Building
200 Union Street SE
Minneapolis, MN 55455-0159 USA

Implementing & Deploying a Non-IP Routing Protocol VIRO in GENI

Braulio Dumba, Hesham Mekky, Guobao Sun, Zhi-Li Zhang

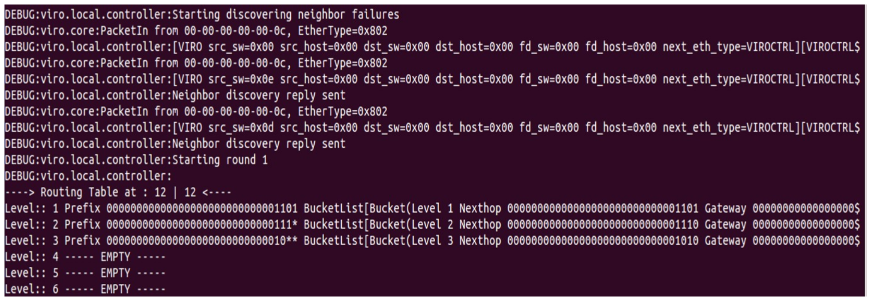
May 2015

Implementing & Deploying a Non-IP Routing Protocol VIRO in GENI

1 INTRODUCTION

We have conducted a number of experiments to test our initial prototype of VIRO using both Mininet and GENI. In this report we describe some of our experiments. First, we investigate VIRO's packet encapsulation/decapsulation overhead at edges switches, and we evaluate and compare VIRO's failure recovery mechanism as discussed in [5]: *Neighbor Echo Request & Reply* and *Port Status*. In addition, we investigate the possibility of using GENI stitching to forwards VIRO frames. Second, we show how the VIRO routing protocol localizes failures (single and multiple link failures), supports fast re-routing and host/users mobility. Then, we present a number of experiments evaluating our novel in-network pathlet switching framework with VIRO for SDN networks [5] and also evaluating our adaptive resilient routing via Preorders in SDN[8]. We conclude by discussing our lessons learned using the GENI test-bed.

In order to set up and run our VIRO's experiments in GENI, we deploy our extend-OVS and VIRO POX controllers (local and remote)[5] to GENI using the following steps. First, we create a GENI node in our slice. Next, we download and install our extend OVS and POX controllers to our GENI node. Then, we create an InstaGENI custom image of our node using Flack [1]. We later use this custom image at each GENI node in our experiments, because it has all the features and applications that we use in our experiments. We use Flack and Jack to reserve the resources in most of our experiments. In the next sections, we discuss our experiments in details.

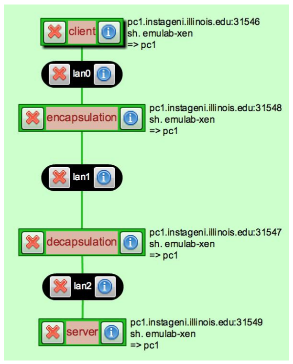


```
DEBUG:viro.local.controller:Starting discovering neighbor failures
DEBUG:viro.core:PacketIn from 00-00-00-00-00-0c, EtherType=0x802
DEBUG:viro.local.controller:[VIRO src_sw=0x00 src_host=0x00 dst_sw=0x00 dst_host=0x00 fd_sw=0x00 fd_host=0x00 next_eth_type=VIROCTRL][VIROCTRL$
DEBUG:viro.core:PacketIn from 00-00-00-00-00-0c, EtherType=0x802
DEBUG:viro.local.controller:[VIRO src_sw=0x0e src_host=0x00 dst_sw=0x00 dst_host=0x00 fd_sw=0x00 fd_host=0x00 next_eth_type=VIROCTRL][VIROCTRL$
DEBUG:viro.local.controller:Neighbor discovery reply sent
DEBUG:viro.core:PacketIn from 00-00-00-00-00-0c, EtherType=0x802
DEBUG:viro.local.controller:[VIRO src_sw=0x0d src_host=0x00 dst_sw=0x00 dst_host=0x00 fd_sw=0x00 fd_host=0x00 next_eth_type=VIROCTRL][VIROCTRL$
DEBUG:viro.local.controller:Neighbor discovery reply sent
DEBUG:viro.local.controller:Starting round 1
DEBUG:viro.local.controller:
----> Routing Table at : 12 | 12 <----
Level:: 1 Prefix 00000000000000000000000000001101 BucketList[Bucket(Level 1 Nexthop 00000000000000000000000000001101 Gateway 0000000000000000$
Level:: 2 Prefix 0000000000000000000000000000111* BucketList[Bucket(Level 2 Nexthop 00000000000000000000000000001110 Gateway 0000000000000000$
Level:: 3 Prefix 000000000000000000000000000010** BucketList[Bucket(Level 3 Nexthop 00000000000000000000000000001010 Gateway 0000000000000000$
Level:: 4 ----- EMPTY -----
Level:: 5 ----- EMPTY -----
Level:: 6 ----- EMPTY -----
```

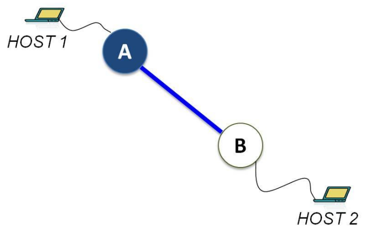
Figure 1: Screen-shot of a VIRO routing table

2 BASIC EXPERIMENTS

We have conducted a number of experiments to test our initial prototype of VIRO using both Mininet and GENI. In this section, we describe three sets of experiments. In the first experiment, we investigate VIRO's packet encapsulation/decapsulation overhead at edges switches. In the second experiment, we evaluate and compare VIRO's failure recovery mechanism as discussed in [5] (*Neighbor Echo Request & Reply* and *Port Status*). Lastly, we investigate the possiblity of forwarding VIRO frames across different GENI Aggregate Managers (AM) using "GENI Stitching". The results of these experiments will help us to improve our prototype of VIRO in GENI, for example: to select the best failure recovery mechanism.

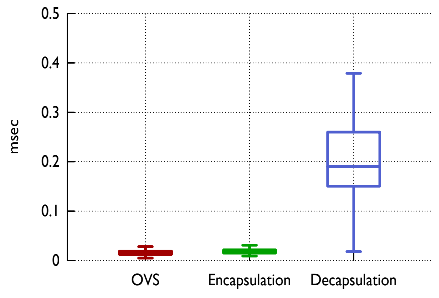


(a) Topology in GENI

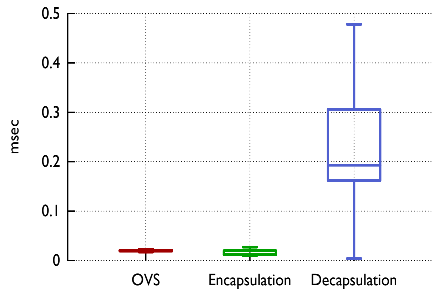


(b) Network topology

Figure 2: VIRO packet processing overhead experimental setup



(a) GENI



(b) Mininet

Figure 3: Packet processing delay

2.1 VIRO Packet Processing Overhead

Experiment Setup: in this experiment, we are interested in the answer to the following question: what is the processing delay overhead imposed by VIRO's packets encapsulation/decapsulation at the edges switches? To achieve this, we create a simple topology with two hosts (h1 and h2), connected by two switches (see Figure 2). In order to isolate and measure the processing delay of individual packets, we use *tcpdump* to obtains the timestamps of packets as they enter and leave a switch. The difference in the timestamps is the delay. The traffic that is sent for the delay measurement is a stream of ping messages[2] from host h1 to host h2. We repeat this experiment using both a traditional OVS (with the standard IP forwarding) and our extended OVS.

Experiment Discussion: h1 pings h2 and we measure the time that it takes for each echo request message to reach the interfaces for both switches. We compute the difference as the "packet processing delay time" - since we do not generate high amounts of traffic, we consider the queue delay negligible. We repeat this experiment both in Mininet and GENI, and the results are shown in Figures 3(a) and 3(b). The plots show the processing time in milliseconds for a traditional OVS, extended-OVS encapsulation (encap-OVS) and extended-OVS decapsulation (decap-OVS). We observe from our simulation results in Mininet that the 95 percentile for packet's processing delay is 2.30×10^{-2}, 2.20×10^{-2} and 3.82×10^{-1} milliseconds for OVS, encap-OVS and decap-OVS. Our experimental results from GENI are similar: 3.11×10^{-2}, 3.01×10^{-2} and 3.50×10^{-1} milliseconds for OVS, encap-OVS and decap-OVS. Our results show that the packet's processing delay for OVS and encap-OVS are very close. However, there is an increase in the packet's processing time for decap-OVS (see Figures 3(a) and 3(b)). In the future, we will investigate why the processing time for VIRO packet decapsulation is significantly larger than packet encapsulation.

[1]Flack is a flash-based Web interface for viewing and requesting GENI resources. It also provides tools to manage the resources.
[2]We generate 100 ping request packets

2.2 VIRO Failure Recovery

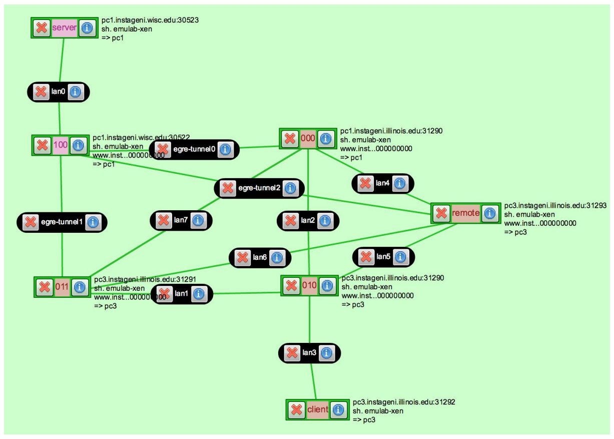
Experiment Setup: in this experiment, we are particularly interested in investigating VIRO's failure recovery mechanisms: *Echo Request & Reply* and *Port Status*. To achieve this, we use the network topology illustrated in Figure 4(b). We attach a client machine to node B and a server to node D. The network tool *iperf* is used to generate traffic from the client to the server for 150 seconds. During this process, we fail the link C-D and measure the time it takes for the network to recovery[3]. We repeat this experiment both in Mininet and GENI.

Figure 4(a) shows the deployment of our experiment in GENI. We use 3 PCs, 7 XenVMs and two GENI Aggregate Managers (AMs): Wisconsin and Illinois. The nodes at the same level in VIRO are placed in the same GENI PC, whereas nodes at different level are at distinct GENI PCs. Hence, from Figure 4(a), nodes 010 and 011 are in the same PC. To connect the nodes at different GENI AMs we use EGRE tunnels.

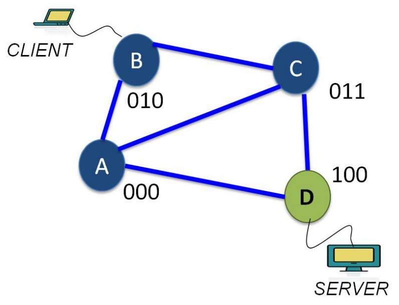
Experiment Discussion: using VIRO's routing protocol, the client at node B sends data packets to the server at node D. Before failure, node B uses its level-3 GW (node C) to communicate with the server. After failure of the link C-D, node C updates its routing table and sends a *GW_Withdraw* message to its level-3 rendezvous point (rdv) - node A. Node A updates its rdv store and sends a *GW_Remove* message to node B. Then, node B updates its routing table and queries its level-3 rdv (Node A) for a new level-3 GW. Node A returns itself as the new level-3 GW for node B.

From Figure 5(b) we observe that the failure happens at 23 seconds. We also observe that it takes 5 seconds for the network to recover using the ports status event mechanism. Whereas for the echo-messages mechanism it takes 57 seconds. Similarly, our experimental results from GENI shows similar trend, see Figure 5(a). The failure occurs at about 20 seconds, and it takes 12 seconds for the network to recover using the port status method. However, it takes about 54 seconds for the network to recover using echo-messages. From these results we observe that the *Port status* method outperforms *Neighbor Echo Request & Reply* method, as expected.

The recovery time for both experiments in GENI and Mininet is significantly large. Hence, in the future we will improve the tuning of our experimental parameters in order to decrease the failure recovery time in the network.



(a) Topology in GENI



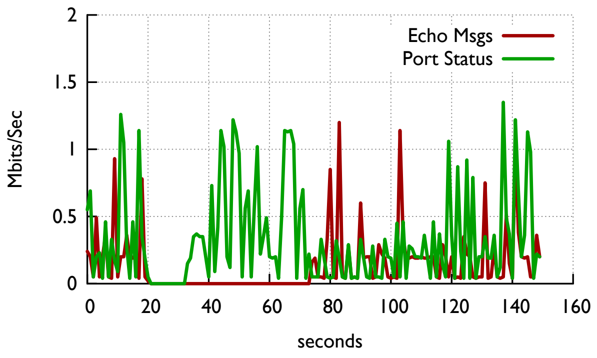
(b) Network Topology

Figure 4: VIRO failure recovery experiment setup

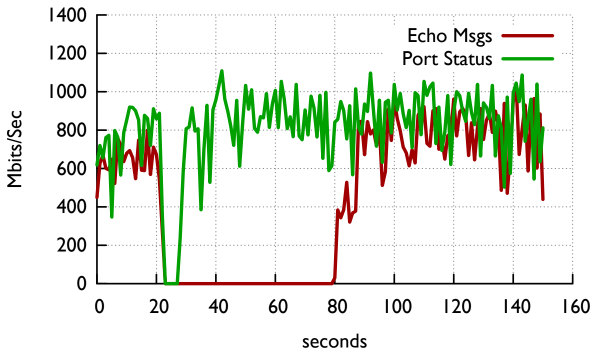
2.3 GENI Stiching

Experiment Setup: in this experiment, we are interested in the answer to the following question: can we route VIRO packets in GENI using stitching? To achieve this, we create a simple topology with two switches (sw1 and sw2) connected by a link (see 19(b)). We deployed this topology in GENI (see 19(a)) in two different AMs – Wisconsin(WI) and Illinois(IL).[4]. To connect the switches at the different AMs, we use a stitching link. We transmit VIRO frames [5] from sw1 to sw2, and used the MAC addresses of both switches to set the SVID and

[3]While the client at B is sending traffic to the server at D
[4] sw1 is at WI and sw2 is at IL
[5]We attach a POX controller at sw1 and use it to generate VIRO frames



(a) GENI

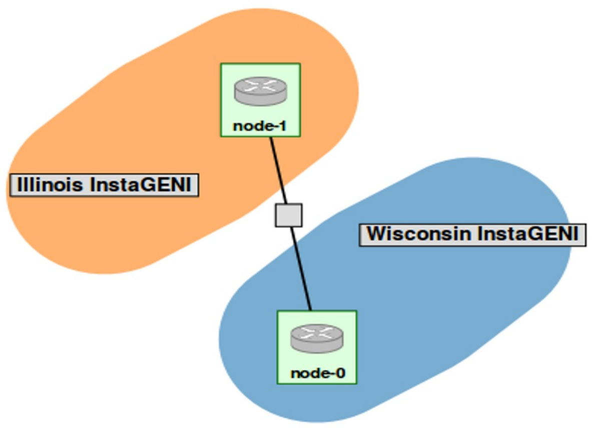


(b) Mininet

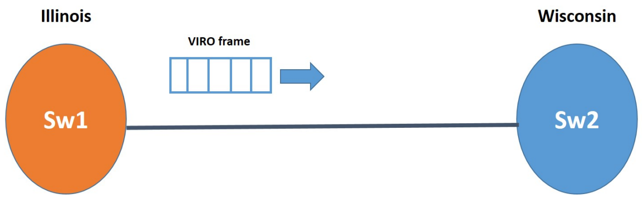
Figure 5: Failure Recovery

DVID in the frame. We use *tcpdump* to observe the packets as they arrive at sw2.

Experiment Discussion: using the experimental set-up described above, we observe the following: VIRO packets with the SVID address equal to sw1's MAC address, always arrive at the destination sw2. However, when we set the SVID with a crafted MAC address, the transmitted packet never arrived at the destination switch. Therefore, from these results, we can infer that the forwarding in stitching is done based on known MAC addresses only. Thus, we are able to forward our VIRO frame, but we cannot forward it using valid vids.



(a) GENI

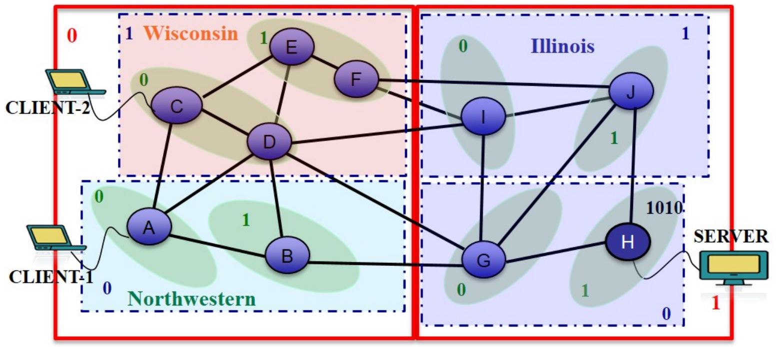


(b) Network Topolgy

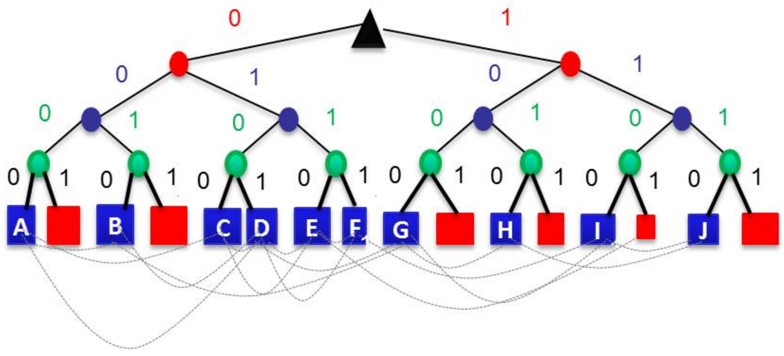
Figure 6: GENI stiching experiment with VIRO frames

3 LARGER EXPERIMENTS

In this section, we describe 5 sets of experiments showing how VIRO routing protocol localizes failures, supports fast re-routing and host/users mobility. In the first three sets of experiments, we show how a VIRO network recovers from a single link and multiple links failures. In the last two sets of experiments, we show how VIRO offers support for hosts/users mobility.

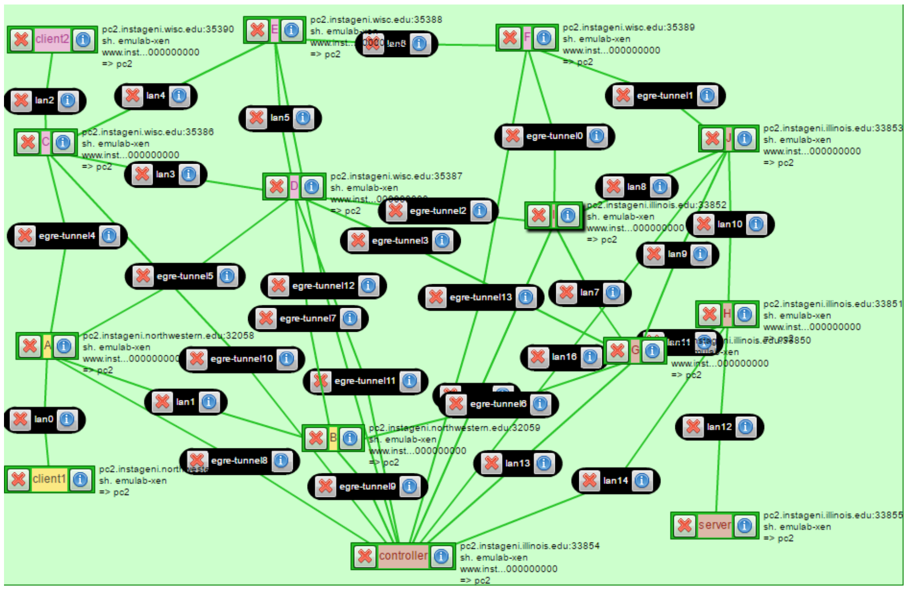


(a) Network Topology



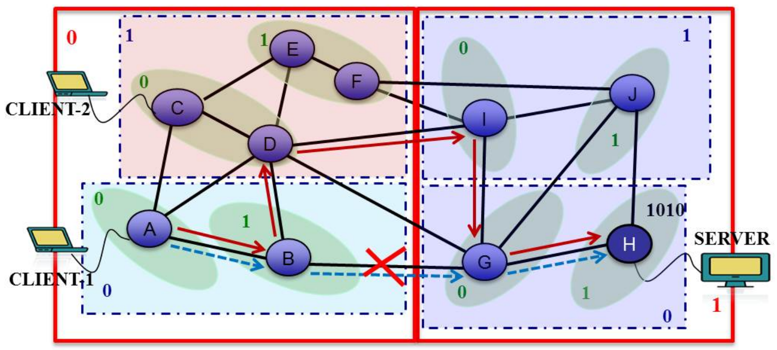
(b) Network topology represented as virtual binary tree: the grey dotted lines denote physical connectivity and the red boxes represent the unused vid's

Figure 7: VIRO failure recovery network topology

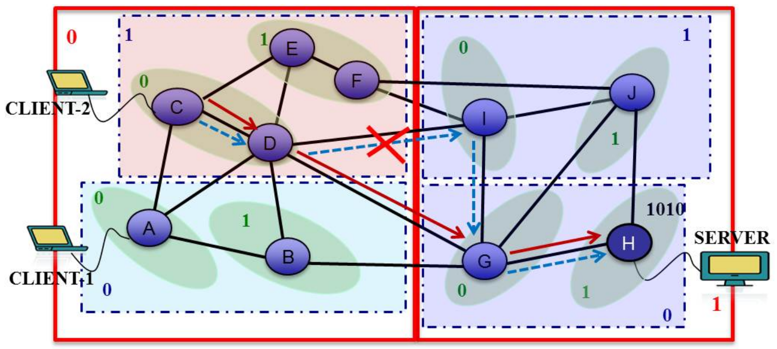


client2
client1
server
controller
egre-tunnel0
egre-tunnel1
egre-tunnel2
egre-tunnel3
egre-tunnel4
egre-tunnel5
egre-tunnel6
egre-tunnel7
egre-tunnel8
egre-tunnel9
egre-tunnel10
egre-tunnel11
egre-tunnel12
egre-tunnel13
lan0
lan1
lan2
lan3
lan4
lan5
lan7
lan8
lan9
lan10
lan12
lan13
lan14
lan16
pc2.instageni.wisc.edu:35390
pc2.instageni.wisc.edu:35388
pc2.instageni.wisc.edu:35389
pc2.instageni.wisc.edu:35386
pc2.instageni.wisc.edu:35387
pc2.instageni.illinois.edu:33853
pc2.instageni.illinois.edu:33852
pc2.instageni.illinois.edu:33851
pc2.instageni.illinois.edu:33855
pc2.instageni.illinois.edu:33854
pc2.instageni.northwestern.edu:32058
pc2.instageni.northwestern.edu:32059
sh. emulab-xen
=> pc2

Figure 8: VIRO failure recovery network topology in GENI



(a) Client-1 Failure Recovery



(b) Client-2 Failure Recovery

Figure 9: VIRO failure recovery (single link failure): the *blue lines* represent the network path before failure and *brown lines* represent the network path after failure

3.1 VIRO Failure Recovery: single link failure (3 GENI Aggregate Managers)

Experiment Setup: in this experiment, we are particularly interested in showing how VIRO handles single network failures. To achieve this, we use the network topology illustrated in Figure 7(a). We attach two client machines to nodes A (client-1) and C(client-2) and a server to node H. The network tool *iperf* is used to generate traffic from the client to the server. To illustrates how VIRO localizes failures, we first fail the link B-G and measure the client-1 throughput. Similarly, we fail the link D-I and measure the client-2 throughput (see Figure9).

Figure 7(a) shows our network topology: 10 VIRO switches, 21 links, 2 clients and 1 server machines. Figure 8 shows our network topology in GENI. We use 14 XenVMs and three GENI Aggregate Managers (AMs): Northwestern (A, B, client-1), Wisconsin (C, D, E, F, client-2) and Illinois (I, J, G, H, server). To connect the nodes at different GENI AMs we use EGRE tunnels.

Experiment Discussion: using VIRO's routing protocol, the client at node A sends data packets to the server at node H using the following path: $A \rightarrow B \rightarrow G \rightarrow H$. However, the client at node C uses the following path to communicate with the server: $C \rightarrow D \rightarrow I \rightarrow G \rightarrow H$. When the link B-D fails, node A updates its level-4 gateway to node D and uses the following new path to communicate to the server: $A \rightarrow B \rightarrow D \rightarrow I \rightarrow G \rightarrow H$. After the failure of link B-D only the nodes A, B, D update its routing tables (RT), and the RT of the remaining nodes in the network is unchanged. Therefore, client-2's path to communicate with the server is unchanged. On the other hand, if link D-I fails, only the nodes D, E, I update their RTs. The new path for client-2 to communicate with the server will be the following: $C \rightarrow D \rightarrow G \rightarrow G \rightarrow H$. These experimental results show that VIRO localizes failures and only the nodes in the neighbourhood of the failed link/node need to update their routing table (see Figure 9).

From Figure 10 we observe that the failure of link B-D happens at 23 seconds. We also observe that it takes 14 seconds for the network to recover. Similarly, our experimental results for client-2 shows similar trend, see Figure 10. The failure occurs at about 41 seconds, and it takes 27 seconds for the network to recover. The recovery time for both experiment is significantly large. Hence, in the future we will improve the tuning of our experimental parameters in order to decrease the failure recovery time in the network.

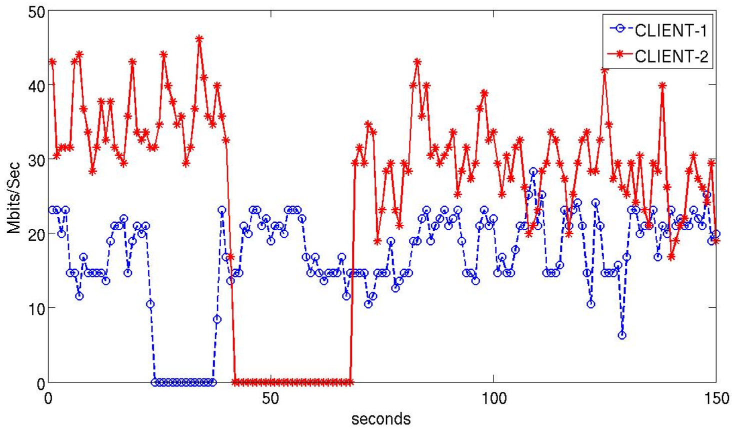


Figure 10: VIRO links failures recovery: Single link failure

3.2 VIRO Failure Recovery: single link failure (8 GENI Aggregate Managers)

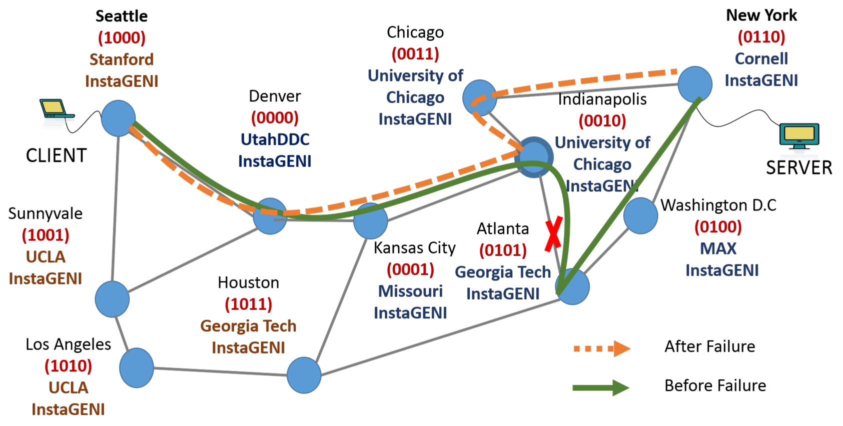


Figure 11: GENI Experiment Topology

Experiment Setup: in this experiment, we are particularly interested in investigating VIRO's failure recovery mechanisms: *Echo Request & Reply* and *Port Status* [7]. To achieve this, we use the network topology illustrated in Figure 11. We deployed this topology in GENI using 8 GENI InstaGENI Aggregate Managers (AMs), 10 raw PCs, 14 Xen VMs and EGRE tunnels to connect the GENI AMs. For this experiment, a client in Seattle communicates with a server in New York.

Experiment Discussion: based on VIRO'svrouting tables the client's path to communicate with the server is the following: $Seattle \rightarrow Denver \rightarrow KansasCity \rightarrow Indianapolis \rightarrow Atlanta \rightarrow D.C. \rightarrow NewYork$ (see Figure 11). During this process, we fail the link $Indianapolis \rightarrow Atlanta$ and measure the time it takes for the network to recover. Before failure, node 0010 is used as the level-3 gateway to reach the server in New York. However, after failure, node 0010 updates its routing table and sends a *GW_Withdraw* message to its level-3 rendezvous point (rdv), $rdv_3(0010) = 0000$. Thus, the rdv updates its rdv store and sends *GW_Remove* messages to all the nodes using node 0010 as their level-3 gateway. Consequently, node 0010 queries node 0000 for a new level-3 gateway. Then, node 0000 returns node 0011 as the new level-3 gateway and the new path for the packets from the client to the server will be the following: $Seattle \rightarrow Denver \rightarrow KansasCity \rightarrow Indianapolis \rightarrow Chicago \rightarrow NewYork$.

We use the network tool *iperf* to generate traffic from the client to the server for 150 seconds. Figure 12 shows the results of our experiment. We observe that the failure of link $Indianapolis \rightarrow Atlanta$ happens at about 20 seconds. It takes 20 seconds for the network to recover using the port status method (see Figure 12). However, it takes about 60 seconds for the network to recover using VIRO echo-messages. These results shows that the *Port status* method outperforms *Neighbor Echo Request & Reply* method, as expected. We also observe that recovery

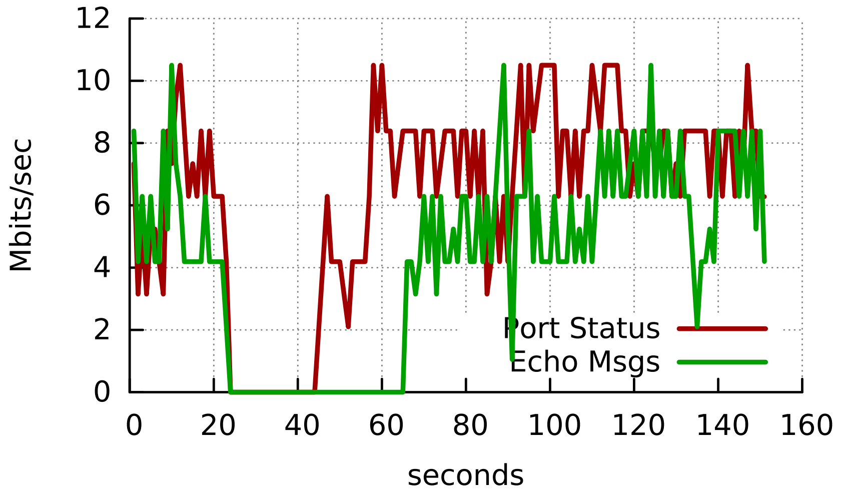


Figure 12: GENI Experimental Results

time for both methods is significantly large. Hence, in the future we will improve the tuning of our experimental parameters in order to decrease the failure recovery time in our experiments.

3.3 VIRO Failure Recovery: multiple links failure

Experiment Setup: in this experiment, we are particularly interested in showing how VIRO handles multiple network failures. To achieve this, we use the network topology illustrated in Figure 7(a). Similarly, we attach two client machines to nodes A (client-1) and C(client-2) and a server to node H. The network tool *iperf* is used to generate traffic from the client to the server. To illustrates how VIRO handles multiple failures we focus on the client-1 communication with the server. We failed the following sets of links and observed how the network recovers (see Figure 14): *B-G, D-I, D-G and F-I.*

Again, Figure 7(a) shows our network topology: 10 VIRO switches, 21 links, 2 clients and 1 server machines. Figure 8 shows our network topology in GENI. We use 14 XenVMs and three GENI Aggregate Managers (AMs): Northwestern (A, B, client-1), Wisconsin (C, D, E, F, client-2) and Illinois (I, J, G, H, server). To connect the nodes at different GENI AMs we use EGRE tunnels.

Experiment Discussion: using VIRO's routing protocol, the client at node A sends data packets to the server at node H using the following path: $A \rightarrow B \rightarrow G \rightarrow H$. However, after the failure of the link B-G, node A updates its level-4 gateway to node D, and it uses the following new path to communicate with the server: $A \rightarrow B \rightarrow D \rightarrow I \rightarrow G \rightarrow H$. However, if the link D-I fails, node D will use a different next-hop (node G) to reach its level-4. Hence nodes A's level-4 gateway (node D) is unchanged, although is level-4 path is replaced with a new path: $A \rightarrow B \rightarrow D \rightarrow G \rightarrow H$. On the other hand, if the link D-G fails, node A will receive a new level-4 gateway (node F) from its level-4 rendezvous point. Then, node's A path to communicate with the server will be the following: $A \rightarrow B \rightarrow D \rightarrow E \rightarrow F \rightarrow I \rightarrow G \rightarrow H$. Similarly, if link F-I fails, node's A level-4 gateway remains unchanged (node F), although is level-4 path to reach the serve is updated to a new path: $A \rightarrow B \rightarrow D \rightarrow E \rightarrow F \rightarrow J \rightarrow H$. These experimental results show how VIRO easily supports topology changes and it is robust to link failures in the network (see Figure 14).

From Figure 13 we observe that the failures of links B-G, D-I, D-G and F-I happen at 21, 75, 184 and 380 seconds. We also observe that the network recovers from all the links failures. However, the recovery time in this experiment is significantly large (see Figure 13). Hence, in the future we will improve the tuning of our experimental parameters in order to decrease the failure recovery time in a network composed with VIRO switches.

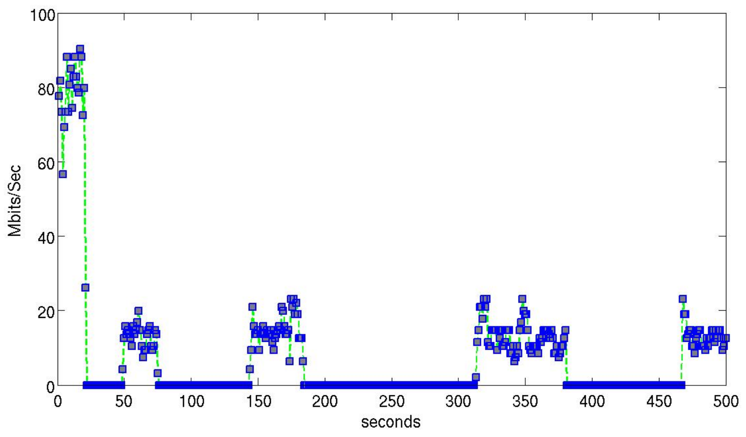


Figure 13: VIRO links failures recovery: multiple links failure

3.4 VIRO Supporting for Host Mobility

3.4.1 VIRO Mobility-1 Experiment

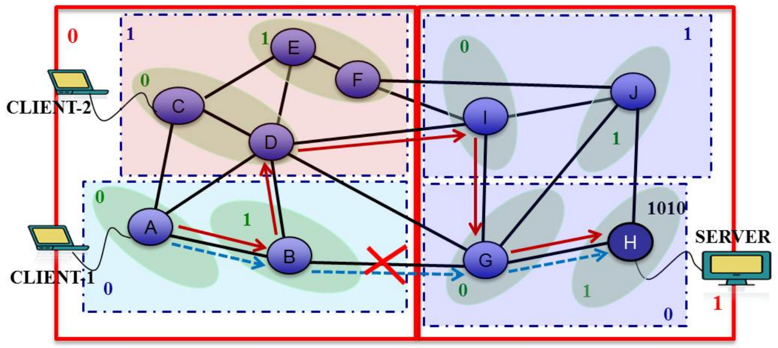
Experiment Setup: In this experiment, we show how VIRO handles host mobility. To achieve this, we use the network topology illustrated in Figure 15 . We attach a client host at node C and an Apache server at node G. To deploy this topology in GENI, we use the two AMs (*Wisconsin* and *Illinois*), 11 XenVMs and 4 PCs. Figure 16(a) shows our experimental set-up in GENI. The nodes at the same level in VIRO are placed in the same GENI PC, whereas nodes at different level are at distinct GENI PCs. Hence, from Figure 15, nodes A,B are in the same PC. Again, to connect the nodes at different GENI AMs we use EGRE tunnels.

Experiment Discussion: using VIRO's routing protocol, the client host moves from node C to B while downloading a large image from the server at G. To implement the host mobility in GENI, we attached an standard OVS to the client and to both nodes C and B (see Figure 15). Thus, we used OpenFlow rules to transfer the client's traffic from node C to B (see Figure 16(b)), this way emulating host mobility in GENI. During this process, a new VIRO vid is assigned to the client after moving to node B by the RC. The server finds the client new vid by issuing an ARP request to the remote controller, in the VIRO management plane. The client TCP connection is unaffected during this process. Therefore, VIRO *topology-aware, structure virtual id* space offers support for host mobility.

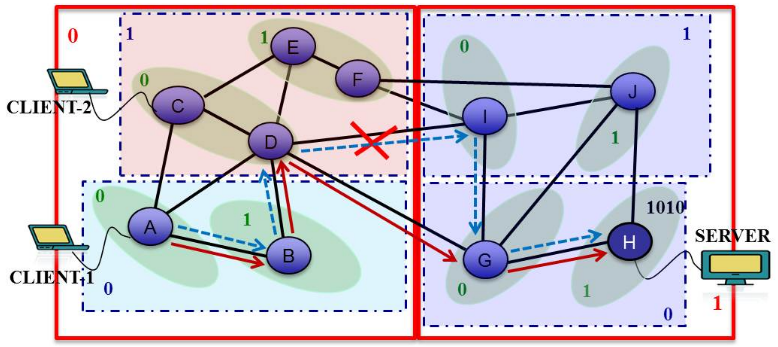
3.4.2 VIRO Mobility-2 Experiment

Experiment Setup: In this experiment, our goal is to show how VIRO handles host mobility. To pursue this goal, we carried out experiments using a network topology with 7 nodes in GENI. This topology is illustrated in Figure 15, the leaf nodes in the binary tree represent VIRO switches, and the color of the nodes represents GENI Aggregate Managers (AM). We attach a client host at node C and an Apache server at node G. To deploy this topology in GENI, we use the two AMs (*Wisconsin* and *Illinois*), 7 XenVMs and 4 PCs. Figure 17 shows our experimental set-up in GENI. The nodes at the same level in VIRO are placed in the same GENI PC, whereas nodes at different level are at distinct GENI PCs. Hence, from Figure 15, nodes A,B are in the same PC. Again, to connect the nodes at different GENI AMs we use EGRE tunnels.

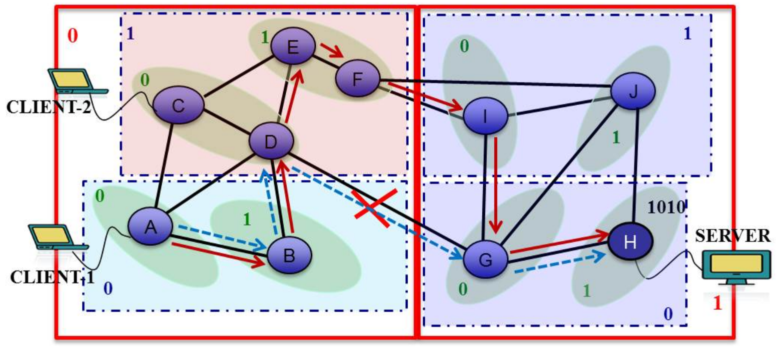
Experiment Discussion: using VIRO's routing protocol, the client host moves from node C to B while streaming a video from the server at G. To implement the host mobility in GENI, we connect 2 external VIRO nodes (B and C) to our VIRO topology in GENI using "Ethernet tunnels" (see Figures 20 and 21). Furthermore, we attached two VIRO wireless access points to our VIRO switches B and C (see Figure 20) and we use a Toshiba laptop as our client. Our experimental set-up is illustrated in Figure 20 and Figure 21 shows the experimental set-up in our lab. Using the above set-up, we connect the client to node C using LAN cables and we emulate client mobility by moving the client from node C to B using the LAN cables attached to each switch. Similarly, we use the VIRO access point 1 and 2 connected to the external nodes to emulate host mobility - client moves from VIRO AP 1 to



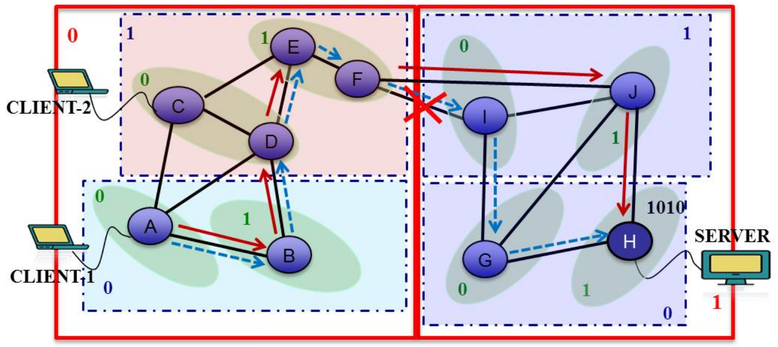
(a) link B-G



(b) link D-I



(c) link D-G



(d) link F-I

Figure 14: VIRO failure recovery (multiple links failure): the *blue lines* represent the network path before failure and *brown lines* represent the network path after failure

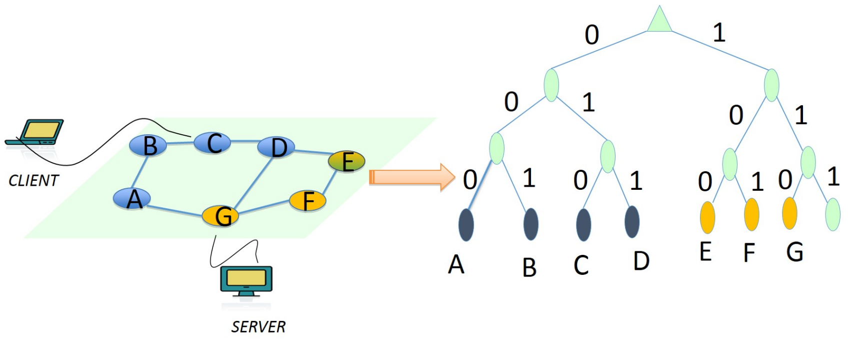
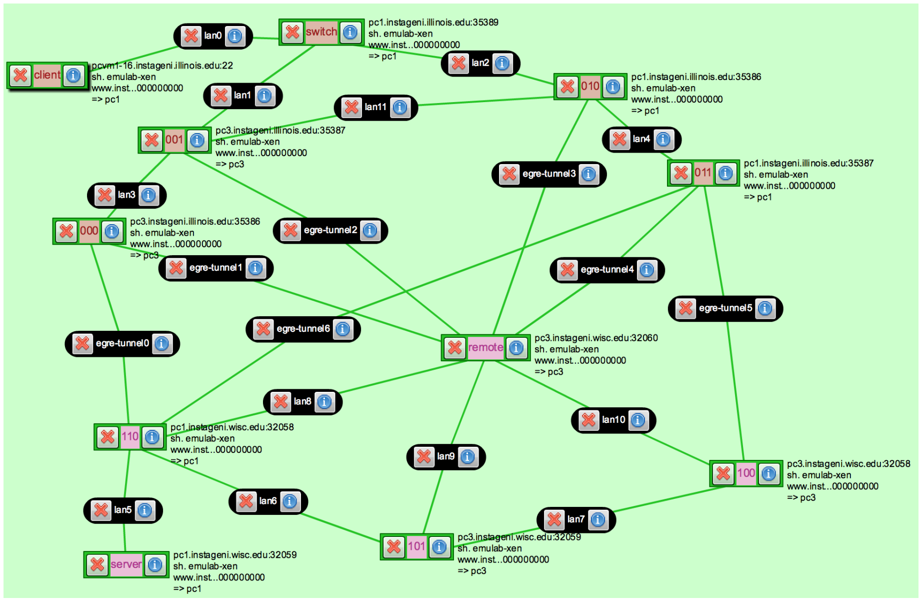
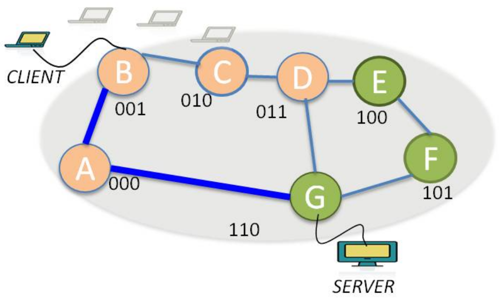


Figure 15: Network topology and its virtual binary tree representation



(a) GENI



(b) Network Topolgy

Figure 16: VIRO mobility-1 experiment setup

2. During this process, a new VIRO vid is assigned to the client after moving to node B (see Figure 25) by the RC. The server finds the client new vid by issuing an ARP request to the remote controller, in the VIRO management plane. The client TCP connection is unaffected during this process. Therefore, VIRO *topology-aware, structure virtual id* space offers support for host mobility.

In summary, existing Internet Protocol (IP) and Ethernet-based network technologies offer poor support for handling host and user mobility and handling network dynamics. For example, suppose you are currently downloading a large file on your laptop connected to a wired network at work. If you have to take your laptop to another floor

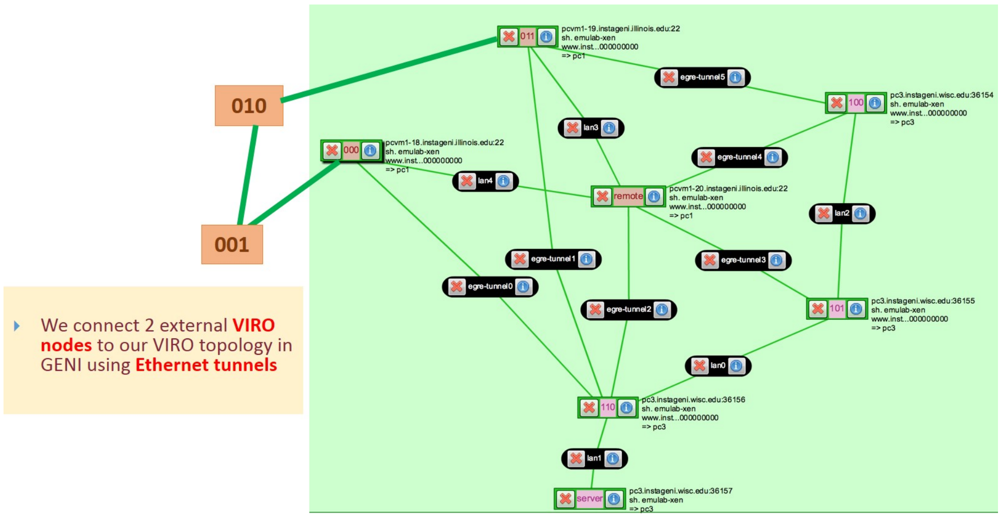


Figure 17: VIRO mobility-2 experiment setup

in your office building, say, to attend a meeting, your file download will stop because its connection to the web server will break, despite that your laptop is also connected to your campus WiFi network. However, with VIRO can successfully support host mobility and handle other network dynamics automatically. With VIRO, you will be able to download a large file on your laptop at work, and take your laptop to any floor in your office building and your file download will continue, because VIRO will dynamically switch between different networks and preserve your laptop connection to the web server (see Figure 22). Besides supporting mobility, with VIRO its also easier to configure, manage and secure your company network, university network or datacenter network, because VIRO is a plug-&-play, scalable and robust routing protocol

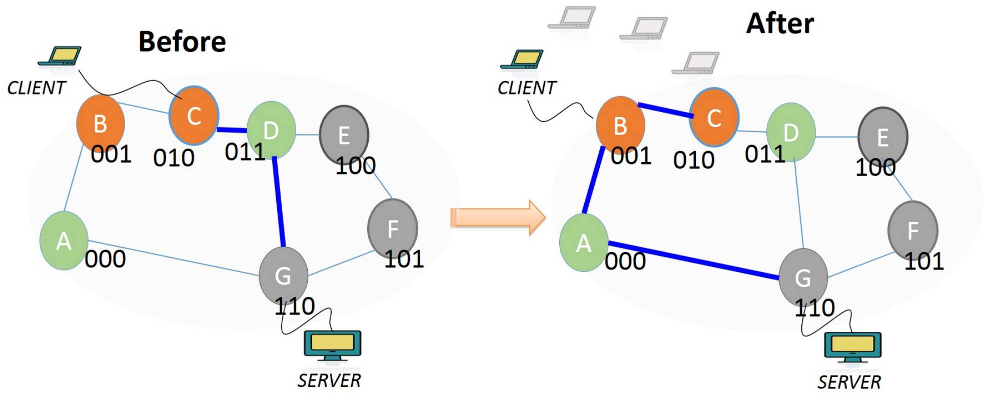


Figure 18: VIRO topology-aware, structured virtual id space offers supports for host mobility

Bucket Distance	Next hop	Gateway
1	D	C
2	B	C
3	D	D

(a) Before mobility

Bucket Distance	Next hop	Gateway
1	A	B
2	C	B
3	A	A

(b) After mobility

Figure 19: VIRO routing tables for node C (Round 3)

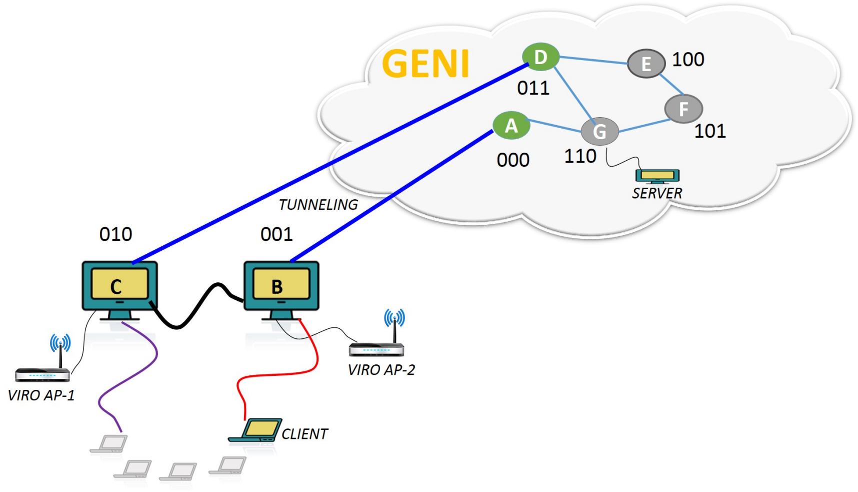
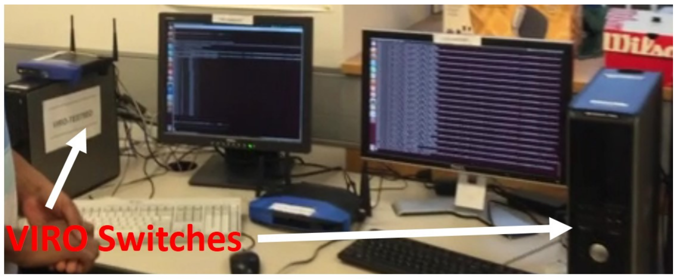
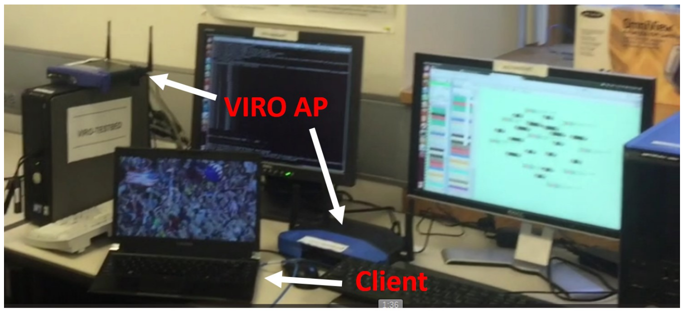


Figure 20: VIRO host mobility-2 experiment: using LAN cables and wireless access points



(a) VIRO switches



(b) VIRO access points and a client machine

Figure 21: VIRO mobility-2 experiment set-up in our lab: external VIRO nodes connected to our topology in GENI

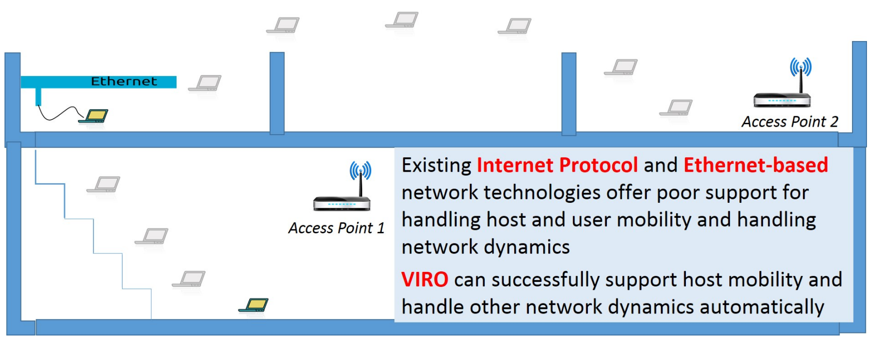
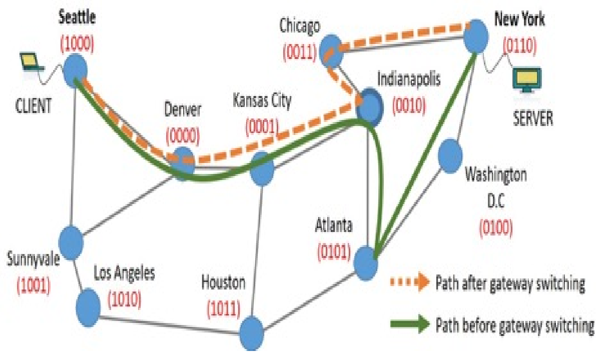

Ethernet
Access Point 2
Access Point 1
Existing Internet Protocol and Ethernet-based network technologies offer poor support for handling host and user mobility and handling network dynamics
VIRO can successfully support host mobility and handle other network dynamics automatically

Figure 22: VIRO supports host mobility better than existing Internet Protocol and Ethernet-based network technologies

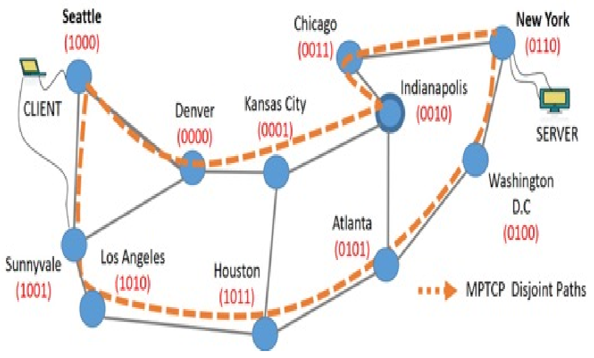
4 ADDITIONAL EXPERIMENTS

In this section, we describe an experiment evaluating our novel dynamic pathlet switching framework using VIRO for SDN netwtorks. Additionally, we describe an experiment evaluating our new routing paradigm – *routing via preorders* – which circumvents the limitations of conventional path-based routing shemes.

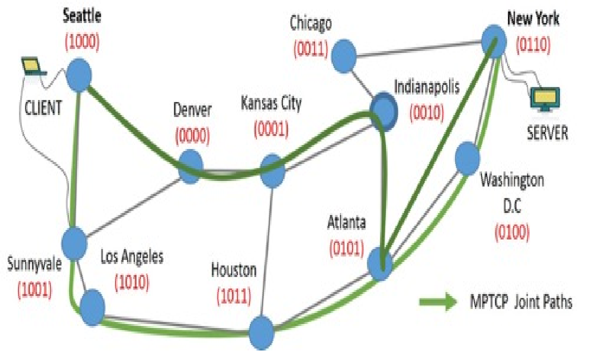
4.1 In-Network Dynamic Pathlet Switching with VIRO



(a) In-network path switching with VIRO-GENI nodes



(b) MPTCP flows using disjoint paths



(c) MPTCP flows using joint paths

Figure 23: Path Switching with VIRO and MPTCP

In [6] we propose a novel dynamic pathlet switching framework with VIRO for SDN Networks. We have conducted a number of experiments to evaluate the potential benefits of our framework in Mininet – in the future, we will extend our experiments to the GENI testbed. In these experiments we simulate[6] our in-network dynamic pathlet switching with VIRO and compare its results with mpTCP. Before presenting and discussing our experimental results, we provide here an example to illustrate how in-network path switching is performed in a network composed of VIRO-GENI switches. In this example as shown in Fig. 23(a), the client in Seattle communicates with a server in New York. Based on VIRO's routing tables the client's path to communicate with the server is the following: $Seattle \rightarrow Denver \rightarrow KansasCity \rightarrow Indianapolis \rightarrow Atlanta \rightarrow D.C. \rightarrow NewYork$ (see Fig. 23(a)). The LC, RC and rdv points will exchange messages, as discussed in [6], about the status of the paths' delay and gateway's throughput in the network. Now let's suppose that the link $Indianapolis \rightarrow Atlanta$ is congested. The RC will notify rdv 0000 about the poor performance of the GW 0010 – recall from [6] that a GW node periodically sends throughput information to the RC. Consequently, the rdv will notify node 0001 about the poor performance of this GW. Hence, node 0001 will change its level-3 gateway and start using node 0011[7] as its new level-3 GW. Then, the new path for the packets from client to server will be the following (*pathlet-switching*): $Seattle \rightarrow Denver \rightarrow KansasCity \rightarrow Indianapolis \rightarrow Chicago \rightarrow NewYork$. We are now in a position to discuss our experiments.

[6] i.e., by manually inserting OpenFlow rules to switch paths based on VIRO routing protocol

[7] a level-3 GW in the list of GWs received from the rdv point

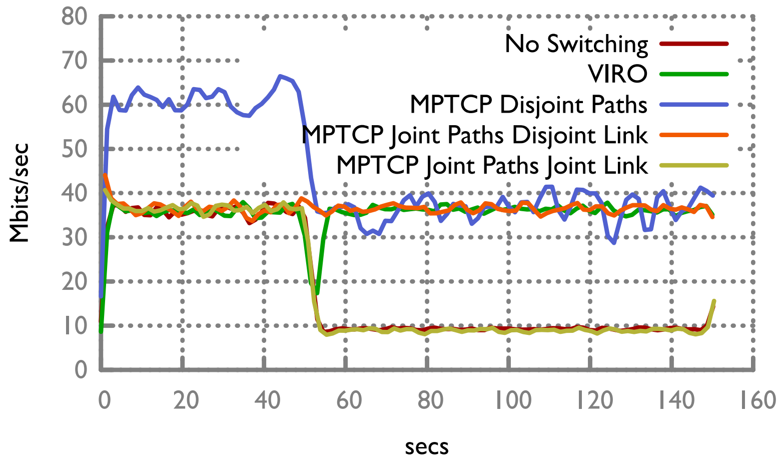


Figure 24: Patch switching experiments in Mininet: VIRO and MPTCP

Experiment Setup: we carry out experiments to investigate the potential benefits of in-network pathlet switching and we compare it with mpTCP. To achieve this, we use the network topology illustrated in Figure 23. As stated above, in this topology, a client in Seattle communicates with a server in New York. The network tool iperf is used to generate traffic from the client to server for 150 seconds. We insert openFlow rules into OVS switches to set-up all the paths in our experiments. The links in our experiments topology are all $40Mbits/sec$. Using the above set-up we carry out the following experiments:

1. No Switching: in this experiment, the client communicates with the server using the following path $Seattle \rightarrow Denver \rightarrow KansasCity \rightarrow Indianapolis \rightarrow Atlanta \rightarrow D.C \rightarrow NewYork$. During the experiment, we throttle the link $Indianapolis \rightarrow Atlanta$ to $10Mbits/sec$ after 50 seconds and we measure the throughput at the client-side (see Figure 23(a)).

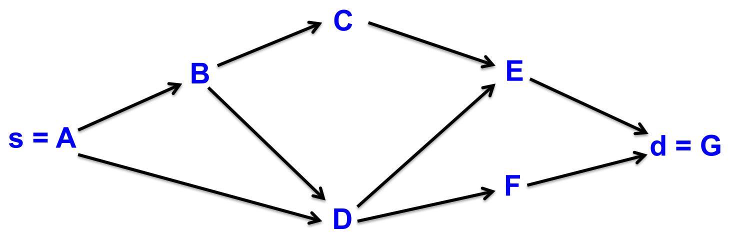
2. VIRO: we repeat experiment 1), but after throttling the link, we use openFlow rules to switch the client path, to a new path (in-network pathlet switching): $Seattle \rightarrow Denver \rightarrow KansasCity \rightarrow Indianapolis \rightarrow Chicago \rightarrow NewYork$. The new path is obtained by changing the VIRO gateways at node 0001, as described in the example above (see Figure 23(a)).

3. MPTCP Disjoint Paths: in this experiment, both client and server have two interfaces and they communicate using mpTCP. The mpTCP flows have the following disjoint paths, illustrated in Figure 23(b): a) $Seattle \rightarrow Denver \rightarrow KansasCity \rightarrow Indianapolis \rightarrow Chicago \rightarrow NewYork$; and b) $SunnyValle \rightarrow LosAngeles \rightarrow Houston \rightarrow Atlanta \rightarrow D.C. \rightarrow NewYork$. Similar to the above experiments, after 50 seconds we throttle the link $Indianapolis \rightarrow Chicago$ to $10Mbits/sec$ and measure the throughput at the client-side.

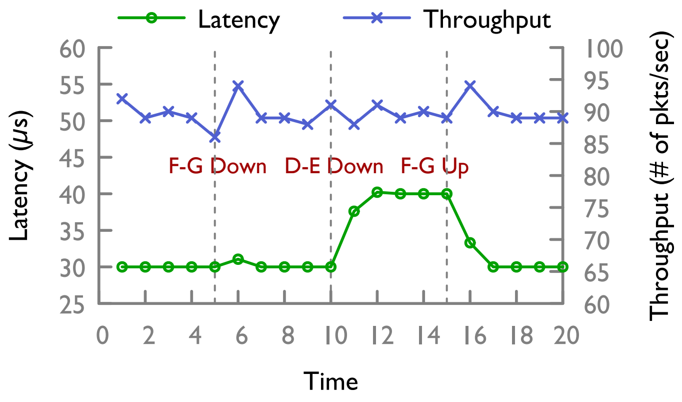
4. MPTCP Joint Paths Disjoint Link: as in 3), in this experiment, both client and server communicate using mpTCP. However, the mpTCP flows now have joint paths(see Figure 23(c)): a) $Seattle \rightarrow Denver \rightarrow KansasCity \rightarrow Indianapolis \rightarrow Atlanta \rightarrow D.C. \rightarrow NewYork$; and b) $SunnyValle \rightarrow LosAngeles \rightarrow Houston \rightarrow Atlanta \rightarrow D.C. \rightarrow NewYork$. Again, after 50 seconds we throttle the link $Indianapolis \rightarrow Atlanta$ to $10Mbits/sec$ and measure the throughput at the client-side.

5. MPTCP Joint Paths Joint Link: we repeat experiment 4), but now we throttle the link $Atlanta \rightarrow D.C.$ to $10Mbits/sec$.

Experiment Discussion: Figure 24 shows the results of our experiments. The results from experiment 1), 2), 3) and 4) show that switching paths significantly improves the application throughput when the performance of the current path degrades. As expected, mpTCP with disjoint TCP sub-flows paths provides the best throughput. However, when the TCP sub-flows share a set S of links in their paths and there is a congested link a with $a \notin S$ (Experiment 4), mpTCP performance is similar to VIRO. In contrast, if $a \in S$ (Experiment 5), mpTCP performs poorly and provides results similar to the traditional TCP. Experiment 2 shows that switching inside the network with VIRO provides similar results to mpTCP, when the TCP sub-flows have shared links, and in some cases can even outperform mpTCP when the TCP sub-flows share a congested link. In summary, mpTCP provides the best performance when the TCP sub-flows have mostly disjoint paths, which is difficult to guarantee in general.



(a) Primary PrOG



(b) Latency & Throughput

Figure 25: Routing Preorders: network topology and experimental results

However, with VIRO we can take advantage of the path diversity inside the networks and switch the network traffic between all the available paths, thereby improving application performance.

4.2 Adaptive Resilient Routing via Preorders in SDN

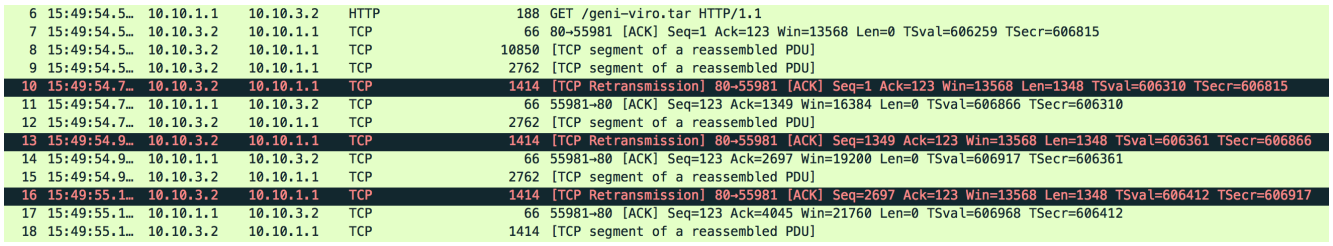
In [8] we propose and advocate a new routing paradigm – dubbed *routing via preorders* – which circumvents the limitations of conventional path-based routing schemes to effectively take advantage of topological diversity inherent in a network with rich topology for *adaptive resilient* routing, while at the same time meeting the quality-of-service requirements (e.g., latency) of applications or flows. We show how routing via preorders can be realized in SDN networks using the "match-action" data plane abstraction.

We have conducted experiments to evaluate the potential benefits of our novel routing paradigm in Mininet – in the future, we will extend our experiments to the GENI testbed. We evaluated routing via preorders in Mininet using the topology shown in 25(a). A UDP traffic generator at host h_1, attached to node A, sends packets to host h_2, attached to node G, with a latency requirement $\tau_F = 40\ \mu s$ under normal operations, and a *relaxed* latency $\tilde{\tau}_F = 60\ \mu s$ under failures. For each link $l = (i, j) \in E$, the value of $\phi(i, j) = 10\ \mu s$ determines the latency of routing packets along the link l. The flow entries in each switch are prioritized to prefer shorter paths, and the neighbor id is used as a tie-breaker, ie in 25(a) the default path is $A \to D \to F \to G$.

The traffic lasts for 20 seconds, sending around 90 packets per second. In normal operations, packets traverse the shortest path. For link failures, after 5 seconds, we fail the link $F \to G$, then F becomes a sink node, and D uses its other outgoing link E. Thereafter, packets traverse the path $A \to D \to E \to G$, which has the same latency as the previous path. Transient packets at node F are rerouted back to D, which forwards them to G through E meeting their relaxed latency $\tilde{\tau}_F$. After another 5 seconds, we fail the link $D \to E$, which makes D a sink node. Thus, D deactivates its incoming links $A \to D$ and $B \to D$. Now, the only remaining path is $A \to B \to C \to E \to G$. Finally, after another 5 seconds, the link $F \to G$ is brought up, and triggers F to activate its incoming link $D \to F$. Then, D activates its incoming links $A \to D$ and $B \to D$. Hence, A sends packets to G through the default shortest path again $A \to D \to F \to G$.

Figure 25(b) shows the average latency of the packets received at G based on the links they traversed. We see that the latency reflects using the paths in the order explained above. After the failure of the link $F \to G$, the latency increased slightly for a short time due to rerouting the transient packets. Moreover, our proof-of-concept prototype shows that the connectivity between the two hosts is not affected by link failures as long as there is a path connecting them, and the throughput is not affected as shown in 25(b). Furthermore, packets were transmitted within the specified (relaxed) latency constraint of the flow, and there was no packet loss. This is due to the deactivation mechanism which trims edges leading to sink nodes, while rerouting transient traffic to another outgoing port which can reach the destination, without the need to change routing rules or involve the SDN controller.

5 Lessons Learned Using GENI and Future Directions



```
 6 15:49:54.5…  10.10.1.1   10.10.3.2   HTTP     188 GET /geni-viro.tar HTTP/1.1
 7 15:49:54.5…  10.10.3.2   10.10.1.1   TCP       66 80→55981 [ACK] Seq=1 Ack=123 Win=13568 Len=0 TSval=606259 TSecr=606815
 8 15:49:54.5…  10.10.3.2   10.10.1.1   TCP    10850 [TCP segment of a reassembled PDU]
 9 15:49:54.5…  10.10.3.2   10.10.1.1   TCP     2762 [TCP segment of a reassembled PDU]
10 15:49:54.7…  10.10.3.2   10.10.1.1   TCP     1414 [TCP Retransmission] 80→55981 [ACK] Seq=1 Ack=123 Win=13568 Len=1348 TSval=606310 TSecr=606815
11 15:49:54.7…  10.10.1.1   10.10.3.2   TCP       66 55981→80 [ACK] Seq=123 Ack=1349 Win=16384 Len=0 TSval=606866 TSecr=606310
12 15:49:54.7…  10.10.3.2   10.10.1.1   TCP     2762 [TCP segment of a reassembled PDU]
13 15:49:54.9…  10.10.3.2   10.10.1.1   TCP     1414 [TCP Retransmission] 80→55981 [ACK] Seq=1349 Ack=123 Win=13568 Len=1348 TSval=606361 TSecr=606866
14 15:49:54.9…  10.10.1.1   10.10.3.2   TCP       66 55981→80 [ACK] Seq=123 Ack=2697 Win=19200 Len=0 TSval=606917 TSecr=606361
15 15:49:54.9…  10.10.3.2   10.10.1.1   TCP     2762 [TCP segment of a reassembled PDU]
16 15:49:55.1…  10.10.3.2   10.10.1.1   TCP     1414 [TCP Retransmission] 80→55981 [ACK] Seq=2697 Ack=123 Win=13568 Len=1348 TSval=606412 TSecr=606917
17 15:49:55.1…  10.10.1.1   10.10.3.2   TCP       66 55981→80 [ACK] Seq=123 Ack=4045 Win=21760 Len=0 TSval=606968 TSecr=606412
18 15:49:55.1…  10.10.3.2   10.10.1.1   TCP     1414 [TCP segment of a reassembled PDU]
```

Figure 26: Wireshark screen-shot: very large frames are being sent out at the server side

With our implementation of VIRO using the extended OVS/SDN software platform, we have successfully deployed our prototype in GENI [5]. In conducting GENI experiments, we find that reserving resources for complex topologies using Flack, Jack and JFed often requires several attempts which can take long time. Thus, Omni[8] could be a better tool to be used to reserve resources for large topologies, although it is less user-friendly and you need to create the RSpec file manually to build the experiment topology. We will use Omni to reserve the resources for our experiments in the future. We also find that once the resources are reserved in GENI, it is not possible to dynamically change the experiment network topology. In addition, we find that the number of available XenVMs is limited in some GENI Aggregate Managers (AMs). Hence we could not deploy large VIRO topologies at these AMs. To connect nodes at different GENI AMs, we first attempted to use stitching, but we were unable to get the resources. Instead, we have used the EGRE tunnels for links across AMs. With the new version of omni (omni v2.6), creating stitching links have become easier, we have explored the possibility of using vLAN stitching links to further improve our implementation. However, despite considerable efforts devoted to this, we were not able to get the needed resources and make VLAN link switching work for VIRO. This is primarily due to the fact that VIRO rewrites and re-purpose the standard Ethernet/VLAN headers for VIRO packet headers, which may confuse the GENI testbed substrate. Our work illustrates the potential limitations of the current GENI testbed in supporting non-IP protocol experimentation in large scales, when network virtualization is insufficient and bare metal testbed facility is needed.

In conducting GENI experiments, we have also run into other issues and difficulties, in particular, as VIRO uses a different layer-2 frame format than Ethernet. When we are using GENI, basically we exchange VIRO frame through "links" we reserved. We ran into an interesting problem when testing VIRO user opt-in experiments to support host mobility. The problem occurred when we were testing the protocol by downloading big files using TCP. Compared with general speed we can get between two VMs on GENI (10MB/s), the performance (10-30KB/s) was really bad when using VIRO. Since we also changed OVS and added some customized actions, we firstly conducted many experiments (both locally and on GENI) to rule out these factors. Long story short, the performance degradation is not due to our protocol, or our modified OVS. Then we tried to capture the wireshark trace when downloading files. Surprisingly, we found many huge frames (6000Bytes, which is much larger than MTU 1500) were sent out from server side (see Figure 26), and those huge frames got dropped on the link. Then server side must re-transmit previous content by using a smaller window size, while still keeping trying to increase the frame size. This causes the bad performance. As a comparison, if we do not use our protocol VIRO and just have two connected VMs, we can still see those huge frames being sent, but this time they arrive at the other end. This phenomenon makes us believe that the MTU setting actually may not work for those virtual interfaces created by Xen. We speculate that since Xen thinks these are both virtual interfaces, doing segmentation may

[8]Omni is a GENI command line tool for reserving resources

not be that useful when the frame can be directly dumped from one place to another. The only thing that Xen needs to care about is to identify where the frame begins and ends, by checking the payload size. (Physically, this is done by checking the specific bit patterns signaling the start and end of a frame.) However, for huge frames that are of a format which Xen doesn't know, Xen seems to drop it because Xen does not know where the frame starts and ends. Since the operating system tends to offload the segmentation work to the interface, then we got a problem here. To validate our hypothesis, we tried disabling the segmentation offload in the OS settings, thereby forcing the operating system to do the segmentation on its own. This time the downloading speed went back to normal, similar to the speed when we use normal Ethernet frame. To disable the offloading function, we used a tool in Linux called "ethtool" (http://www.linuxfoundation.org/collaborate/workgroups/networking/tso). In addition, when attempting to deploy our pathlet switching framework using VIRO in GENI and compare it with mpTCP, we fond that when installing mpTCP kernel in the Xen VMs in GENI, we are unable to change the default kernel. This seems to be a issue with the Xen configuration itself in GENI. We have sought help resolving this issue, but no avail so far. This limits our ability to conduct further experiments in GENI. All these difficulties and issues further illustrate the potential limitations of the current GENI testbed in supporting non-IP protocol experimentation in large scales.

We plan to expand our current prototype of VIRO to include additional functionalities. These include further extend the OVS software platform to support multi-path routing and resilient routing as well as additional management functions such as access control mechanism. In addition, we plan to evaluate the scalability of our architecture in GENI over larger topologies and to incorporate VIRO in GENI – as a non-IP service – to support research, experiments and educational activities by other GENI researchers.

References

[1] GENI: Exploring Networks of the Future. [Online]. Available: https://www.geni.net/

[2] OpenvSwitch. [Online]. Available: http://www.openvswitch.org

[3] S. Jain, Y. Chen, and Z. Zhang, "VIRO: A Scalable,Robust and Namespace Independent Virtual Id Routing for Future Networks", in Proc. of INFOCOM, 2011.

[4] H. Mekky, C. Jin, and Z. Zhang, "VIRO-GENI: SDN-based Approach for a Non-ip Protocol in GENI", in Proc. GREE, 2014.

[5] B. Dumba, G. Sun, H. Mekky, Z. Zhang, "Experience in Implementing & Deploying a Non-IP Routing Protocol VIRO in GENI ", in Proc. CNERT, 2014.

[6] B. Dumba, H. Mekky, G. Sun, Z. Zhang, "In-Network Dynamic Pathlet Switching with VIRO for SDN Networks", in Proc. CNERT, 2015.

[7] B. Dumba, H. Mekky, S. Jain, G. Sun, Z. Zhang, "A Virtual Id Routing Protocol for Future Dynamics Networks and Its Implementation Using the SDN Paradigm", in Journal of Network and Systems Management, 24(3), 578-606.

[8] E. Ramadan, H. Mekky, B. Dumba, Z. Zhang, "Adaptive Resilient Routing via Preorders in SDN", in Proc. DCC'16, 2016.