TECHNICAL REPORT

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

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

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

October 2014

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.

1 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 [12][13][6][8] 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.

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 [31][17]. 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) [3], 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 [12]. 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 [12]. 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 a nutshell, VIRO is designed with two broad sets of goals: i) to support with minimal manual configuration (future) large, dynamic networks which connect tens or hundreds of thousands of diverse devices with rich physical topologies; and ii) to meet the high availability, robustness, mobility, manageability and security requirements of these networks and the services running on top of them. These goals are motivated partly by the rise of huge data centers, emergence of cloud computing and services, Internet of Things (IoT) as well as the continued trends in large campus, enterprise and ISP (wired, wireless and cellular data) networks to use 1/10/100 Gigabit Ethernet as the core (layer-2) networking technology.

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 researchers. To our knowledge, we are the first to deploy and test a non-IP routing protocol in GENI. Furthermore, we take advantage of the built-in fast rerouting and load balancing capabilities of VIRO to propose a novel in-network pathlet switching framework for software-defined networks that fully exploit the path diversity available in the network. The contributions of this paper are as follows:

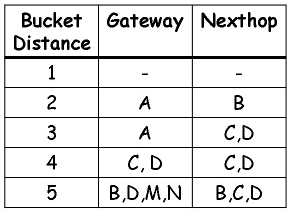
- We implement and deploy an initial prototype of VIRO in GENI using the SDN platform.
- We perform experiments to evaluate VIRO packet's encapsulation/decapsulation overhead and our failure recovery mechanisms. In addition, we carry out experiments to evaluate VIRO supporting for host mobility and forwarding of VIRO frames with GENI stitching. The results of these experiments will help us to improve and extend our VIRO prototype.
- We propose a novel in-network dynamic pathlet switching framework with VIRO for software-defined networks
- 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 2 provides an overview of VIRO. Section 3 discusses our implementation and deployment of VIRO in GENI. In Section 4 we present our in-network pathlet switching framework with VIRO for SDN networks. In Section 5 we present our new routing paradigm – dubbed *routing via preorders* – which circumvents the limitations of conventional path-based routing schemes. Section 6 presents our experiments and discusses our experimental results. We conclude and discuss future work in Section 7.

2 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 [12].

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 [12]. VIRO's structured vid space embeds the physical network topology formed by the connections among physical network



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

Figure 1: VIRO routing table for node A.

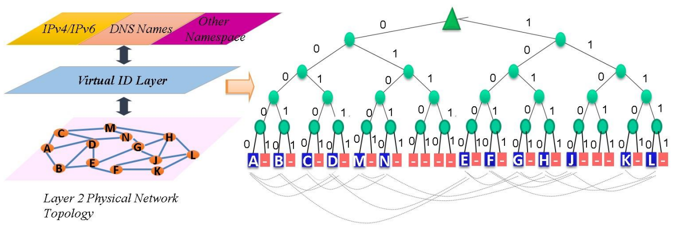


Figure 2: vid space as a virtual binary tree: the grey dotted lines denote physical connectivity and the red boxes represent the unused vid's

components. Such embedding is illustrated in Figure 2 using a Kademlia-like [21] *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:

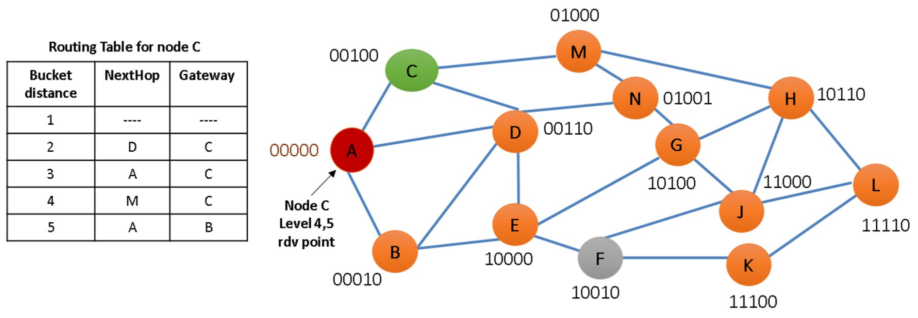
Vid space construction and vid assignment: at the network bootstrapping phase, the topology-aware, structured vid space is constructed. VIRO uses a Kademlia-like *virtual binary tree* to structure the vid space, where each VIRO switch (a leaf node of the tree) is assigned an L-bit string *vid* corresponding 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 gateway information to reach specific level of the virtual binary tree to *rendezvous points* (rdv). In VIRO, a *gateway* (GW) is a node that has a direct (physical) edge to a node in a neighboring subtree of the same level in the vid space. The rdv nodes store GWs information to reach specific levels in the vid space. The connectivity information stored at rdv is a pair of `[level, gateways list]` to reach a sub-tree in the vid space. The set of all routing information stored at any rdv is called *rendezvous store*. Each rdv maintains the list of nodes using the gateways in its store: $\{(GW_x : nodex, nodey); (GW_y : nodek, nodez); ...\}$.

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 vids 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. Routing in VIRO is done based on destination vid and GWs information. For example, suppose node C, $vid(C) = 00100$, wants to send a data packet to node F, $vid(F) = 10010$ (see Fig. 3). According to VIRO routing protocol, node C will compute the *logical* distance between its vid and F's vid, namely, L - length of the longest common prefix between them, which is 5. Assuming its level 5 routing table is empty, node C will then query its level-5 rdv point (node A) for a level-5 GW. Next, node A will return node B (which is directly connected to E in a neighboring level-5 subtree) as node's C level-5 GW. Afterwards, node C will look up its routing table for a nexthop (a directly connected neighbor, in this case node A) to reach B for packets destined to F. For more detail about VIRO forwarding operations, the reader is referred to [12].

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


Routing Table for node C

Bucket distance	NextHop	Gateway
1	----	----
2	D	C
3	A	C
4	M	C
5	A	B

Figure 3: VIRO routing tables and rendezvous point

3 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.

3.1 Implementation Challenges

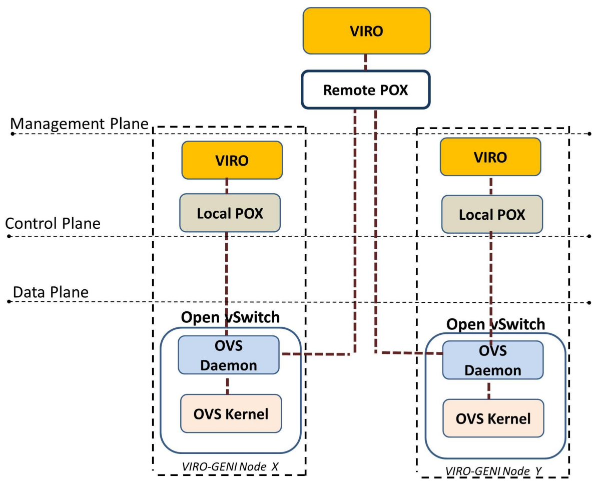


Figure 4: Software Stack in VIRO-GENI Node

The OVS software platform is derived from the OpenFlow switch specifications and SDN paradigm. When compared to 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 [12] 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.

3.2 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 4. 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 4. 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.

3.3 Data Plane

In this subsection, we describe the data-plane implementation in VIRO-GENI node. First, we discuss how we repurpose the standard Ethernet frame to represent a VIRO frame, and we conclude by discussing the modifications we make in OVS in order to forward VIRO frames.

3.3.1 VIRO Frame

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 [23], see Figure 5. 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)[12]. The remaining bytes in the VIRO frame are used to encapsulate the EtherType and the payload of the original Ethernet frame [2]. In the original Ethernet frame, we add a new EtherType 0x0803 for VIRO control packets (see Section 3.4.3). 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 [23].

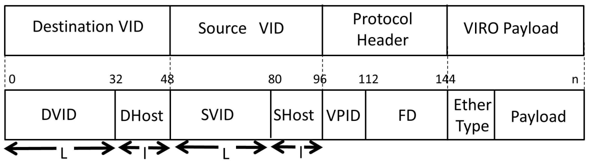


Figure 5: VIRO Frame

3.3.2 OVS Extensions

The Open vSwitch implementation consists of two components: a kernel (fast path) and a userspace (slow path). The kernel implements the forwarding engine responsible for per-packet lookup, modification and forwarding. In addition, it maintains counters for each forwarding table entry[28]. However, the majority of the OVS functionality is implemented within the user space. The main component in the user space is the "ovs-vswitchd" module. It communicates with kernel module over netlink and with outside world using OpenFlow. This module is responsible

[2]They form the payload of the VIRO frame

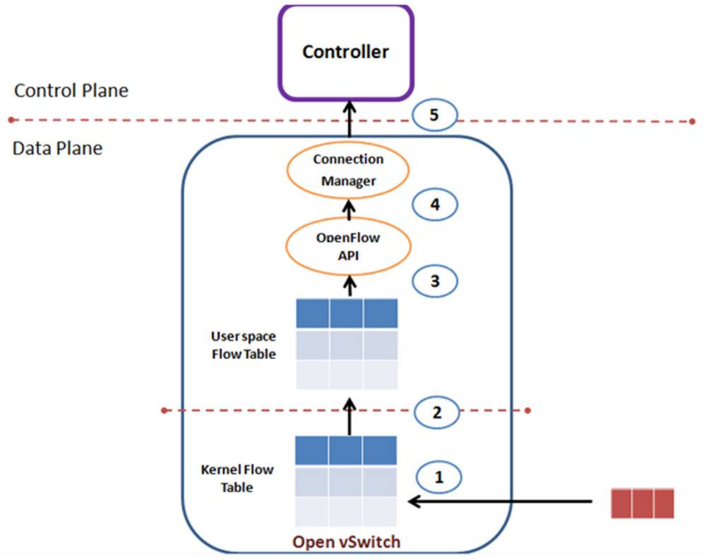


Figure 6: Packet processing in OpenVswitch

for reading the openFlow configuration from ovsdb-server[3][29]. Its packet classifier supports efficient flow lookup with wildcards and checks datapath flow counters to handle flow expiration and statistics requests[29].

When a packet arrive to an OVS, it is first processed by the fast path. In the Kernel, the packet header fields are extracted. Then, these header fields are hashed and used as an index into a set of large hash tables. If an entry is found, the actions corresponding to this entry are applied to the packet and OVS counters are updated. Otherwise, the packet is sent to the userspace and the OVS miss counter is incremented. In the slow path, when a packet is received from Kernel, it is given to the classifier to look for matching flows in the flow tables. If there is a table-miss the OpenFlow API calls the connection manager to encapsulate the packet in a Packet_In message and send it out to the SDN controller attached to the switch. When the controller receives the Packet_In message, one or more applications running on the controller may process the message and install rules in the openFlow table in the switch via a Flow_Mod message, so that future packets can be processed on the switch [22]. Figure 6 illustrates this process (Steps 1-5).

The current OpenFlow matching operations, header fields and allowable actions are still tied to the Ethernet/IP/TCP protocol stack. On the other hand, VIRO has its unique routing and forwarding behaviour. Thus, VIRO's forwarding cannot be directly using the standard "match-action" of OpenFlow. Therefore, to forward our VIRO frame in the OVS data-path, we modify and extend match/actions 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 1 shows the list of new actions, we have added in both fast and slow paths:

[3]Database that holds switch-level configuration

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 1: 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.

Hence, for a VIRO switch that runs our extended version of OVS, there are typically three scenarios when a packet needs to be forwarded:

- **Case 1:** If a normal ethernet packet enters into a VIRO network, we need to encapsulate the ethernet packet into a VIRO frame and forward it to the nexthop VIRO router. To transform it to a VIRO frame, we need to first rewrite the source MAC address to its corresponding source VID, and push FD and VIRO ethertype to the frame. Then we lookup the routing table and forward the packet to the corresponding port according to destination VID. Figure 7 shows the actions that are taken inside the egde router.

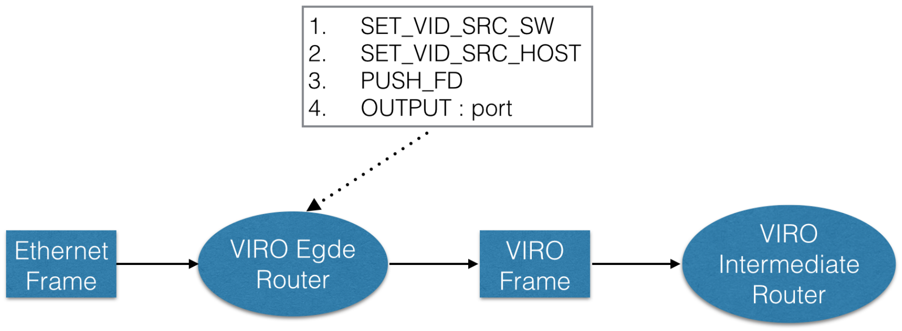


Figure 7: Encapsulation: From Ethernet Frame to VIRO Frame

- **Case 2:** If a VIRO packet is about to leave the VIRO network, we need to decapsulate the VIRO packet into an ethernet packet and forward it to the outside netowrk. To transform it to an ethernet frame, we need to first rewrite the destination VID to destination MAC address, and pop out FD and ethertype from the frame. Then we forward the packet to the corresponding host that is attached to this edge router. Figure 8 shows the actions that are taken inside the egde router.

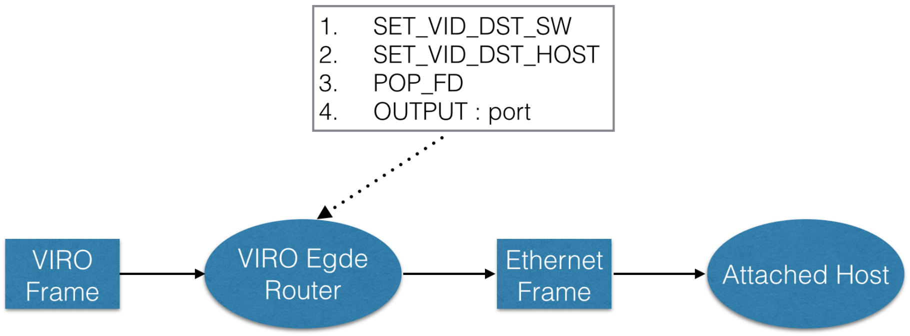


Figure 8: Decapsulation: From VIRO Frame to Ethernet Frame

- **Case 3:** If source host and destination host are attached to a same VIRO router, the router only simply need to forward the ethernet frame from one host to another. Figure 9 shows the actions that are taken inside the egde router.

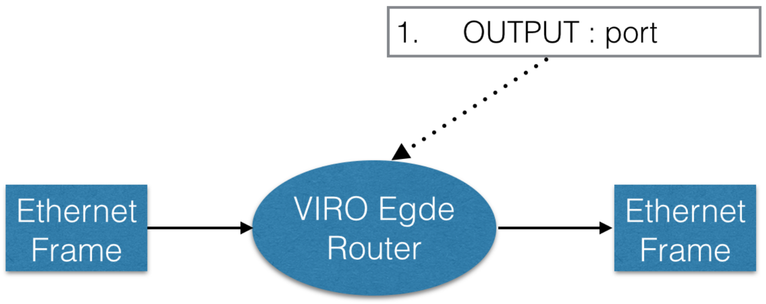


Figure 9: Directly Forward Ethernet Frame from One Host to Another

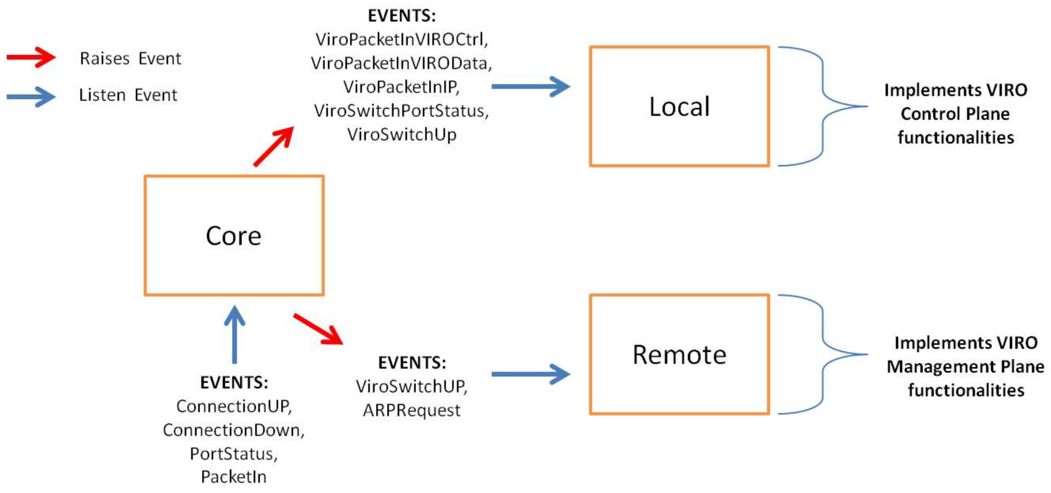


Figure 10: Main components used to build VIRO control and management planes

3.4 VIRO POX Controller

In this subsection, we describe how we implement our control and management planes. Firstly, we describe the types of events we implement to be handle by those modules. Then, we discuss our core, local and remote modules, the keys pieces used to construct our VIRO control and management planes. Figure 10 illustrates how these modules are combined together to implement VIRO routing protocol.

3.4.1 Events

We create the following list of events[4]:

- ViroSwitchUp: raised when a switch connects to a controller.
- ViroSwitchDown: raised when a switch disconnects from a controller.
- ViroSwitchPortStatus: raised when the controller receives an OpenFlow port-status message.
- ViroPacketInIP: raised when the controller receives an IPv4 packet.

[4]This events are handled by our remote and local POX modules

- ViroPacketInIPV6: raised when the controller receives an IPv6 packet.
- ViroPacketInARP: raised when the controller receives an ARP packet.
- ViroPacketInLLDP: raised when the controller receives an LLDP packet.
- ViroPacketInVIROData: raised when the controller receives a VIRO data packet.
- ViroPacketInVIROCtrl: raised when the controller receives a VIRO control packet.

3.4.2 Core Module

This module listens and handles the following OpenFlow events: ConnectionUp, ConnectionDown, PacketIn and PortStatus:

- ConnectionUp Event: upon receiving this event, the core module raises a ViroSwitchUp event.
- ConnectionDown Event: upon receiving this event, the core module raises a ViroSwitchDown event.
- PortStatus Event: upon receiving this event, the core module raises a ViroSwitchPortStatus.
- PacketIn Event: upon receiving a packetIn event, the core modules raises a event based on the packet type, e.g. IP, VIRO, ARP, etc.

3.4.3 Local Module

This module implements the VIRO LC functionalities running on each node. 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). This module listen and handles the following events raised by the core module:

ViroPacketInVIROCtrl Event: upon receiving this event, the LC processes the VIRO control packets accordingly. The types of control packets 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 [12] and to Section 3.6.

ViroPacketInVIROData Event: upon receiving a VIRO data packet the LC compares the packet destination vid to the vid of the switch attached to it. If both vids are the same, the LC converts the VIRO frame into an standard Ethernet frame by replacing the DMAC by the DVID, removes the VIRO protocol header and route the generated Ethernet frame (packet decapsulation) to the destination host. Otherwise, it routes the VIRO frame to the next hop based on its VIRO table. Lastly, RC adds a OpenFlow rule for subsequent packets.

ViroPacketInIP Event: upon receiving an IP packet, the LC convert the standard Ethernet Frame into a VIRO frame, by replacing the SMAC with the SVID and pushing the VIRO headers (packet encapsulation). Furthermore, it routes the generated VIRO frame to the next hop.

ViroSwitchPortStatus Event: upon receiving a VIRO port status event, the LC checks the type of event. If it is equal to "modified", it look up its local topology table to find if the modified port is attached to a host machine. If the port is attached to a host, the LC removes the specific host from its topology table. Furthermore, it notifies the RC that the host was disconnected from the switch [5].

[5]This functionality is still being implemented

ViroSwitchUp Event: upon receiving this event, the LC adds OpenFlow rules to the switch to receive all packets miss of type VIRO and IP packets. In addition, it sends its controller ID to the switch.

In summary, the VIRO module attached to LC running on each node is responsible handling VIRO control packets used in VIRO routing protocol, to encapsulate/decapsulate IP packets from/to host machines and to handle packet miss from the data-plane (see Section 3.6 and 3.7).

3.5 Remote Module

This module implements the VIRO RC functionalities running on each node. It listens and handles the following events raised by the core, arpd, dhcpd modules [6]: ViroSwitchUp, ARPRequest, DHCPLease.

ViroSwitchUp Event: upon receiving a VIRO switch up event, the RC sends its Id to the switch. Additionally, it assigns the switch vid and add the respective switch to its topology table, see Section 3.6. for more details.

ARPRequest Event: upon receiving this event, the RC controller performs the mapping IP=VID and replaces the DMAC with the DVID in the ARP reply.

DHCPLease Event: upon receiving this event, the RC finds from its topology table the switch attached to the host issuing the DHCP request. Then, it assigns the host vid, which is composed of switch vid prepended to host l-bit identifier.

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. By listening to the events described above, 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 and IP/VID/MAC/PORT mapping (Global view). In Section 3.6 we discuss in details the RC's functions.

3.6 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 [7]. 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. More precisely, the VIRO LC listen for *Port Status* events, and upon receiving such event checks if the event type is equal to "modified". If so, the LC finds if

[6]We reuse POX's ARP and DHCP modules
[7]We will use these echo messages in the future for RC failure discovery

the modified port is attached to a host by looking up its topology table. If a host is associated with this port, the LC removes the specific host attached to the modified port from its local topology table. Furthermore, it notifies the remote controller that the host was disconnected from the switch. The RC uses this information to update its topology table in order to keep a consistent global view of the network [8]. VIRO handles nodes failures without resorting to flood of failure notification (as used in OSPF), instead, it utilizes a *withdraw and update* mechanism [12].

Routing Table Construction: the local VIRO module in each VIRO-GENI node builds its routing tables using the *publish-&-query* algorithm described in [12] (*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 [23].

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[12], 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).

Network topology discovery/maintenance: the RC has a global view of the network. It maintains a topology table with the information of all the switches attached to the network, and their list of attached hosts. Recall that the RC discovery the switches in the network during "ViroSwitchUp" events and the list of hosts attached to a switch during DHCP leases. However, to maintain a consistent view of the network, RC relies of notifications from the LC whenever a host disconnects from a switch. In addition, during an ARP request, the RC compares the vid of the switch attached to the host issuing the ARP request with switch's vid for this respective host from its topology table. If both informations are not consistent, the RC removes the obsolete entry from its topology table and assigns a new vid to the respective host. Lastly, it sends a *local_host* message to the respective LC with the address mapping for this host [9]. In the future, we will use "connection down" event to find the switches disconnected from the network.

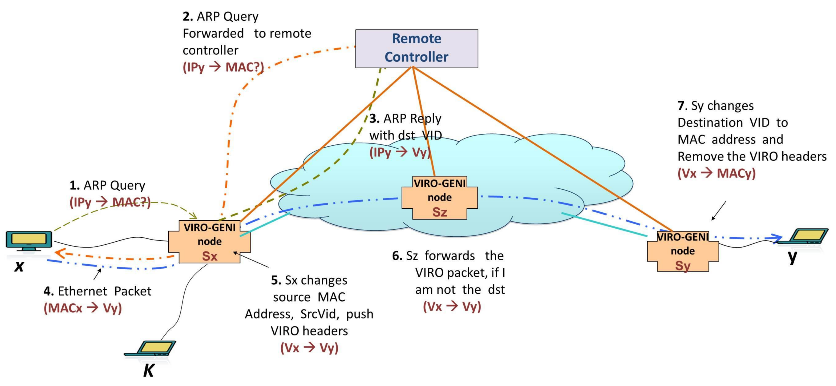
In summary, whenever a new VIRO-GENI switch is attached to the network. Firstly, it connects to the RC and LC controller. Consequently, it receives its vid from the RC. Then, it discovers the physically connected neighbor by generating *Neighbor Echo Request & Reply messages.* It uses these packets or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

3.7 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;

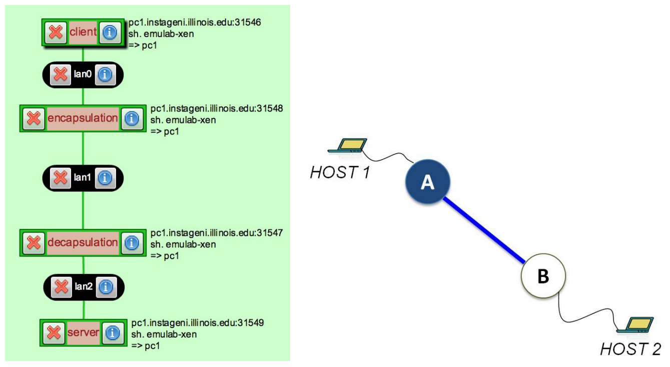
- Host x sends a ARP query to resolve host y IP address.
- VIRO-GENI switch x forwards the ARP query to RC.
- RC returns the ARP reply packet and it replaces the DMAC with the vid of host y, which is composed of switch y vid prepended to host y l-bit identifier (recall that RC has a global view of the 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.

[8]This functionality is still being implemented
[9]This functionality is still being implemented



2. ARP Query Forwarded to remote controller (IPy → MAC?)
Remote Controller
3. ARP Reply with dst VID (IPy → Vy)
7. Sy changes Destination VID to MAC address and Remove the VIRO headers (Vx → MACy)
1. ARP Query (IPy → MAC?)
VIRO-GENI node Sz
VIRO-GENI node Sx
VIRO-GENI node Sy
x
y
4. Ethernet Packet (MACx → Vy)
5. Sx changes source MAC Address, SrcVid, push VIRO headers (Vx → Vy)
6. Sz forwards the VIRO packet, if I am not the dst (Vx → Vy)
K

Figure 11: VIRO packet forwarding between two host machines



client
pc1.instageni.illinois.edu:31546 sh. emulab-xen => pc1
lan0
encapsulation
pc1.instageni.illinois.edu:31548 sh. emulab-xen => pc1
lan1
decapsulation
pc1.instageni.illinois.edu:31547 sh. emulab-xen => pc1
lan2
server
pc1.instageni.illinois.edu:31549 sh. emulab-xen => pc1
HOST 1
A
B
HOST 2

(a) Topology in GENI (b) Network topology

Figure 12: VIRO packet processing overhead experimental setup

- 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.

4 IN-NETWORK DYNAMIC PATHLET SWITCHING

The current best-effort IP protocol cannot readily provide the bandwidth and other service guarantees that many applications such as video streaming require today. End system based path switching and load balancing across multiple paths have been proposed as an alternative approach to meet the bandwidth requirements of today's applications. These solutions require end systems to be *multi-homed* so as to exploit the path diversity in today's network. In this section, we propose a novel *in-network pathlet switching framework* for software-defined networks using the Virtual Id routing Protocol (VIRO). In our framework, we take advantage of the *built-in* fast re-routing and load balancing capabilities of VIRO to perform dynamic pathlet selection, switching and load balancing to fully exploit the path diversity available in the network.

In the next subsections, we discuss in detail our in-network dynamic pathlet switching framework using VIRO, where we have extended the functionalities of VIRO's Rendezvous nodes, Local and Remoter controllers to support pathlet switching (see Figure 14).

4.1 Overview

In-network pathlet switching is a mechanism that allows network devices (e.g. routers, switches) to dynamically switch among several paths to a destination based on their performance. To achieve in-network pathlet switching, two conditions must be met. Firstly, it is necessary to obtain performance information of the current path as well as alternative paths in the network. Secondly, it is necessary to have a mechanism and/or component responsible for making the path switching decision inside the network, when the performance of the current path degrades significantly.

In this work, we propose an in-network dynamic pathlet switching system by taking advantage of the VIRO routing protocol and the SDN paradigm. We use VIRO's routing capability and SDN switch's statistics to make local and global path switching decisions. We collect information about "latency and throughput" and use VIRO enabled SDN controllers to make path switching decisions inside the network. The latency information is collected by adding additional mechanism to measure the latency to gateway per node. Furthermore, we use the statistics reported by OVS to get the throughput information per gateway approach or flow throughput. Next, we discuss the functions of each component used in our in-network dynamic pathlet switching system: *local controller, remote controller and rendezvous point.*

4.2 Local Controller

The VIRO LC in our VIRO-GENI node will be responsible to estimate per gateway throughput and to monitor the latency to others gateways from a node point of view. To compute the latency, LC will periodically send probes packets with unique identifier (ID) to each available gateway. Upon, receiving a probe packet, a gateway node will forward this packet back to the sender. Then, each VIRO LC will record the time at which it sends and receives probes messages. These values, alongside the probe ID in each probe packet, will be used to estimate a node latency to a gateway. In addition, a LC attached to a gateway (GW) node will use the OVS statistics to compute the GW's throughput. The LC will periodically report these values [10] to RC.

4.3 Remote Controller

VIRO RC maintains a full view of the network topology (hosts, switches and links). It receives the list of GWs from the rvd points in the network and maintains a mapping of rdv nodes and their respective list of GWs. In addition, it also receives messages with GW's throughput information from LCs attached to GWs nodes. Using this information, the RC periodically notifies the rdv points about the status of their gateways. Furthermore, the RC can query nodes, in the network, for information about their latency to any gateway, as well as, provide information about the quality of any path in the network.

4.4 Rendezvous Point

Recall from Section 2 that each VIRO node publishes and queries routing information to *rendezvous* nodes(rdv), in order to build its VIRO's routing table. The rdv is one of the components used to implement our in-network

[10] throughput and delay

pathlet switching. In this framework, the rdv node periodically reports to RC its list of GWs (rdv store), and it receives notifications from the RC about the status of its GWs. Then, when rdv receives a rdv query message, it will also include information about the GW's throughput for the entire list of GWs returned in the rdv reply message. In addition, the rdv periodically sends GW's throughput information to the nodes using any GWs in its rdv store [11].

The advantage of this framework is that path-switching decisions can be made at different levels in the network. For example, the LC can use the information about the measured latency to all its gateways, along with the information about the GW's throughput, to initialize path switching inside the network. Furthermore, it can also coordinate with the RC in order to make path switching decisions in the network. Similarly, the rdv point, based on the VIRO routing protocol, receives GW failure notifications for the list of GWs in its rdv store. Hence, when any of its GWs fails, rdv can notify the others nodes using the "failed GWs" to switch to a different GW, thereby initializing path switching in the network. Lastly, the rdv can also coordinate with the RC in order to make path switching decisions for all flows in the network. In the future, we will develop algorithms that take advantage of the different levels of decisions for path switching in our proposed framework.

4.5 In-Network Pathlet Switching End-to-End Example

In this subsection, we explain how in-network path switching is performed in a network composed with VIRO-GENI switches. To achieve this, we use the example illustrated in Figure 13. In this example, we use the Abilene network [1] topology and we assign VIRO vids to each node – for more on VIRO vid assignment the reader is referred to [12]. In Figure 13 each node represents a VIRO-GENI switch. Hence, they have both local (LC) and remote (RC) VIRO. SDN controllers, but for simplification we are omitting the controllers illustration in that Figure.

In Fig. 13, 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. 13). The LC, RC and rdv points will exchange messages, as discussed in Section 4.4, 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 Section 4.2 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[12] as its new level-3 GW. Then, the new path for the packets from client to server will be the following (*pathlet-switching*): $Seattle \rightarrow Denver \rightarrow KansasCity \rightarrow Indianapolis \rightarrow Chicago \rightarrow NewYork$. Using this mechanism, we can potential discover and react to network performance degradation faster than the traditional methods used at the end-points.

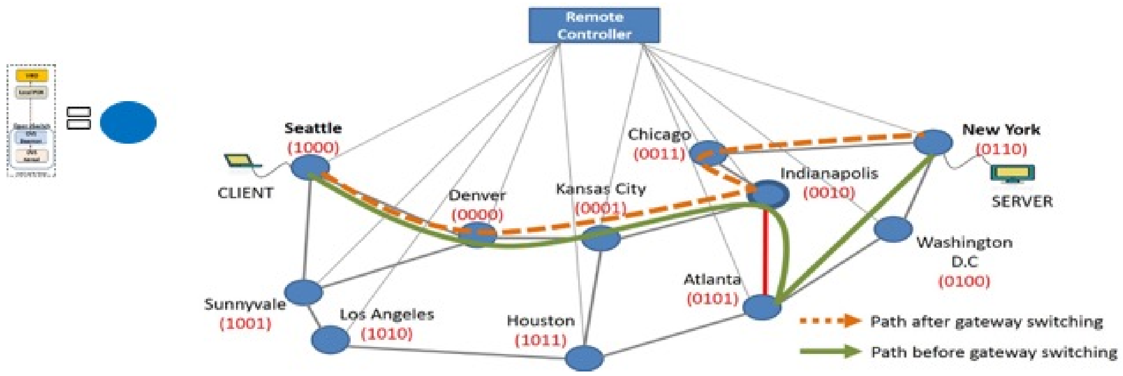


Figure 13: In-network path switching with VIRO

[11]recall that rdv maintains a mapping of gateway and the list of nodes using them
[12]a level-3 GW in the list of GWs received from the rdv point

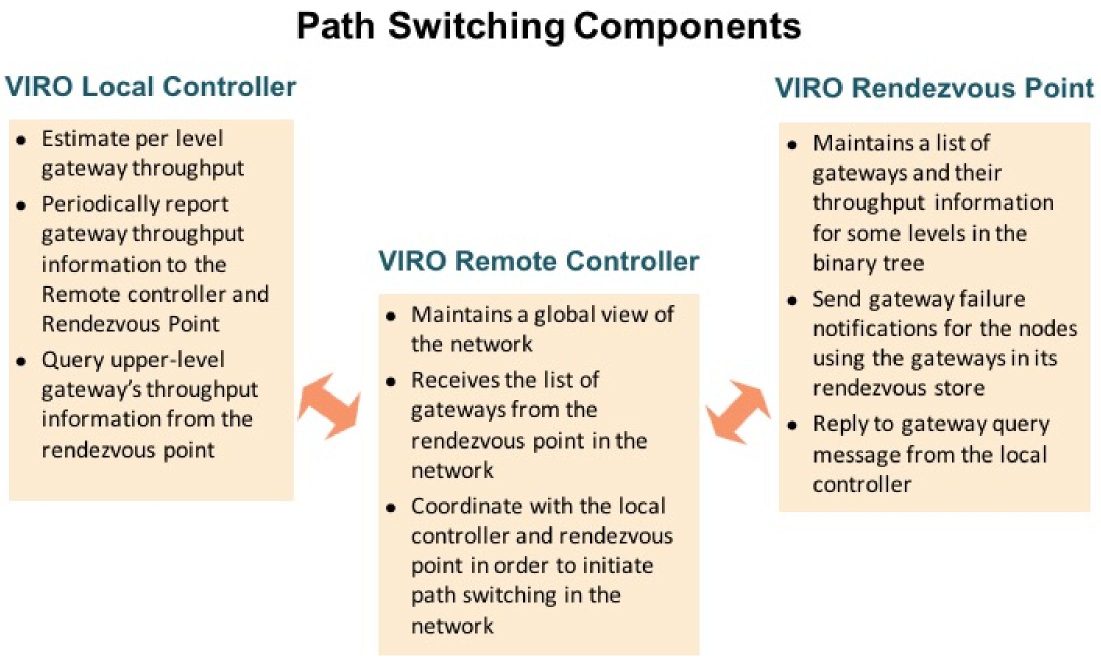

Path Switching Components
VIRO Local Controller
• Estimate per level gateway throughput
• Periodically report gateway throughput information to the Remote controller and Rendezvous Point
• Query upper-level gateway's throughput information from the rendezvous point
VIRO Remote Controller
• Maintains a global view of the network
• Receives the list of gateways from the rendezvous point in the network
• Coordinate with the local controller and rendezvous point in order to initiate path switching in the network
VIRO Rendezvous Point
• Maintains a list of gateways and their throughput information for some levels in the binary tree
• Send gateway failure notifications for the nodes using the gateways in its rendezvous store
• Reply to gateway query message from the local controller

Figure 14: In-network dynamic pathlet switching framework: path switching components

5 Adaptive Resilient Routing via Preorders in SDN

In this section, 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. Next, we discuss in details our novel routing paradigm.

5.1 Limitations of Path-Based Routing

We first use a simple toy example to illustrate the fundamental limitations of path-based routing. Consider two *edge-disjoint* paths P_1 (the upper green path) and P_2 (the lower red path) from $s = v_1$ to $d = v_{2L+1}$ in Figure 15, both are of length $2L$. Let U denote the collection of nodes on P_1 and P_2. We see that there are in fact many paths of length $2L$ (in fact, $2^L!$) from s to d in U. To exploit topological diversity, the *rigidity* of a path as a sequence of nodes and links requires us to enumerate and *pre-specify* all paths from s to d in U for path-based routing. As the network size increases, the number of paths from s to d can grow exponentially in the worst case – the *combinatorial curse* [13]. Perhaps more importantly, a path is a very *fragile* object in that if any link $l = (v_i, v_{i+1})$ or a node v_i along a path fails, the path ceases to exist. For example, using P_1 as a primary path and P_2 as a backup path in Figure 15, any two-link failure event with one from P_1 and the other from P_2 would render both paths invalid. This is despite the fact that as long as the two failed links are not incident at the same node, there always exists another path (other than P_1 and P_2) from s to d. Only in the case that both links incident on a node v_{2k+1} fail (or node v_{2k+1} itself fails), s cannot reach d as the network is partitioned.

Conventional routing schemes are primarily path-based, thus suffer from the fundamental limitations discussed above. Classical IP routing protocols such as RIP and OSPF employ a single shortest-path. In addition, multipath routing has been proposed (see [19, 20, 32]), where multiple paths are established between a source-destination pair. Multipath routing can help improve latency and network resilience. However, it is still path-based and thus suffers from the same limitations of path-based routing mentioned above; it also cannot exploit all possible paths between

[13]When not restricted to shortest paths, in a network with *rich* topology (i.e a dense graph where $m = |E| \gg n = |V|$) the number of paths can in fact grow in the order of $O(n!)$.

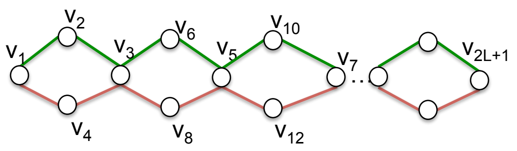
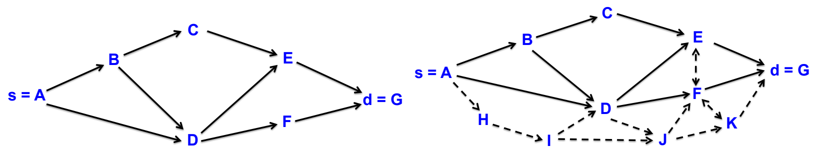


Figure 15: Toy Example



(a) Primary PrOG (b) Backup PrOG

Figure 16: Example of Routing Preorders and their associated PrOGs

a source-destination pair, apart from explicitly enumerating and setting all of them up for multipath routing. While not constrained to shortest paths, MPLS relies on label-switched paths for both primary and protection routing. With IP destination-based forwarding, part of the challenges in designing proactive fast rerouting schemes lies in ensuring consistency between primary and backup paths to avoid *forwarding loops* under failures. This is achieved in Failure Insensitive Routing (FIR) [26, 16, 25] by resorting to *interface-specific forwarding* (ISF), which provides the first provably correct IP Fast Rerouting algorithm for handling *single* link (or node) failures. Various IP fast rerouting schemes (see [14, 27, 33] and references therein) have since been proposed, none of which are resilient against arbitrary multiple link/node failures. As the number of possible (concurrent) link failures increases – thus also the number of possible failure combinations, the *combinatorial* nature of the problem renders path-based routing approaches less effectual, if not infeasible (see [7] for an interesting *negative* result regarding the infeasibility of resilient routing under ISF for protecting against arbitrary k link failures when $k > 1$).

Other alternative resilient routing schemes have also been proposed, e.g via failure-carrying packets [24, 15] where a new path is computed *on the fly* at each node when encountering failures. The classical *link reversal* algorithm [10] is the first non-path-based routing scheme, where a new *directed acyclic graph* (DAG) is constructed in a distributed fashion when encountering a failure, and is provably resilient against arbitrary k link failures in that reachability between any two nodes is guaranteed unless the network is partitioned. Dynamic link reversal mechanisms are adopted in [18] to ensure data plane connectivity under arbitrary link failures. The main drawback of link reversal mechanisms is that in the worst case it takes $O(n^2)$ steps to converge. The authors in [5] cleverly implement a graph search algorithm (e.g a depth first search) via pre-installed "match-action" rules in SDN switches, and a dynamically adjusted "tag" carried in packets to achieve resiliency against arbitrary link failures. The downside of this scheme is that under failures all packets may take up to $O(n)$ steps in the worst case to reach the destination, as they "walk around" (e.g in a depth first search manner) the network to search for an available path. Neither [18] nor [5] takes into account the latency constraint when exploring alternative paths. Given that most applications run on TCP, we believe that bounding the latency of packets (both under the normal and failure scenarios) is important, as long delayed packets will likely lead to time-outs, triggering unnecessary packet retransmission; they will further reduce the throughput of applications. Lastly, we would like to point out that various diverse and resilient routing schemes (e.g [30, 4, 11]) have been "customer-designed" for data center networks, which often only work for specific types of network topologies (e.g Fat-Tree or Leaf-Spine networks).

5.2 Resilient Routing via Preorders

Basic Notions and Illustration. Let $G_{s\to d} = (U_{s\to d}, \Xi_{s\to d})$ be a *connected*[14] subgraph of G, where $s, d \in U_{s\to d} \subseteq V$. We will orient the edges of $G_{s\to d}$ to create a special *preorder* on U, referred to as a *routing preorder* ($\Gamma_{s\to d}$) from s to d .

Mathematically, a *preorder* $\lesssim$ on a node set U is a binary relation that is *reflective* and *transitive*[15]. For any

[14]Namely, any node in $U_{s\to d}$ can reach any other node in $U_{s\to d}$ without traversing any node outside $U_{s\to d}$. In particular, $G_{s\to d}$ is $s - d$ *connected.* For conciseness, we drop $s \to d$ from $U_{s\to d}$ when the context is clear.

[15]We note that if $\lesssim$ is *antisymmetric*, it yields a *partial order*, denoted by $\prec$ on U. $(U, \lesssim)$ is called a *preordered set* or *proset*, and

$u, v \in U$, if $v \lesssim u$, we say v is a predecessor of u, and u a successor of v; in addition, v is a *child* of u, and u a *parent* of v, if $v \lesssim u$ but $u \not\lesssim v$ (we denote this relation by $v \lnsim u$), and $\nexists\, w \in U$ such that $v \lnsim w \lnsim u$. Given $S \subseteq U$, $w \in U$ is called an *upper bound* of S, if $\forall\, v \in S$, $v \lesssim w$, and it is a *least upper bound* of S, if for any upper bound w' of S, $w \lesssim w'$. The (greatest) lower bounds of S can be similarly defined. We say a preorder is a *(routing) preorder* from s to d (and $U_{s\to d}$ a *routing cover*), if the following conditions are met: i) s is the *unique* greatest lower bound of U, and d is the *unique* least upper bound of U; and ii) for any $u \in U$, $s \lesssim u \lesssim d$.

Figure 16 shows an example of a routing preorder from s to d, where $v \to u$ indicates v is a child of u, i.e., $v \lnsim u$; and $v \leftrightarrow u$ indicates $u \lesssim v$ and $v \lesssim u$ (in this case we refer to u and v as siblings). Intuitively, the condition ii) above implies that, for any $u \in U$, there is a *directed* path (or a *chain*) from s to u and from u to d. We call the *directed graph* induced by a (routing) preorder as a *preordered graph* or *PrOG* in short. As bi-directed edges are allowed in *PrOG*, it is in general *not* a directed acyclic graph (DAG).

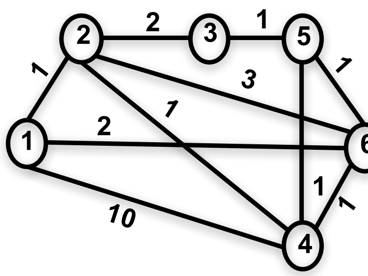
Given a flow F from s to d, instead of mapping it to a single path or multiple paths for routing, we select a routing cover $U_{s\to d}$ and construct a preorder $\lesssim_{s\to d}$ for routing packets of F from s to d: at any node $u(\neq d) \in U_{s\to d}$, packets of F can be proportionally routed to either one of its parents and may also to one of its siblings. In other words, packets are dynamically routed along all feasible paths contained in the PrOG by appropriately merging and splitting traffic at each node, thus utilizing all relevant links, *without the need to enumerate and specify all paths.* With an appropriately chosen PrOG, routing via preorders has *built-in resiliency.* Consider again the toy example in Figure 15 where the routing PrOG from $s = v_1$ to $d = v_{2L+1}$ is defined by orienting all edges from left to right. In contrast to path-based routing (say, with two equal-cost, edge-disjoint paths), routing via preorders, using this PrOG, is resilient against arbitrary link failures as long as they do not partition the network. In fact, we argue that it is possible to construct PrOGs in such a manner that routing via preorders attains the *(latency-constrained) optimal resiliency* as we will expound on further below.

Preorder Selection for Optimal Resilient Routing. Clearly, what preorder (or PrOG) is selected for routing a flow determines its resiliency. In general, with more nodes and edges included in the routing PrOG, the added diversity increases the built-in resiliency against link or node failures. However, the enhanced resiliency is achieved at the expenses of increasing latency and latency variability of the PrOG. *Bounding the overall latency and latency variability* under both normal operations and under failures is important for delay-sensitive (e.g interactive) applications. It can also affect throughput-sensitive applications (e.g bulk transfer), as re-ordered packets can trigger TCP time-out and unnecessary duplicate packet transmission, thus reducing the overall application throughput. With *bounded* latency and latency variability, we can also effectively employ GRO and other mechanisms [11] at the destination to put out-of-ordered packets in order before passing them on to TCP.

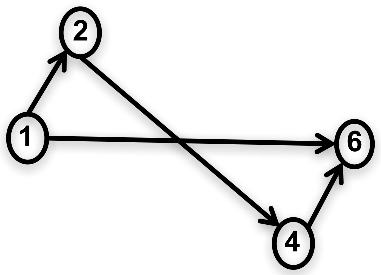
Consider a network $G = (V, E)$ with a link latency matrix $\Phi = [\phi_{ij}]$, where ϕ_{ij} indicates the latency of link (i, j), if $(i, j) \in E$; otherwise $\phi_{ij} = 0$. The latency of a path P in G, $\phi(P)$, is thus the sum of its links. For simplicity of exposition, we will equate path length with its latency. Consider a flow F from s to d with a latency requirement τ_F under normal operations (and a possibly relaxed latency requirement $\tilde{\tau}_F \geq \tau_F$ under failures). We would like to construct a *primary* τ_F-complete routing preorder, or equivalently a PrOG, $\Gamma_{s\to d}(\tau_F)$, such that all paths in $\Gamma_{s\to d}(\tau_F)$ meet the criterion, $\phi(P) \leq \tau_F$, and also construct a *backup* $\tilde{\tau}_F$-complete PrOG $\Gamma_{s\to d}(\tilde{\tau}_F)$, such that $\Gamma_{s\to d}(\tau_F) \subseteq \Gamma_{s\to d}(\tilde{\tau}_F)$. This leads us to introduce the following notion: a PrOG $\Gamma(\tau_k)$ is said to be τ_k*-complete* if any path P from s to d in G such that $\phi(P) \leq \tau_k$ is contained in $\Gamma(\tau_k)$ (as a *directed* path). A τ_k-Complete PrOG can be constructed using a two-phase process based on a modified version of breadth-first search that we sketch below: i) we start with the destination d and perform a breadth first search, where at each step we record for each node, its minimum latency, and the minimum latency of its neighbors; ii) then, we start with the source s, retrace the steps and prune any branches whose latency exceeds τ_k. The complexity of this algorithm is $O(n^2)$. Due to space limitation, the detailed description of the algorithm and its correctness proof are omitted here.

As an illustration, consider the simple network shown in Figure 17(a) where link weight indicates latency. Figures 17(b)-d show three τ-complete PrOGs for $s = 1$ and $d = 6$, with $\tau_1 = 3$, $\tau_2 = 5$, and $\tau_3 = 15$. Given a flow F from s to d with a latency τ_F, we can simply pick one of τ_k's, e.g the largest one, such that $\tau_k \leq \tau_F$. Routing via preorders using a τ-complete PrOG $\Gamma_{s\to d}(\tau)$ has the following *latency-constrained optimal resiliency* property: it is resilient against arbitrary k link or node failures as long as those failures do not partition $\Gamma_{s\to d}(\tau)$ while still meeting the latency requirement τ. This property follows from τ-completeness of the PrOG used for routing as well as the *monotonicity* of τ-complete PrOGs under failures: let $\hat{G}$ be the network under k link (or node) failures; clearly $\hat{G} \subset G$. If $\hat{\Gamma}_{s\to d}(\tau)$ is the τ-complete PrOG for $\hat{G}$, then $\hat{\Gamma}_{s\to d}(\tau) \subseteq \Gamma_{s\to d}(\tau)$. Hence if k failures partition

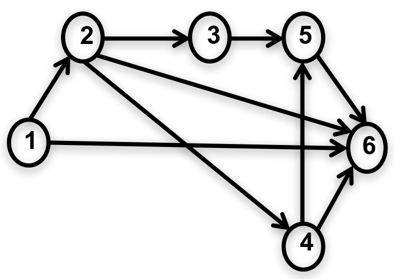
$(U, \prec)$ is a *partially ordered set* or *poset*, yielding a DAG.



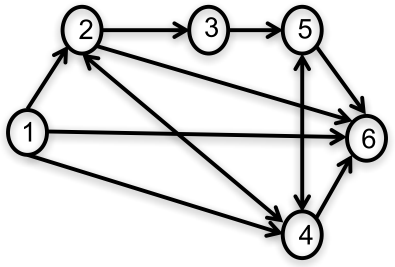
(a) Network Topology



(b) $\tau_1 = 3$



(c) $\tau_2 = 5$



(d) $\tau_3 = 15$

Figure 17: τ_k-complete routing preorders

$\Gamma_{s \to d}(\tau)$, but s can still reach d after these failures, namely, there is a path $\tilde{P}$ from s to d in $\hat{G}$, then $\phi(\tilde{P}) < \tau$.

Resilient Routing via Preoders: Handling Failures and Recoveries. For resilient routing via preorders, we employ a τ_F-complete PrOG[16] $\Gamma(\tau_F)$ as the *primary* preorder for routing under normal operations with the latency requirement τ_F, and a $\tilde{\tau}_F$-complete PrOG $\Gamma(\tilde{\tau}_F)$ where $\tilde{\tau}_F \geq \tau_F$ as the *backup* preorder for resilient routing under failures. As $\Gamma(\tau_F)$ is contained in $\Gamma(\tilde{\tau}_F)$ i.e $\Gamma(\tau_F) \subseteq \Gamma(\tilde{\tau}_F)$, the directed edges in $\Gamma(\tau_F)$ are considered primary links which are used to forward packets of F during normal operations, whereas the directed edges in $\Gamma(\tilde{\tau}_F) \setminus \Gamma(\tau_F)$ are considered backup edges which may only be invoked for packet forwarding during failures. These backup nodes/edges are installed in the relevant switches as (lower priority) backup rules and get invoked when there are failures in the network. Hence, resilient routing with $\tilde{\tau}_F$-resiliency is achieved under arbitrary link/node failures by dynamically *deactivating* certain directed edges that are affected by the failures and *activating* certain backup directed edges when necessary. In the following, we will describe how failures and recoveries are handled via the *deactivation* and *activation* processes, using the PrOG in Figure 16(a) as an example.

Deactivation Process: Suppose the link $F \to G$ goes down. This failure renders F a *sink* (i.e it has no outgoing link). In this case, F simply deactivates the incoming link $D \to F$ by notifying D not to forward packets from $s = A$ to $d = G$ to it. However, this does not affect the reachability from A to G, D simply uses the other outgoing link $D \to E$ to forward packets from A to G. Suppose shortly afterwards, link $D \to E$ goes down, which renders D a sink. This would trigger D to deactivate its two incoming links, $A \to D$ and $B \to D$, and notify its two neighbors. Any existing packets destined to $d = G$, that are buffered at D, will simply be rerouted back to A or B, if they have not exceeded their latency deadline. As a result of these failures and subsequent (*local*) actions at the affected nodes, the original PrOG dynamically shrinks by shedding the deactivated links, and the packets from $s = A$ are now routed solely in the remaining unaffected portion of the original PrOG, namely a sub-PrOG on the node set $\{A, B, C, E, G\}$.

Activation Process: When a failed link or node comes back up, a recovery process is initiated to re-activate the relevant portion of the PrOG. Suppose link $F \to G$ comes back up, then node F is not a sink anymore, which triggers it to activate its incoming link $D \to F$ by notifying D to forward packets from A to G to it. Thus, node D is not a sink anymore as well, which triggers it to activate its incoming links $A \to D$ and $B \to D$ by notifying them. This allows packets from A to G to be routed through the shortest path again (assuming all links have the same weight), i.e the path $A \to D \to F \to G$. Hence, we emphasize that another key advantage of our routing via preorders is that when failed links/nodes are all recovered, it is guaranteed to return to the same PrOG that was

[16]For conciseness, we drop $s \to d$ from $\Gamma_{s \to d}(\tau)$ when the context is clear.

originally selected for resilient routing.

Compared to existing resilient schemes such as [18, 5], which also guarantee resiliency against arbitrary link failures, our proposed routing via preorders does not require any dynamic DAG recomputation (which in the worst case takes $O(n^2)$ steps to converge), or *on-the-fly* graph search by re-routing packets along various paths to search for an available path (which in the worse case takes $O(n)$ steps), our scheme is far simpler and only involves deactivating certain pre-installed rules (directed edges) when a node becomes a sink. Neither [18] nor [5] can ensure only paths satisfying certain latency constraints are used for forwarding under failures. Furthermore, only a small portion of packets, that have already been forwarded to the *deactivated* portion of the PrOG, need to be rerouted, if they have not exceed their latency deadlines; otherwise they will be dropped. This is in contrast to [18] where during the convergence period *all* packets will be subjected to bouncing around in the network until a new DAG is found, and to [5] where after failures *all* packets will "walk around" (with repetitive returns to the sources) in the network following a fixed procedure dictated by the graph search algorithm implemented via SDN rules.

5.3 Realization in SDN

In this section, we show that *routing via preorders* can be realized in SDN, and how packet forwarding is performed under both normal and failure scenarios. We will also briefly touch on the issues and challenges involved - as routing via preorders also requires some additional functionality that goes beyond existing OpenFlow switches.

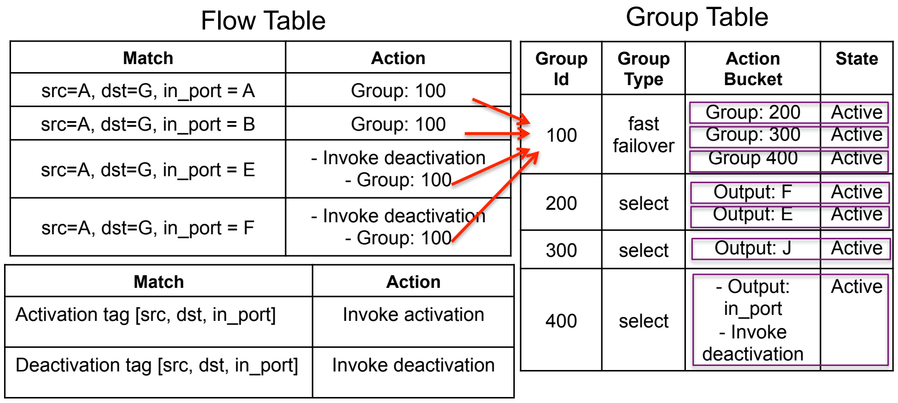
The controller computes the routing preorder that satisfies the latency constraints for each flow, then installs the corresponding rules in the relevant switches. Each switch forwards packets using the *match-action* data plane abstraction, with small added functionality that switches[17] need to be able to maintain and update certain internal states, and generate an *activation tag* or a *deactivation tag* that can be piggybacked in data packets in response to link/node failure and recovery events. Additionally, the switch performs some actions to handle link activation, deactivation and failure.

Under conventional path-based routing, a flow F is first mapped to a pre-selected path P, then the SDN controller sets up the corresponding flow entries in the switches along the path. In routing via preorders, we map a flow F to a primary *PrOG* and a backup *PrOG*, which are translated into a set of match-action rules to be installed in the relevant switches. Preorders can be realized in SDN as match-action rules by using group tables introduced in OpenFlow 1.1.0 [9]. The ingress port (in-port) will always be part of the match fields, so as to ensure only packets from incoming links that are part of the primary or backup textitPrOGs will be forwarded to an eligible outgoing link.

Match-Action Rules, Switch States, and Packet Forwarding. Using the PrOGs in Figure 16(a) (primary) and Figure 16(b) (backup) as an example. Consider node D, the match-action rules for which are shown in Figure 18. At each switch, we need to differentiate between primary and backup links. This can be realized using a "fast failover" group table entry. When a packet of flow F destined to $d = G$ arrives at node D, it is matched, along with its ingress port, against the rules associated with the flow F. Since there are two active primary outgoing links $D \to E$ and $D \to F$ (thus the rules associated with them are inserted in a separate group (GrpId:200), which has higher priority than that with the backup outgoing link $D \to J$ (GrpId:300). The is because "fast failover" group type in OpenFlow executes the first live bucket, thus the order matters, and the liveness of each bucket is controlled by the liveness of its associated port or group. Hence, the primary and backup outgoing links are listed into different buckets in different groups, with the group of primary links listed first in the "fast failover" group (GrpId:100). Only, when all the primary outgoing links are not live i.e (GrpId:200) is inactive, then the backup links are used to forward packets. Later on, if the backup links also become inactive i.e (GrpId:300) is inactive, the switch forwards packets back on any of its in_ports, and invokes the "deactivation process" described earlier (GrpId:400).

Since not all paths in the primary PrOG $\Gamma(\tau_F)$ meet the latency requirement τ_F under normal operations, nor in the backup PrOG $\Gamma(\tilde{\tau}_F)$ meet the (relaxed) latency requirement $\tilde{\tau}_F$ under failures, only a subset of outgoing links at each node i may be eligible for forwarding packets under normal operations and failures. This can be accomplished via two steps. First, we insert two header fields in packets representing its latency requirements, a latency field (similar to the standard TTL field) and a latency offset field. At the source s, the latency field is set to τ_F and the latency offset field is set to $\tilde{\tau}_F - \tau_F$. Any time a packet is forwarded along a link $i \to j$, the latency

[17]Switch and node are used interchangeably.



Flow Table

Match	Action
src=A, dst=G, in_port = A	Group: 100
src=A, dst=G, in_port = B	Group: 100
src=A, dst=G, in_port = E	- Invoke deactivation - Group: 100
src=A, dst=G, in_port = F	- Invoke deactivation - Group: 100

Match	Action
Activation tag [src, dst, in_port]	Invoke activation
Deactivation tag [src, dst, in_port]	Invoke deactivation

Group Table

Group Id	Group Type	Action Bucket	State
100	fast failover	Group: 200	Active
		Group: 300	Active
		Group 400	Active
200	select	Output: F	Active
		Output: E	Active
300	select	Output: J	Active
400	select	- Output: in_port - Invoke deactivation	Active

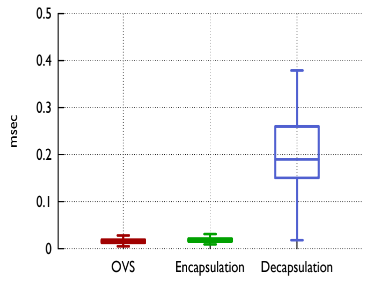
Figure 18: Match-action rules for switch D

field is decremented by ϕ_{ij}; the packet is dropped when this value reaches 0. In the following, we will describe how the latency offset field is used when network failures are encountered. Second, for each outgoing link $i \to j$, we maintain a (per-destination) latency state information which records the minimal (path) latency achievable from node i when forwarding a packet along the outgoing link j to destination d, denoted by $\tau^d(i \to j)$. Hence, given a packet with its current latency header value τ, only those outgoing links with $\tau^d(i \to j) \leq \tau$ are eligible for forwarding the packet.

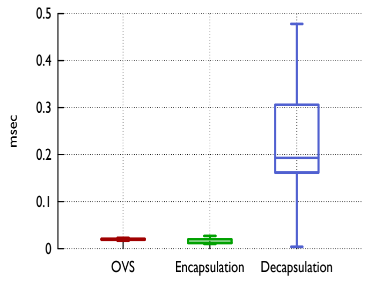
To handle failure (and recovery) events, the state of each outgoing link $i \to j$ indicates whether the match-action rule associated with the (primary or backup) outgoing link $i \to j$ is active or inactive. For example, if the outgoing port j is down or when a deactivation notification is received from node j (see below), the corresponding rule is marked inactive. Hence at node i, if it detects a failure event, e.g an outgoing link/port is down (or a neighboring node is unresponsive), it would mark the rules associated with the outgoing link/port as inactive. If the link/port/node is recovered, the rules are reset to active. The switch also keeps track of the total number of active outgoing links/ports associated with the ruleset of flow F. If this number becomes 0, this renders switch i a sink, namely, no active outgoing link is available for forwarding packets of F. This triggers switch i to invoke the deactivation process: it will generate a special deactivation notification containing the appropriate information (e.g the source-destination pair associated with the affected PrOG) and send it back along all its (active) incoming links $h \to i$; upon receiving this deactivation notification, the downstream switch h would mark the outgoing link $h \to i$ inactive. If this results in switch h becoming a sink, it will continue this deactivation process further downstream. The deactivation notification can be implemented either as a tag inserted in the packet header and can be piggybacked by any data packet traveling along the (reverse) link $i \to h$; or it can be implemented as a special error message sent by i to h. Either mechanisms can be realized as a match-action rule, as shown in Figure 18. When a failed link or node comes back up, a recovery process can be initiated to re-activate the relevant portion of the PrOG.

Packet Forwarding Under Failures. In the event of network failures, e.g when both the primary outgoing links $D \to E$ and $D \to F$ become inactive, the group associated with them also becomes inactive, then the eligible backup outgoing link $i \to j'$ can be invoked for packet forwarding. When a packet is sent along an eligible backup link j' (here, $j' = J$), we first add the value in the latency offset field to the latency header field before the latency header is decremented by $\phi_{ij'}$ and reset the latency offset to 0. If the updated value in the latency header field > 0, the packet is forwarded; otherwise it is dropped. This way we ensure that the packet will meet its (relaxed) latency requirement.

We have implemented routing via preorders in Mininet with a slightly modified Open vSwitch (OVS) using version (2.4) which supports group tables. We added modules to introduce adding extra parameters to group buckets, manipulate extra header fields in packets, dynamically update the state of the ports (logically and not only physically (Up/Down)), select eligible outgoing links, support the activation & deactivation of links, and exchange activation & deactivation tags. We also used a patch to support OpenFlow group chaining as it was not supported before version (2.5). As part of our proposed routing paradigm, we will investigate different methods for efficiently representing and compacting the match-action rules and state information in a way to minimize the space requirements. We plan to deploy and test our novel routing paradigm in the GENI test-bed.



(a) GENI



(b) Mininet

Figure 19: Packet processing delay

6 GENI 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 four 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 3.6 (*Neighbor Echo Request & Reply* and *Port Status*). In the third experiment, we show how VIRO supports host mobility. In the last experiment, we investigate the possibility of using GENI stitching to forwards VIRO frames. 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.

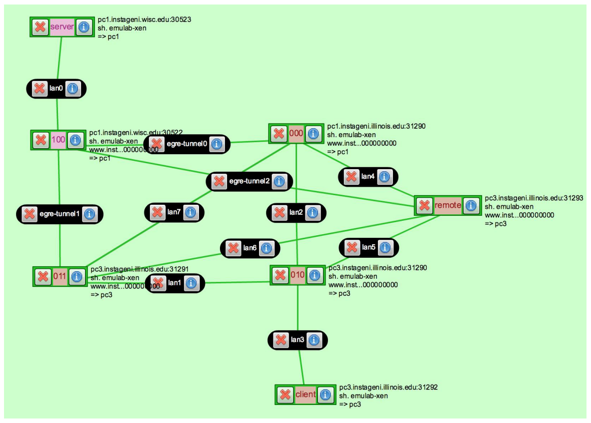
In order to set up and run our VIRO's experiments in GENI, we need to deploy our extend-OVS and VIRO POX controllers (local and remote) to GENI. To achieve this, we first create a GENI node in our slice. Next, we download and install our extend OVS and POX controllers to our GENI node. Then, we create an InstaGENI custom image of our node using Flack [18]. 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.

6.1 VIRO Packet Processing Overhead

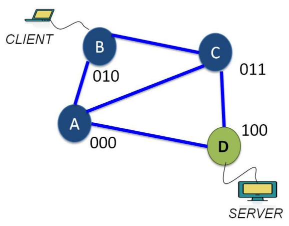
Experiment Setup: in this experiment, we are interested in the answer to the following question: what is the processing delay overhead imposed by VIRO's packets encapsulation/decapsulation at the edges switches? To achieve this, we create a simple topology with two hosts (h1 and h2), connected by two switches (see Figure 12). 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[19] 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 19(a) and 19(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 19(a) and 19(b)). In the future, we will investigate why the processing time for VIRO packet decapsulation is significantly larger than packet encapsulation.

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



(a) Topology in GENI



(b) Network Topology

Figure 20: VIRO failure recovery experiment setup

6.2 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 20(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[20]. We repeat this experiment both in Mininet and GENI.

Figure 20(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 20(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.

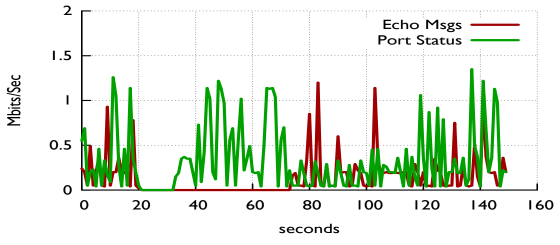
From Figure 21(b) we observe that the failure happens at 23 seconds. We also observe that it takes 5 seconds for the network to recover using the ports status event mechanism. Whereas for the echo-messages mechanism it takes 57 seconds. Similarly, our experimental results from GENI shows similar trend, see Figure 21(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.

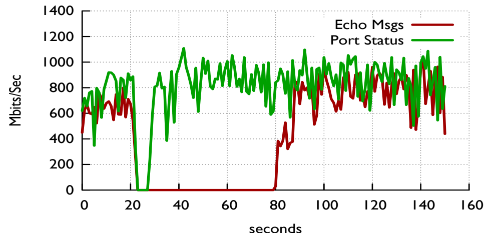
6.3 VIRO Host Mobility

Experiment Setup: In this experiment, we show how VIRO handles host mobility. To achieve this, we use hte network topology illustrated in Figure 10. 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 **??** 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 **??**, nodes A,B are in the same PC. Again, to connect the nodes at different GENI AMs we use EGRE tunnels.

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

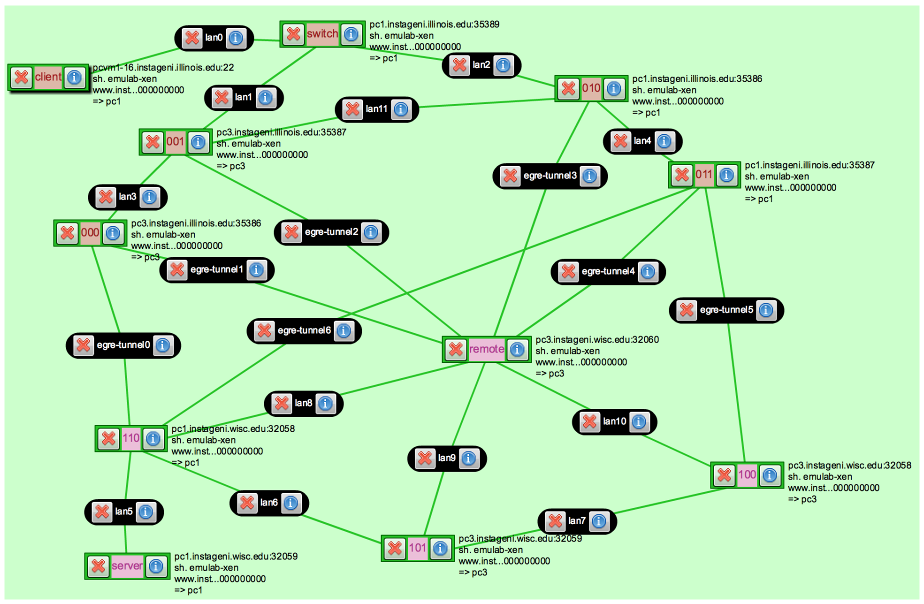


(a) GENI

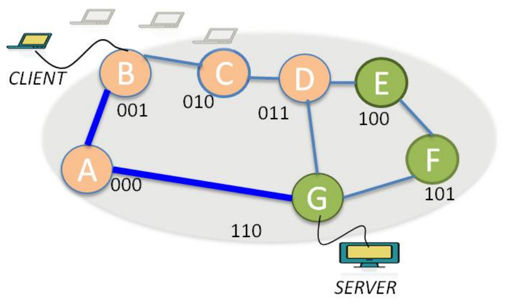


(b) Mininet

Figure 21: Failure Recovery



(a) GENI



(b) Network Topolgy

Figure 22: VIRO mobility experiment setup

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 10). Thus, we used OpenFlow rules to transfer the client's traffic from node C to 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.

6.4 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 Figure 6). We deployed this topology in GENI in two different AMs – Wisconsin(WI) and Illinois(IL).[21]. To connect the switches at the different AMs, we use a stitching link. We transmit VIRO frames [22] from sw1 to sw2, and used the MAC addresses of both switches to set the SVID and 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.

7 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 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[23] 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

[21] sw1 is at WI and sw2 is at IL
[22]We attach a POX controller at sw1 and use it to generate VIRO frames
[23]Omni is a GENI command line tool for reserving resources

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 perform an experiment to explore the possibility of using stitching links to further improve our implementation. However, we find that we cannot forward VIRO frames using vids because stitching forwarding is based on known MAC addresses only. 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. Furthermore, we will implement and deploy our in-network dynamic pathlet switching framework in GENI and develop algorithms that take advantage of the different levels of decisions for path switching, and explore the rich diversity of paths in todays networks.

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] Internet topology zoo. [Online Available].

[2] Open vSwitch. [Online Available].

[3] GENI: Exploring networks of the future. [Online Available].

[4] M. Alizadeh, T. Edsall, S. Dharmapurikar, et al. Conga: Distributed congestion-aware load balancing for datacenters. In *SIGCOMM*, 2014.

[5] M. Borokhovich, L. Schiff, and S. Schmid. Provable data plane connectivity with local fast failover: Introducing openflow graph algorithms. In *HotSDN*, 2014.

[6] M. Caesar, T. Condie, J. Kannan, K. Lakshminarayanan, and I. Stoica. Rofl: routing on flat labels. In *ACM SIGCOMM Computer Communication Review*, volume 36, pages 363–374. ACM, 2006.

[7] J. Feigenbaum, B. Godfrey, et al. On the resilience of routing tables. *arXiv preprint*, 2012.

[8] B. Ford. Unmanaged internet protocol: taming the edge network management crisis. *ACM SIGCOMM Computer Communication Review*, 34:93–98, 2004.

[9] O. N. Foundation. OpenFlow Switch Specification, version v1.1.0, February 28, 2011.

[10] E. M. Gafni and D. P. Bertsekas. Distributed algorithms for generating loop-free routes in networks with frequently changing topology. *IEEE Transactions on Communications*, 1981.

[11] K. He, E. Rozner, K. Agarwal, et al. Presto: Edge-based load balancing for fast datacenter networks. In *SIGCOMM*, 2015.

[12] S. Jain, Y. Chen, and Z.-L. Zhang. Viro: A scalable, robust and namespace independent virtual id routing for future networks. In *INFOCOM, 2011 Proceedings IEEE*, pages 2381–2389. IEEE, 2011.

[13] C. Kim, M. Caesar, and J. Rexford. Floodless in seattle: a scalable ethernet architecture for large enterprises. In *ACM SIGCOMM Computer Communication Review*, volume 38, pages 3–14. ACM, 2008.

[14] A. Kvalbein, A. F. Hansen, et al. Fast ip network recovery using multiple routing configurations. In *INFOCOM*, 2006.

[15] K. Lakshminarayanan, M. Caesar, et al. Achieving convergence-free routing using failure-carrying packets. *ACM SIGCOMM CCR*, 2007.

[16] S. Lee, Y. Yu, S. Nelakuditi, Z.-L. Zhang, and C.-N. Chuah. Proactive vs reactive approaches to failure resilient routing. In *INFOCOM*, 2004.

[17] D. Levin, A. Wundsam, B. Heller, N. Handigol, and A. Feldmann. Logically centralized?: state distribution trade-offs in software defined networks. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 1–6. ACM, 2012.

[18] J. Liu, A. Panda, A. Singla, B. Godfrey, M. Schapira, and S. Shenker. Ensuring connectivity via data plane mechanisms. In *NSDI*, 2013.

[19] M. K. Marina and S. R. Das. On-demand multipath distance vector routing in ad hoc networks. In *Network Protocols,*, 2001.

[20] M. K. Marina and S. R. Das. Routing in mobile ad hoc networks. In *Ad Hoc Networks*. 2005.

[21] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *Peer-to-Peer Systems*, pages 53–65. Springer, 2002.

[22] H. Mekky, F. Hao, S. Mukherjee, Z.-L. Zhang, and T. Lakshman. Application-aware data plane processing in sdn. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 13–18. ACM, 2014.

[23] H. Mekky, C. Jin, and Z.-L. Zhang. VIRO-GENI: SDN-based approach for a non-ip protocol in GENI. In *The 3rd GENI Research and Educational Experiment Workshop*, 2014.

[24] S. Nelakuditi, S. Lee, et al. Blacklist-Aided Forwarding in Static Multihop Wireless Networks. In *SECON*, 2005.

[25] S. Nelakuditi, S. Lee, et al. Fast local rerouting for handling transient link failures. *Transactions on Networking*, 2007.

[26] S. Nelakuditi, S. Lee, Y. Yu, and Z.-L. Zhang. Failure insensitive routing for ensuring service availability. In *IWQoS*, 2003.

[27] P. Pan, G. Swallow, and A. Atlas. Fast reroute extensions to rsvp-te for lsp tunnels, 2005.

[28] J. Pettit. Open vswitch: A whirlwind tour, 2011.

[29] J. Pettit. Ovs deep dive, 2013.

[30] A. Singla, C.-Y. Hong, et al. Jellyfish: Networking data centers randomly. In *NSDI*, 2012.

[31] A. Tootoonchian and Y. Ganjali. Hyperflow: A distributed control plane for openflow. In *Proceedings of the 2010 internet network management conference on Research on enterprise networking*, pages 3–3, 2010.

[32] J. Tsai and T. Moors. A review of multipath routing protocols: From wireless ad hoc to mesh networks. In *ACoRN*, 2006.

[33] X. Yang, D. Clark, and A. W. Berger. Nira: a new inter-domain routing architecture. *ACM Transactions on Networking*, 2007.