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

Braulio Dumba, Guobao Sun, Hesham Mekky, Zhi-Li Zhang
{braulio,gsun,hesham,zhzhang}@cs.umn.edu
University of Minnesota, Twin Cities

***Abstract*—In this paper, we describe our experience in implementing a non-IP routing protocol – *Virtual Id Routing (VIRO)* – using the OVS-SDN platform in GENI. As a novel, "plug-&-play", routing paradigm for future dynamic networks, VIRO decouples *routing/forwarding* from *addressing* by introducing a *topology-aware*, *structured* virtual id layer to encode the locations of switches and devices in the physical topology for scalable and resilient routing. Despite its general "match-action" forwarding function, the existing OVS-SDN platform is closely tied to the conventional Ethernet/IP/TCP header formats, and cannot be directly used to implement the new VIRO routing/forwarding paradigm. As a result, we repurpose the Ethernet MAC address to represent VIRO virtual id, modify and extend the OVS (both within the user space and the kernel space) to implement the VIRO forwarding functions. We also utilize a set of *local* POX controllers (one per VIRO switch) to emulate the VIRO *distributed* control plane and one *global* POX controller to realize the VIRO (*centralized*) management plane. We evaluate our prototype implementation through the Mininet emulation and GENI deployment test and discuss some lessons learned using the test-bed.**

I. Introduction

The rapid growth in the number of computers, mobile devices, smart appliances and other machines connected to the internet today has increased the burden on the network substrate. Such rapid growth also expedited the need to address some of the well-known shortcomings of existing networking technologies that "glue" the Internet together. For instance, the Internet Protocol (IP) tightly couples network layer functions such as addressing and routing, making it difficult to transition from IPv4 to IPv6. It has poor support for mobility. Furthermore, IP routers require extensive manual configuration. In contrast, layer-2 technologies such as Ethernet need only minimal configurations: Ethernet switches automatically learn MAC addresses of hosts to build switching tables. However, Layer-2 Ethernet technology does not scale to large (& wide-area networks), as it provides sub-optimal routing and is not robust to failures.

To address these challenges, we need better layer-2/layer-3 networking technologies that is more *scalable* (e.g., with small routing tables with fast lookup speed), provide better support for *mobility* (e.g., by separating location/addressing and identity/naming), provide high *availability and reliability* (e.g., via proactive failure discovery and by localizing effects of failures). Furthermore, such technologies should be easy to *manage and deploy* – ideally, with the abilities to self-configure and self-organize, and are endowed with stronger security capabilities. Several Non-IP based routing and network architectures [3][6][4][5] have been proposed to mitigate some of the limitation of the current Internet technologies. However, deployment and testing of these solutions at scale have always been a huge challenge.

978-1-4799-6204-4/14$31.00

The emergence of Software Defined Networks (SDNs) and OpenFlow capable switches, such as Open vSwitch (OVS) [2] makes testing and experimenting with future technologies easier. SDN increases network programmability by decoupling the data and control planes [10][9]. It provides an unified API through which a centralized controller can configure and control the forwarding behaviors of switches. Hence, it simplifies the task of configuring and managing large networks. The SDN paradigm has been widely embraced by the research community and adopted in large test-beds such as the Global Environment for Network Innovation (GENI) [1], a wide-area test-bed developed by the research community to enable network innovations and large scale experimentations. As part of its network infrastructure, GENI has employed the OVS-SDN software platform to facilitate testing and deployment for large scale experiments.

Virtual Id Routing (VIRO) is a novel "plug-&-play" routing paradigm for future large dynamic networks [3]. It addresses the limitations faced by the layer-3 (L3) IP routing protocols as well as the layer-2 (L2) Ethernet switching technology, while retaining the latter's plug-&-play feature. VIRO decouples routing/forwarding from addressing, and provides a (L2/L3) *convergence* layer that unifies the conventional L2/L3 routing/forwarding functionalities. VIRO is *namespace-independent* and allows new addressing schemes to be introduced into networks with no changes in the core routing and forwarding functions in the network data plane devices. The fundamental idea of VIRO is the introduction of a *topology-aware, structured virtual id space* onto which physical identifiers and high level names can be mapped. VIRO employs a DHT (distributed hash table) style routing algorithm to build routing tables, look up objects (name, addresses, vid's, etc) and forward packets [3]. Therefore, VIRO eliminates flooding both in the data and control planes. Furthermore, VIRO is highly scalable, localizes failures, supports multi-homing, fast rerouting, multipath routing and it is easy to manage and deploy. By decoupling addressing from routing, it also enables access control as packets enter a network, and allows other security features to be incorporated into the network control and management more seamlessly.

In this paper, we describe our experience in implementing and deploying VIRO in GENI using the SDN platform. We have implemented an initial prototype of VIRO in GENI, and our goal is two-fold: firstly, to test and evaluate VIRO's functionality and performance in GENI, and in the long term to incorporate VIRO in GENI, as a non-IP service, to support research, experiments and educational activities by other GENI

Bucket Distance	Gateway	Nexthop
1	-	-
2	A	B
3	A	C,D
4	C, D	C,D
5	B,D,M,N	B,C,D

Fig. 1: VIRO routing table for node A.

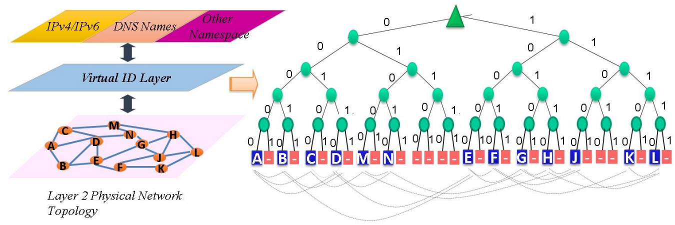


Fig. 2: vid space as a virtual binary tree: the grey dotted lines denote physical connectivity and the red boxes represent the unused vid's

researchers. To our knowledge, we are the first to deploy and test a non-IP routing protocol in GENI. The contributions of this paper are as follows:

- We implement and deploy an initial prototype of VIRO in GENI using the SDN platform.
- We perform experiments to evaluate VIRO packet's encapsulation/decapsulation overhead and our failure recovery mechanisms. The results of these experiments will help us to improve and extend our VIRO prototype.
- We describe our experience and lessons learned in implementing and deploying a non-IP protocol in GENI.

The remainder of the paper is organized as follows. Section II provides an overview of VIRO. Section III discusses our implementation and deployment of VIRO in GENI. Section IV presents our experiments and discusses our experimental results. We conclude and discuss future work in Section V.

II. VIRO: Virtual Id Routing Protocol

In this section, we provide an overview of VIRO's three main components: *vid space construction and vid assignment, VIRO routing, and vid lookup and forwarding*. For more details about the VIRO routing protocol, the reader is referred to [3].

VIRO is a topology-aware, structured virtual id (vid) routing protocol for future networks. It introduces a self-configurable, self-organizing virtual id layer (layer-2/3 convergence layer) where both physical identifiers (e.g. MAC addresses), as well as higher layer addresses/names (e.g., IPv4/IPv6 or flat-id names) are mapped [3]. VIRO's structured vid space embeds the physical network topology formed by the connections among physical network components. Such embedding is illustrated in Figure 2 using a Kademlia-like [7] *virtual binary tree*, where the physical devices (e.g. switches) are represented by the leaf nodes. All intermediary nodes in the virtual binary tree are logical nodes labelled with the bit-strings representing the *vid's* of the VIRO switches residing in that subtree. Next, we describe the main components of VIRO:

Vid space construction and vid assignment: at the network bootstrapping phase, the topology-aware, structured vid space is constructed. VIRO uses a Kademlia-like *virtual binary tree* to structure the vid space, where each VIRO switch (a leaf node of the tree) is assigned an L-bit string *vid* corresponding to the bits from the root to that leaf node. After the network bootstrapping process, the *vid* of a new VIRO node joining the network is assigned based on the *vid's* of its physical neighbors. When an end-host attaches to a VIRO switch, it is assigned an extend *vid* comprised of the L-bit vid of the switch plus a random l-bit host id. This virtual id space preserves the physical proximity of the nodes.

Routing Tables Construction: VIRO routing tables are constructed based on the *vid* logical distance(σ)[1] between the nodes for each level of the vid space. It employs a DHT-like "publish-&-query" mechanism, where each node publishes and queries routing information to *rendezvous* nodes[3]. VIRO completely eliminates network flooding in both the control and data planes. VIRO's *vid* prefixes are used to aggregate routing information for sections of the network (e.g. 10***). Thus, VIRO routing table size is O(log N), where N = number of nodes in the network. In VIRO, failure of a link or switch are localized because no switch needs to maintain a network-wide full topology.

Virtual Id lookup and Forwarding: forwarding of packets in a VIRO network is performed using *vid's* only. However, at the networks edges the *vids* are mapped to persistent identifier (pid), e.g. MAC address/IP address, or vice-versa, in order to locate the end-hosts or to route VIRO packets in the network. We will illustrate the packet forwarding mechanism using Figures 1 and 2. Suppose node A with vid = 00000 wants to send a data packet to node F with vid = 10010 (see Figure 2). Node A will first compute the logical distance (σ) between its vid and F's vid, which in this case is 5[2]. Then, node A will look up for a level-5 gateway(GW) in its routing table (see Figure 1), and forward the packet to the next-hop C to reach M[3]. The next-hop follows similar process until the packet is delivered to the destination. However, node A will directly forward the packet to F, if it is physically attached to node F. Similar process is used to forward control packets, whose destination *vid* identifies a VIRO switch.

III. DESIGN AND IMPLEMENTATION

In this section, we present the design & implementation framework of VIRO-GENI. First, we discuss the challenges in implementing VIRO, a non-IP protocol using the OVS/SDN platform. Next, we describe our data/control/management plane solutions to address the challenges. We conclude this section by providing a detailed example of how forwarding is done in a VIRO-GENI network.

A. Implementation Challenges

The OVS software platform is derived from the OpenFlow switch specifications and SDN paradigm. When compared to

[1] σ = L - length of the longest common prefix.

[2] L= 5; $\sigma = 5 - 0 = 5$

[3] In this example, M is the default GW for A's level-5.

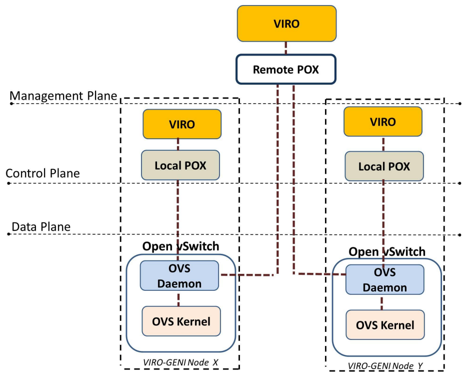


Fig. 3: Software Stack in VIRO-GENI Node

traditional network devices (e.g. Ethernet switches and IP routers), Openflow and OVS enable a far more flexible data plane with configurable forwarding behaviors at the "flow" level, which are defined by the "match-action" rules specified by a SDN controller. Nonetheless, the existing Openflow/OVS/SDN platforms are strongly tied to the conventional Ethernet/IP/TCP protocol stack. In contrast, VIRO has its own "topology-aware" addressing (vid's) scheme, with its unique routing and forwarding behaviors. It employs a *distributed routing* protocol with a novel "pub-sub" mechanism [3] and it has build-in multipath and fast failure (re)routing capabilities. VIRO forwarding is done by using both the destination vid (via *vid* prefix matching) and a forwarding directive to look up VIRO routing tables to select a gateway and then the next-hop. Thus, VIRO's forwarding behavior cannot be directly implemented using the standard "match-action" function of OpenFlow.

GENI has employed the OVS-SDN software platform. Hence, we cannot directly deploy and test a non-IP protocol in the testbed because of the limitations of existing OVS-SDN platform. In the following, we present our design & implementation framework as well as solutions to overcome the challenges in adapting the OVS/SDN platform in implementing a non-IP protocol such as VIRO in GENI.

B. Design Overview

We modify the OVS software to implement VIRO switching functions in VIRO-GENI switches (nodes), and adapt SDN controllers to implement VIRO control and management plane functions, see Figure 3. As will be detailed further below, VIRO-GENI nodes use OVS in the data plane and POX controllers in the control and management planes. Each node runs the software stack shown in Figure 3. In the data plane, we repurpose the Ethernet MAC address to represent VIRO virtual id. We also modify and extend the OVS match-action (both within the user and kernel spaces) to realize VIRO packet forwarding functions. The (slow-path) OVS daemon (in the user space) connects to an OpenFlow local controller (LC) that executes the VIRO module which is responsible for running the VIRO routing protocol. Furthermore, the OVS daemon connects to a remote controller (RC), which is responsible for VIRO's management plane.

C. Data Plane

For the data plane implementation, we use OVS version 1.0 with Nicira's extensions. To route VIRO packets in the data plane, we first define the VIRO frame [8], see Figure 4. It extends the Ethernet frame in a similar way to the VLAN protocol. VIRO frame has the EtherType 0x0802 to differentiate it from standard Ethernet protocols such as IP, LLDP and ARP. In a VIRO frame, we reuse the 6-bytes of the source and destination MAC addresses (SMAC and DMAC) of the standard Ethernet frame, to set the VIRO virtual addresses (Vids). From the DMAC, it uses 4-bytes for the destination switch's vid (DVID), and 2-bytes for the destination host (DHOST) identifier. Similarly for the SMAC, it uses 4 and 2 bytes to set the source switch's vid (SVID) and source host (SHost) identifier.

After the 12 bytes in the Ethernet frame, we introduce new 6 bytes for the VIRO protocol header, where we have 2-bytes for VIRO's protocol identifier (VPID), and the last 4-bytes are the forwarding directive (FD)[3]. The remaining bytes in the VIRO frame are used to encapsulate the EtherType and the payload of the original Ethernet frame [4]. In the original Ethernet frame, we add a new EtherType 0x0803 for VIRO control packets (see Section III-D). By adding 6-bytes to the Ethernet frame header, to include VIRO protocol header, we use Path MTU Discovery at the end-hosts to reduce their frame size, in order to avoid encapsulation without using any fragmentation [8].

To forward our VIRO frame in the OVS data-path, we extend the match/actions in the OpenFlow protocol, because the current OpenFlow standard is still tied to the Ethernet/IP/TCP protocol stack. Thus, in order to route VIRO packets, we modify and extend both the OVS fast and slow path with new actions: insert/remove VIRO headers, rewrite the forwarding directive and match on VIRO switch's vid. With these additions, the OVS fast and slow path are now responsible for the following tasks:

- *OVS Daemon (Slow-Path)*: to translate between IP packets/VIRO packets (EtherType, FD) and to insert rules for routing at Kernel.
- *OVS Kernel (Fast Path)*: to translate between IP packets/VIRO packets (end-host), to forward IP packets among local machines and to forward VIRO packets.

Table I shows the list of new actions, we have added in both fast and slow paths:

Actions	Descriptions
`PUSH_FD`	add VPID and FD
`POP_FD`	remove VPID and FD
`SET_VID_SRC_SW`	set the first 4 bytes of the SVID
`SET_VID_SRC_HOST`	set the last 2 bytes of the SHost
`SET_VID_DST_SW`	set the first 4 bytes of the DVID
`SET_VID_DST_HOST`	set the last 2 bytes of the DHost
`SET_VID_FD_SW`	set first 4 bytes of the FD
`SET_VID_FD_HOST`	set the last 2 bytes of the FD

TABLE I: List of the new actions added to our extended OVS.

In addition to routing VIRO packets, the data-plane also forwards normal Ethernet frames for packets transmitted among local hosts, attached to the same VIRO-GENI node for example.

In the remaining of this section, we summarize the functions implemented by the different modules illustrated in Figure 3 (data, control and management planes).

[4]They form the payload of the VIRO frame

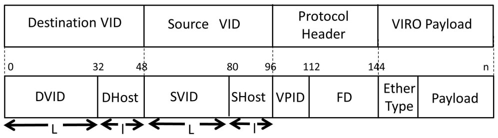


Fig. 4: VIRO Frame

D. Control Plane

The VIRO module attached to LC running on each node is responsible for the following control-plane functions: neighbor discovery, failure recovery, building the routing table, handling VIRO packet encapsulation/decapsulation and packet miss from the data-plane (see Section III-F and III-G). VIRO control packets are identified by the protocol ID 0x0803 in the frame payload (EtherType) to differentiate them from VIRO data packets (e.g. IP packets). The types of control packets handle by LCs are the followings:

- *RDV_Publish, RDV_Query, RDV_Reply*: used to publish, query or reply routing information from/to VIRO rendezvous nodes.
- *GW_Withdraw, GW_Remove*: used to advertise failed gateways information to others nodes.
- *Controller_Echo*: used to assign switch's vids by the RC.
- *Neighbor Echo Request & Reply*: heartbeat messages used to discover the physically attached switches.
- *Local_Host*: used to send host addresses mapping to the LC.

For an explanation of how these packets are used in the VIRO routing protocol, the reader is referred to [3] and to Section III-F.

E. Management Plane

As illustrated in Figure 3, every VIRO-GENI node will be connected to a single remote controller. Unlike the local controller (LC), the remote controller (RC) is the only instance that all OVSs in the network connect to. The purpose of this controller is to simplify the management plane functions that can be performed in a centralized fashion. For instance, the RC is responsible for the following: network topology discovery and maintenance (host/switch added or removed), vid assignment (host and switches), ARP and DHCP request[5] and IP/VID/MAC/PORT mapping (Global view). In Section III-F we discuss in details the RC's functions.

F. VIRO-GENI Network Boostrapping

In this subsection, we present the main events that occur during the bootstrap of a VIRO-GENI network. Recall that the OVS in each node in the network is connected to a local controller running the VIRO module, and all the nodes are connected to the same remote controller for management plane functions.

Connection Up: initially, when a VIRO-GENI switch starts it connects to both local controller(LC) and remote controller (RC) using the standard OpenFlow protocol. The RC will insert rules to receive all the ARP and DHCP packets generated by host machines. We assign Ids to the controllers (RC-Id = 1 and LC-Id = 2). The VIRO-GENI switch uses these Ids to differentiate both controllers, e.g.: ARP packets are sent to controller with Id=1.

Vid Assignment: a VIRO-GENI switch gets its vid from the RC. The RC constantly sends *Controller_Echo* message every k seconds to the LCs with the vid of the respective switch [6]. However, host's vids are assigned when a host issues a DHCP request. Whenever a RC leases an IP addresses, it also assigns the host vid – first L-bits (4-bytes) from the host access node and last l-bits (2-bytes) for the DHost.

After assigning the vids, the RC saves the mapping DPID/VID for switches and the mapping MAC/IP/VID/PORT for hosts to its topology table, in order to build its global view of the network. In addition, after the host's vid assignment, RC will add the host to the list of "attached host" for the respective switch (access-node). Furthermore, it sends the host's address mapping information to the respective LC.

Neighbor and Failure Discovery: the Local VIRO modules attached to each node find the physically attached switches by exchanging *Neighbor Echo Request & Reply* messages every i seconds. The VIRO module, in each node, has a table for saving the neighbors' vids. This table is updated whenever a neighbor Echo Reply message is received, and the last updated time is recorded for each entry. We use this table to find the failed neighbors, for example: if an entry is not updated after j seconds, then we consider this neighbor switch as failed. We also use OpenFlow *Port Status* messages for neighbor failure discovery. VIRO handles nodes failures without resorting to flood of failure notification (as used in OSPF), instead, it utilizes a *withdraw and update* mechanism [3].

Routing Table Construction: the local VIRO module in each VIRO-GENI node builds its routing tables using the *publish-&-query* algorithm described in [3] (*RD_Publish, RDV_Query, RDV_Reply* messages). These routing tables are installed into the OVS slow-path flow-tables to immediately perform any intermediate packet forwarding [8].

End-Host Discovery: the VIRO module connected to the LC discovers the end-host attached to its local switch from *Local_Host* messages receive from the RC during the DHCP lease process. It stores the mapping IP/MAC/VID/PORT for future end-host name resolution, and it uses this mapping to build its local view.

Pid-Vid Resolution: in the original design of VIRO[3], a one-hop (multi-hop) DHT is used for pid-vid look-up and resolution. For simplicity, in our current implementation, we use a centralized approach for pid-vid resolution: i) When an end host (VM) joins a VIRO network, it first runs DHCP. The DHCP request is captured and sent to RC by the VIRO switch attached to it. After leasing an IP address to an end host, RC assigns the host vid and it saves the mapping pid-vid in its topology table; ii) When one end host x wants to communicate with another host y in a VIRO network, it first issues an ARP request. The VIRO switch attached to it forwards this packet to the RC. Then, RC returns host y vid in the ARP reply by replacing DMAC with host y vid (recall that RC has a global view of the network).

In summary, whenever a new VIRO-GENI switch is attached to the network. Firstly, it connects to the RC and LC controller. Consequently, it receives its vid from the RC. Then, it discovers the physically connected neighbor by generating *Neighbor Echo Request & Reply messages*. It uses these

[5] We reuse POX's ARP and DHCP modules

[6] We will use these echo messages in the future for RC failure discovery

packets or PortStatus events to discover the failure of its directly connected nodes. In addition, it builds its routing table by exchange control packets with the others LCs (VIRO's *publish-&-query algorithm*). Lastly, it discovers its attached hosts during the host vid assignment process (*Local_Host* messages).

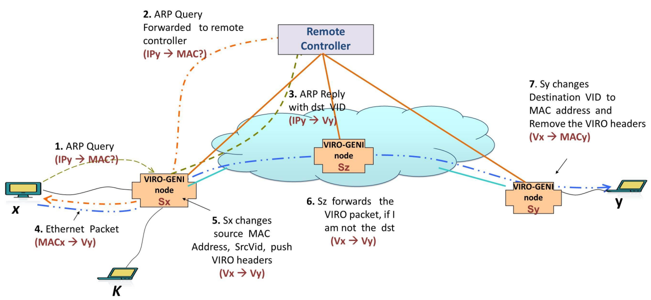
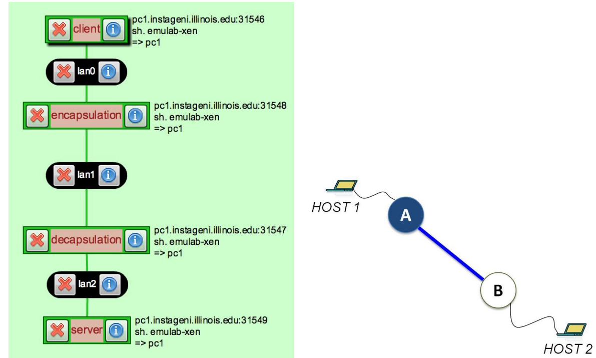


Fig. 5: VIRO packet forwarding between two host machines

G. Packet Forwarding in a VIRO-GENI Network

In this subsection, we explain how the address/vid mapping and packet forwarding is performed in a network composed with VIRO-GENI switches. To achieve this, we use the example illustrated in Figure 5. In this example, host x communicates with host y, using the following steps;

- Host x sends a ARP query to resolve host y IP address.
- VIRO-GENI switch x forwards the ARP query to RC.
- RC returns the ARP reply packet and it replaces the DMAC with the vid of host y, which is composed of switch y vid prepended to host y l-bit identifier (recall that RC has a global view of the network).
- Host x receives the ARP reply and generates the first Ethernet frame, whose DMAC address is host y vid. This frame is forwarded to switch x.
- The Ethernet frame will be received by the source's access node (switch x), and it will generate a miss in the OVS fast-path and slow-path. Then, the frame will be send to VIRO LC, and it will replace the SMAC with the SVID. In addition, it will push the VIRO headers into the Ethernet frame and forward the packet to the next destination, according to its routing table. Lastly, it will add OpenFlow rules to insert the VIRO packet header into packets received from host x and to set the SVID and SHost appropriately. This will cause future packets to be forwarded by the fast path.
- The intermediary VIRO switches (e.g. switch z) will forward the VIRO packets to the next hop, according to their VIRO routing table (this process may include rewriting the FD).
- When the VIRO packet is received by the destination VIRO switch y, it will first generate a miss in the OVS fast and slow path. Then, the packet will be send to VIRO LC. Next, LC will find that it is attached to the destination access node (switch y), by comparing the packet DVID with the access node's vid. Hence, LC will pop the VIRO header and replace the DVID with host y MAC address (recall that LC has local view of all host attached to it). Afterwards, LC will forward the packet to host y. Furthermore, it will add OpenFlow rules to remove the VIRO packet header and rewrite the destination MAC address for subsequent packets.
- All packets between host x and y are transmitted in the VIRO-GENI network using a similar process.
- Packets transmitted between host x and k use the standard Ethernet frame, because both hosts are attached to the same access node VIRO-GENI switch x.



(a) Topology in GENI (b) Network topology

Fig. 6: VIRO packet processing overhead experimental setup

IV. EXPERIMENTS

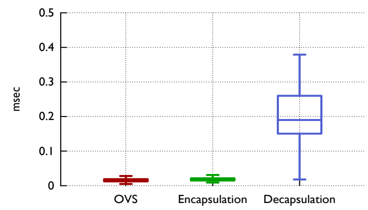
We have conducted a number of experiments to test our initial prototype of VIRO using both Mininet and GENI. In this section, we describe two sets of experiments. In the first experiment, we investigate VIRO's packet encapsulation/decapsulation overhead at edges switches. In the second experiment, we evaluate and compare VIRO's failure recovery mechanism as discussed in Section III-F (*Neighbor Echo Request & Reply* and *Port Status*). 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.

In order to set up and run our VIRO's experiments in GENI, we need to deploy our extend-OVS and VIRO POX controllers (local and remote) to GENI. To achieve this, we first 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 [7]. 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 to reserve the resources for our experiments.

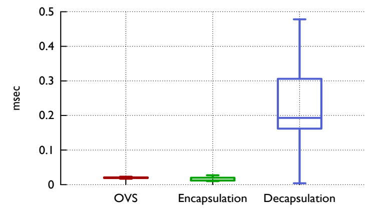
A. VIRO Packet Processing Overhead

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 6). 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

[7]Flack is a flash-based Web interface for viewing and requesting GENI resources. It also provides tools to manage the resources.



(a) GENI



(b) Mininet

Fig. 7: Packet processing delay

messages[8] 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 7(a) and 7(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 7(a) and 7(b)). In the future, we will investigate why the processing time for VIRO packet decapsulation is significantly larger than packet encapsulation.

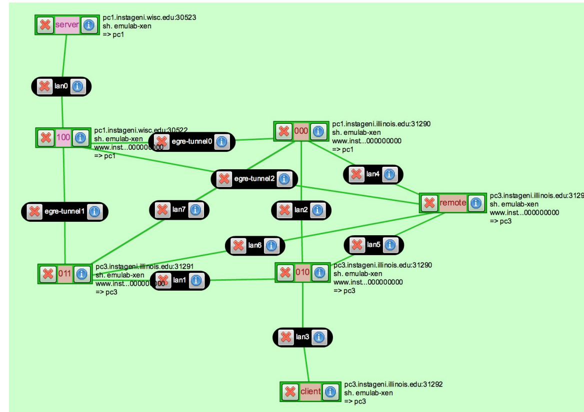
B. VIRO Failure Recovery

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 8(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[9]. We repeat this experiment both in Mininet and GENI.

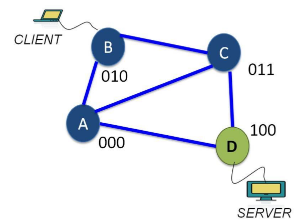
Figure 8(a) shows the deployment of our experiment in GENI. We use 3 PCs, 7 XenVMs and two GENI Aggregate Managers (AMs): Wisconsin and Illinois. The nodes at the

[8] We generate 100 ping request packets

[9] While the client at B is sending traffic to the server at D



(a) Topology in GENI



(b) Network Topology

Fig. 8: VIRO failure recovery experiment setup

same level in VIRO are placed in the same GENI PC, whereas nodes at different level are at distinct GENI PCs. Hence, from Figure 8(a), nodes 010 and 011 are in the same PC. To connect the nodes at different GENI AMs we use EGRE tunnels.

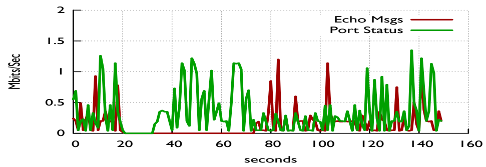
Experiment Discussion: using VIRO's routing protocol, the client at node B sends data packets to the server at node D. Before failure, node B uses its level-3 GW (node C) to communicate with the server. After failure of the link C-D, node C updates its routing table and sends a *GW_Withdraw* message 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 9(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 9(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.

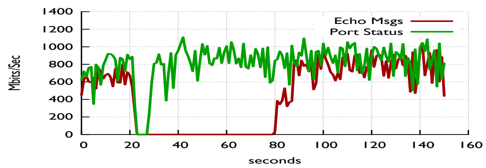
The recovery time for both experiments in GENI and Mininet is significantly large. Hence, in the future we will improve the tuning of our experimental parameters in order to decrease the failure recovery time in the network.

V. Lessons Learned Using GENI and Future Directions

In this paper, we have described our experience in implementing a non-IP protocol – VIRO – in GENI. VIRO



(a) GENI



(b) Mininet

Fig. 9: Failure Recovery

is a “plug-&-play” routing paradigm for future networks. We have developed an initial prototype of VIRO using the OVS-SDN platform. Because the existing OVS-SDN platform is closely tied to the conventional Ethernet/IP/TCP protocol stack, we have modified and extended the OVS (both in the user space and the kernel space) to implement VIRO forwarding functions. We have used POX controllers to build VIRO’s control and management planes. We have carried out experiments to test VIRO failure recovery mechanism and its packet encapsulation/decapsulation overhead at the edges switches.

GENI has incorporated the OVS/SDN software platform running on virtualized machines as part of its support for networking innovations and experiments. When compared with traditional IP routers and Ethernet switches, the SDN paradigm provides a far more flexible framework for controlling and configuring network elements (switches and routers), and therefore allows researchers to develop and experiments with innovative network management services or new applications that require more flexible control of data packets at the “flow” level. However, as the current OVS software and SDN paradigm are closely tied to the existing TCP/IP/Ethernet protocol stack (especially in terms of header fields, matching operations and allowable actions), such constraints make development and experiments of *non-IP protocols* in GENI harder. In the case of VIRO, we are able to *repurpose* the Ethernet/VLAN frame formats to emulate VIRO packet headers. But we have to modify and extend (both the user and kernel spaces of) the OVS software platform to introduce new match-action functions. This requires intimate knowledge of the inner workings of the OVS platform and incurs significant development efforts. On the other hand, we are able to re-use SDN POX controller as is to build VIRO routing and management functions and deploy them in distributed and centralized modes to realize VIRO control/management plane functions. It is possible that not all non-IP protocols that have been – or have yet to be – proposed by the networking research community can be easily retrofitted into the TCP/IP/Ethernet header formats; furthermore, these non-IP protocols may contain data plane functions that cannot be implemented within the “match-action” framework of OVS/SDN. This perhaps calls for more general and powerful data plane abstractions and software platforms to be incorporated in GENI in the future.

With our implementation of VIRO using the extended OVS/SDN software platform, we have successfully deployed our prototype in GENI. In conducting GENI experiments, we find that reserving resources for complex topologies using Flack often requires several attempts which can take long time. Thus, Omni[10] 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. 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. Recently, with the new version of omni (Omni v2.6), creating stitching links have become easier. We will explore the possibility of using stitching links to further improve our implementation in the future. In addition, We find that it is very easy to download and install scripts to the allocated GENI resources, to select the type of links between GENI AMs, to bound VMs to PCs and to create InstaGENI Custom Images, for example. These tools/functionalities make testing and experimenting in GENI easy.

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.

Acknowledgment

The work was supported in part by a Raytheon/NSF subcontract 9500012169/CNS-1346688, the NSF grants CNS-1017092, CNS-1117536 and CRI-1305237, and the DTRA grant HDTRA1- 09-1-0050.

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] M. Caesar, T. Condie, J. Kannan, K. Lakshminarayanan, and I. Stoica, “ROFL: Routing on Flat Labels”, in Porc. of SIGCOMM, 2006.

[5] B. Ford, “Unmanaged Internet Protocol: Taming the Edge Network Management Crisis”, SIGCOMM CCR, vol. 34, 2004.

[6] C. Kim, M. Caesar, and J. Rexford, “Floodless in Seattle: A Scalable Ethernet Architecture for Large Enterprises”, in Proc. of SIGCOMM, 2008.

[7] P. Maymounkov and D. Mazieres, “Kademlia: A Peer-to-Peer Information System Based on the XOR Metric”, in Proc. of IPTPS, 2002.

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

[9] D. Levin, A. Wundsam, B. Heller, N. Handigol, and A. Feldmann, “Logically centralized?: State Distribution Trade-offs in Software Defined Networks”, in Proc HotSDN, 2012.

[10] A. Tootoonchian and Y. Ganjali, “Hyperflow: A Distributed Control Plane for OpenFlow”, in Proc INM/WRE, 2010.

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