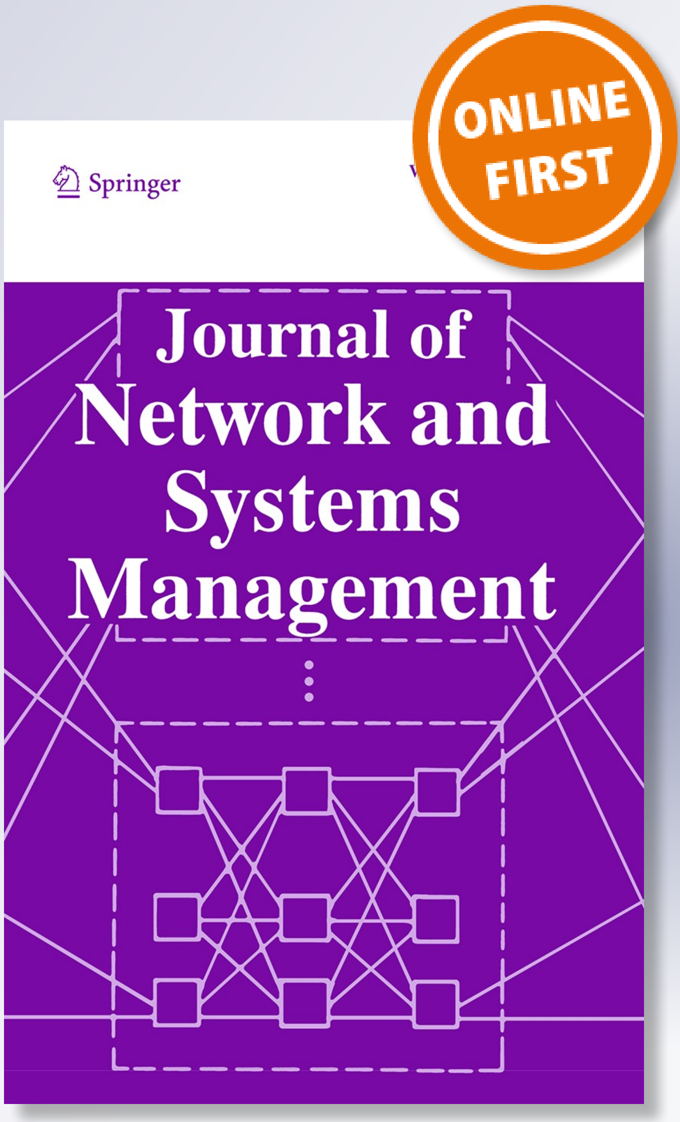
A Virtual Id Routing Protocol for Future Dynamics Networks and Its Implementation Using the SDN Paradigm

Braulio Dumba, Hesham Mekky, Sourabh Jain, Guobao Sun & Zhi-Li Zhang

Journal of Network and Systems Management

ISSN 1064-7570

J Netw Syst Manage
DOI 10.1007/s10922-016-9373-0



Springer

Your article is protected by copyright and all rights are held exclusively by Springer Science +Business Media New York. This e-offprint is for personal use only and shall not be self-archived in electronic repositories. If you wish to self-archive your article, please use the accepted manuscript version for posting on your own website. You may further deposit the accepted manuscript version in any repository, provided it is only made publicly available 12 months after official publication or later and provided acknowledgement is given to the original source of publication and a link is inserted to the published article on Springer's website. The link must be accompanied by the following text: "The final publication is available at link.springer.com".

Springer

Author's personal copy

J Netw Syst Manage
DOI 10.1007/s10922-016-9373-0

A Virtual Id Routing Protocol for Future Dynamics Networks and Its Implementation Using the SDN Paradigm

Braulio Dumba[1] · Hesham Mekky[1] · Sourabh Jain[2] · Guobao Sun[1] · Zhi-Li Zhang[1]

Received: 10 August 2015 / Accepted: 28 March 2016
© Springer Science+Business Media New York 2016

Abstract In this paper, we propose Virtual Id Routing (VIRO) a novel "plug-&-play" non-IP routing protocol for future dynamics 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. It completely eliminates network-wide *flooding* in both the *data* and *control* planes, and thus is highly scalable and robust. VIRO effectively localizes the effect of failures, performs fast re-routing and supports multiple (logical) topologies on top of the same physical network substrate to further enhance network robustness. We have implemented an initial prototype of VIRO using Open vSwitch, and we extend it (both within the user space and the kernel space) to implement VIRO switching functions in VIRO switches. In addition, we use the POX SDN controller to implement VIRO's control and management plane functions. We evaluate our prototype implementation through emulation and in the GENI (the Global Environment for Network Innovations) testbed using many synthetic and real topologies. Our evaluation results show that VIRO has better scalability than link-state based

✉ Braulio Dumba
braulio@cs.umn.edu

Hesham Mekky
hesham@cs.umn.edu

Sourabh Jain
simply.sourabh@gmail.com

Guobao Sun
gsun@cs.umn.edu

Zhi-Li Zhang
zhzhang@cs.umn.edu

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

[2] Google, 1600 Amphitheatre Pkwy, Mountain View, CA 94043, USA

Published online: 06 April 2016

Author's personal copy

protocols (e.g. OSPF and SEATTLE) in terms of routing-table size and control overhead, as well as better mechanisms for failure recovery.

Keywords VIRO · Open vSwitch · Software Defined Networks · GENI

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, inherently reactive approaches for handling network failures and IP routers require extensive manual configuration. In contrast, layer-2 technologies such as Ethernet are largely plug-&-play: hosts are equipped with persistent MAC addresses, and Ethernet switches automatically learn about host addresses and their locations, adapt to changes in the network topology as well as host mobility, and perform packet forwarding seamlessly with minimal operator configuration and intervention. However, Layer-2 Ethernet technology does not scale to large (& wide-area networks), as it provides sub-optimal routing, it is slow to adapt to changes in the network and it is not robust to failures. Therefore, it can hardly meet the scale as well as the requirements for efficiency and robustness imposed on today's large and dynamics networks.

To address these challenges, we need better layer-2/layer-3 networking technologies that are *scalable* (e.g., small routing tables with fast lookup speed), provide better support for *mobility* (e.g., by separating location/addressing and identity/naming), and 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 [1–4] have been proposed to mitigate some limitations of the current Internet technologies. In addition, a flurry of "fixes" [5–10] have also been proposed to address some of these limitations.

In this paper, we propose VIRO—a non-IP routing protocol. It is a novel "plug-&-play" routing paradigm for future large dynamic networks [1]. 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

Author's personal copy

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 [1]. Therefore, VIRO eliminates flooding both in the data and control planes. Furthermore, VIRO is highly scalable and robust, while offering flexible support for multi-homing, mobility, and access control (for enhanced security). Unlike the link-state shortest path routing protocols such as OSPF, VIRO effectively localizes failures and possesses built-in mechanisms for fast rerouting and load-balancing. In addition, VIRO can be readily extended to enable multiple (logical) topologies or multiple virtualized networks on top of the same physical network substrate to further enhance network robustness and service isolation. Because of these features, VIRO is capable of connecting hundreds of thousands of diverse physical devices (with different layer-2 capabilities)—with relative ease and minimal configuration—to form a large, dynamic and heterogeneous network (as a single autonomous system or domain).

We have implemented VIRO using Open vSwitch (OVS). To achieve this, we modify and extend the OVS software to implement VIRO switching functions, in VIRO switches(nodes), and we adapt SDN controllers to implement VIRO's control and management plane functions. Moreover, we evaluate our implementation by running VIRO in the Global Environment for Network Innovation (GENI) [11] testbed. GENI is a wide-area testbed developed by the research community to enable network innovations and large scale experimentations. As part of its network infrastructure, GENI has employed the SDN platform to facilitate testing and deployment for large scale experiments.

The remainder of the paper is organized as follows. Section 2 discusses our related work and we define our notations in Sect. 3. Section 4 describes VIRO and its three key components: vid space construction and vid assignment, VIRO routing and vid-lookup/forwarding. Section 5 discusses our implementation of VIRO using OVS. Section 6 presents our experiments and discusses our experimental results. We conclude and discuss future work in Sect. 7.

2 Related Work

The concept of using distributed hash tables (DHT) for routing over peer-to-peer networks was first introduced in Kademlia [12]. Several adaptations have since been proposed for routing in general, most noticeable of which are the Unmanaged-Internet-Protocol (UIP) [4], the Virtual-Ring-Routing (VRR) protocol [13] and Routing on Flat Labels (ROFL) [3]. These schemes advocate the replacement of the current global IP address space with a flat universal id space and the use of a DHT-style random and consistent hashing for the ids—creating an id-space agnostic to the underlying network topology—to perform routing based on logical distances. They often incur a stretch penalty (which is unbounded in the worst case). Furthermore, several works such as NoGeo [14] and PathDCS [15] have explored coordinate based id assignment based on geo-physical node locations. Although useful for geographically dispersed networks (e.g., a wide-area network), it is of little use in data center (localized) and fast-ethernet like topologies that are often geographically

Author's personal copy

localized. By introducing a topology-aware and structured vid space, VIRO circumvents these problems (see Sects. 4, 6.1).

Closely related to our work is SEATTLE [2]. It employs the OSPF-style shortest path routing in layer-2 to build switching tables in Ethernet switches. Hence, it addresses the scalability issues of Ethernet. Such solutions, unfortunately, still requires network-wide flooding in the control plane for building routing tables; moreover, it suffers the same scalability and robustness limitations plaguing shortest-path routing. In addition, with SEATTLE is not easy to implement load-balancing and fast rerouting. Unlike the link-state shortest path routing protocols, VIRO completely eliminates *network-wide flooding* in both the *data* and *control* planes, localizes failures and possesses *built-in* mechanisms for fast rerouting and load-balancing.

3 Notations and Definitions

In this section, we define and list the key notations and terminologies that will be used to describe VIRO in the remaining sections. Below, we describe and explain our notations:

Logical distance ($\sigma(x, y)$) the logical distance between any two nodes say x and y in an L-bit *vid* space is defined as:

$$\sigma(x, y) = L - lcp(vid(x), vid(y)) \tag{1}$$

Here, $vid(x)$ and $vid(y)$ are the virtual ids for the nodes x and y, $lcp(vid(x), vid(y))$ is the length of the longest common prefix for binary strings $vid(x)$ and $vid(y)$, e.g, if $vid(x) = 0011$, $vid(y) = 0101$, and $L = 4$,then $\sigma(x, y) = 4 - lcp(vid(x), vid(y)) = 4 - 1 = 3$.

Bucket (B_k) for a given node x, the kth bucket $B_k(x)$, is the set of nodes which are at a logical distance of k from node x.

Sub-tree ($S_k(x)$) for a given node x, the kth sub-tree, $S_k(x)$, is the set of nodes which are at no more than logical distance of k from node x.

Rendezvous point ($rdv_k(x)$) for a node x, a *rendezvous point* at level k, $rdv_k(x)$, is a node in the sub-tree $S_{k-1}(x)$, which stores the connectivity information to reach its kth bucket $B_k(x)$. It is the node which is closest to the *vid* given by $vid_{l-r}(x)hash_r(vid_{l-r}(x))$ for $r = k - 1$ in the virtual id space. As seen from the *vid* of the $rdv_k(x)$, it ensures a unique kth level rendezvous point for all the nodes in the sub-tree $S_{k-1}(x)$. The connectivity information stored at $rdv_k(x)$ is the set of edges $(y \leftrightarrow z)$ in the given topology which connect the nodes $y \in S_{k-1}(x)$ to $z \in B_k(x)$. If the subtrees in the bucket $B_k(x)$ are disconnected then it also maps each connectivity information $(y \leftrightarrow z)$ to the prefix in $B_k(x)$ it connects to.

Gateway the *gateway* for a node x to reach Bucket $B_k(x)$ is a node $y \in S_{k-1}(x)$ such that it has a (physical) edge to a node $z \in B_k(x)$. In this case, we refer to node z as distance k *logical neighbor* of node x.

Pid we use *pid* to denote either the physical address (e.g., MAC address), IPv4/IPv6 addresses, persistent name (e.g., a flat-id name) or other addresses/names that

Author's personal copy

Table 1 Routing table for node C as seen in Fig. 1

Bucket	Prefix	NextHop	Gateway
1	00101	—	—
2	0011*	D	C
3	000**	A	C
4	01***	M	C
5	1****	A	B

are used by either lower layer or higher layer to address, name or identify a given entity (an end-host, information or service of interest, etc). The term *pid* is defined in contrast to *vid*, and is primarily used in address/name resolution and first-hop/last-hop data delivery (between a VIRO switch and an end-host) in VIRO.

Host-node a *host-node* for an end-host is the node in the network that it is directly connected to.

Access-node an *access-node* for an end-host is the node which stores the mapping $pid \Rightarrow vid$. An access-node for a given *pid* is determined using the $vid = hash_L(pid)$. It is the node closest (based on the *xor* distance) to the *vid* given by the *hash* value of the *pid*.

Reachability information it is a 4-tuple set, which contains information about the reachability to a given bucket $B_k(i)$ for node *i*. We denote this by $R_k(i)$, and it consists of the following values:

- Bucket level *k*.
- *vid* prefix in $B_k(i)$ that is reachable using this entry.
- *Nexthop* to reach any node in this bucket.
- Logically closest gateway to reach this $B_k(i)$ prefix.

Routing table the routing table for node *x* consists of the reachability information for all the buckets $B_k(x)$ for *x*. Table 1 shows an example of a VIRO routing table.

4 VIRO: Virtual Id Routing Protocol

4.1 Overview

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. VIRO's structured vid space embeds the physical network topology formed by the connections among physical network components. Such embedding is illustrated in Fig. 1 using a Kademlia-like *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 labeled with the bit-strings representing the *vid's* of the VIRO switches residing in that subtree. Next, we describe the main components of

Author's personal copy

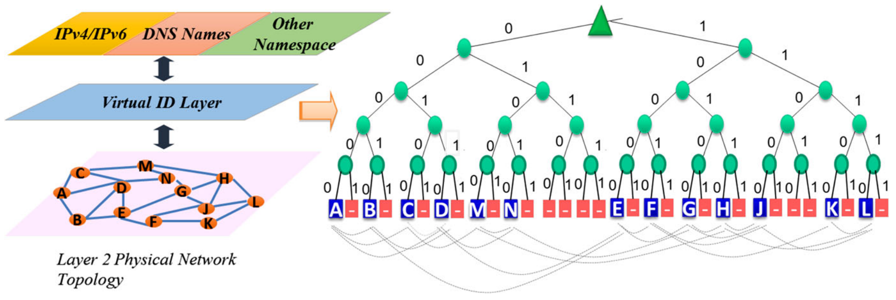


Fig. 1 Vid space as a virtual binary tree: the *grey dotted lines* denote physical connectivity and the *red boxes* represent the unused vid's (Color figure online)

VIRO: virtual id space construction and vid assignment, VIRO routing and vid lookup and forwarding. Moreover, we also describe VIRO's gateway selection and failure recovery mechanisms.

4.2 Virtual Id Assignment

In VIRO, the virtual id space is constructed at the network bootstrapping phase and it is represented by a Kademlia-like (see Fig. 1) virtual binary tree, where only the leaf nodes correspond to physical devices (e.g., VIRO switches) and all the intermediate nodes in the virtual binary tree are logical nodes. The virtual id (vid) of a (leaf) node is a L-bit string vid corresponding to the bits from the root to that leaf node, e.g, in Fig. 1, the *vid* of node B is 00010. The parameter L represents the length of the vid space. It is configured at the bootstrapping phase and it can be selected depending on the size of the target network or other design considerations (e.g., $L = 32, 48, 64, 128$).

During the vid space construction, two key invariant properties must always be maintained:

- *Closeness* if nodes x and y are close in the physical topology, then they are also close in the vid space.
- *Connectivity* for any two logically adjacent sub-trees, $S_k(x)$ and $S_k(y)$, at the same level k, there must exist at least one edge $e \in E$ connecting a pair of nodes (u, v), where $u \in S_k(x)$ and $v \in S_k(y)$.

We have designed two modes of vid space construction/vid assignment: a *centralized* algorithm and a *distributed* algorithm. Both algorithms guarantee that the constructed *vid* space satisfies the above properties:

Centralized vid assignment it employs a top-down approach to assign vids. It starts from the root of the virtual binary tree and recursively partitions the network topology into two connected subgraphs and appends 1-bit to the (already assigned) vid prefixes of the nodes in each of the sub-graphs (see Fig. 2 and Algorithm 1).

Author's personal copy

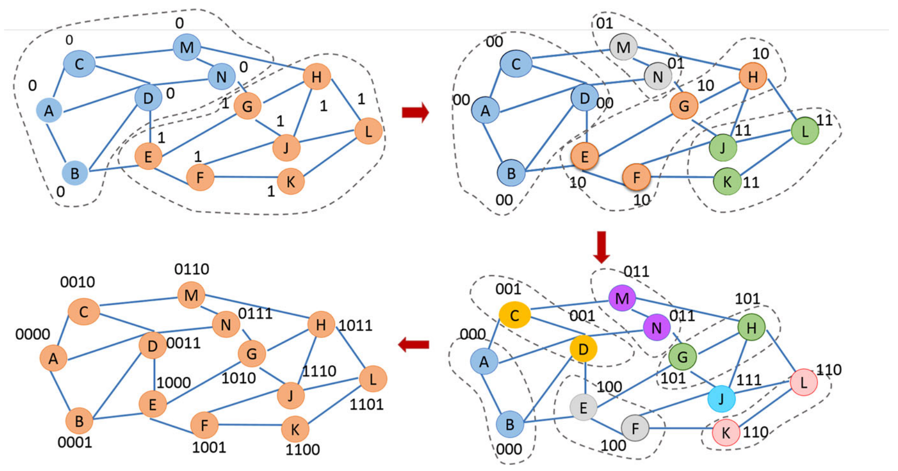


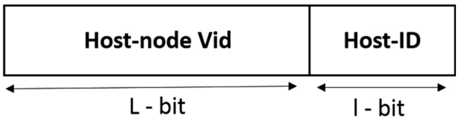
Fig. 2 Virtual id assignment process using top-down algorithm. In this example, nodes are initially partitioned in two clusters, S_1 and S_2. Nodes in S_1 are assigned vid 0 and S_2 are assigned 1, before the bootstrap process. In the subsequent steps, the clusters are further sub-divided and '0' or '1' bit is prepended to their *vids*

Distributed vid assignment it employs a bottom-up approach to assign vid's, by starting from the leaf nodes to the root of the binary tree. In this algorithm, lower-level vid bits are determined first and higher-level bits are recursively assigned.

The *centralized* algorithm is designed for networking environments where the (at least the initial) topology is pre-planned and thus known a priori, e.g, ISPs, large enterprise and data center networks. However, the *distributed* algorithm is more suitable for networking environments where networks are set up in a piecemeal, unplanned or ad hoc fashion, e.g., home or small office networks and wireless ad hoc networks.

After the network is set up and the initial *vid* assignment is completed, using either a *centralized* or *distributed* algorithm, the vid of a subsequent node, say z, joining the network is assigned based on its location and the vid's of its physical neighbors: *it is assigned one of the unused vid's that are closest to one of its physical neighbors*. However, for an end-host that joins the network, its vid is assigned by its host-node and it comprises of two parts (see Fig. 3): the L-bit host-node vid part that identifies the host-node (VIRO switch), and a random l-bit host vid that distinguishes it from other end-hosts attached to the same host-node.

Fig. 3 Vid structure for the hosts attached to a VIRO switch



Author's personal copy

Algorithm 1 Vid assignment using top-down graph-partitioning

```
1: assign_vids($G(E, N)$)
2: $N$ is the set of nodes, $E$ is the set of edges
3: if $|N| \leq 2$ then
4:     Directly assign 1-bit long vids
5: else
6:     $[G_0(E_0, N_0), G_1(E_1, N_1)] = partition\_graph(E, N)$
7:     Append bits 0 and 1 in the vids of the nodes in $N_0$ and $N_1$ respectively
8:     assign_vids($G_0(E_0, N_0, Vid)$);
9:     assign_vids($G_1(E_1, N_1, Vid)$);
10: end if
```

4.3 VIRO Routing

4.3.1 Overview and the Routing Invariant Property

In VIRO a node maintains a routing table with (at most) L entries—one for each level in the vid space.[1] Hence, given a random node x in a VIRO network, all other nodes in the network fall within one of L buckets, $B_k(x)$, $1 \leq k \leq L$. Thus, each node x in VIRO only needs to maintain L entries in its routing table. Then, for any level-k non-empty $B_k(x)$, $1 \leq k \leq L$, as long as node x knows how to reach another node within $B_k(x)$, say $y \in B_k(x)$, then x can reach via y any other node in $B_k(x)$.

VIRO uses a *bottom-up* procedure to build up end-to-end connectivity that obeys the key invariants (closeness and connectivity) discussed in Sect. 4.2. In this method, for each level $k = 2, \ldots, L$, the first $k - 1$ routing entries are constructed before the level-k routing entry is built. More specifically, by the *closeness* property, if node $y \in B_1(x)$ then y must be physically connected to node x. Hence, level-1 route entry can be trivially built. For all other level-k (e.g., $2, \ldots, k$), if $B_k(x)$ is not empty, then by the *connectivity* property there must exist another node y connected to x within $S_{k-1}(x)$ that is physically connected to another node z in $B_k(x)$. Thus, node x can reach any node in $B_k(x)$ via y. Therefore, node y is a level-k gateway to $B_k(x)$, since $y \in S_{k-1}(x)$ and $\sigma(x, y) = k - 1$. Using y we can build the level-k routing entry to reach any node in $B_k(x)$ and node x uses its level $k - 1$ routing entries to reach y. This lead us to the following *Routing Invariant Property* that any VIRO routing table construction algorithm must satisfy:

Routing invariant property let V be the set of all VIRO nodes, and E the set of all *physical* links between VIRO nodes. Suppose the following *connectivity* condition holds for any $x \in V$ and non-empty $B_k(x)$, where $1 \leq k \leq L$:

$$\exists z \in B_k(x) \bigwedge \exists y \in S_{k-1}(x) \quad \textit{such that} \quad (y, z) \in E \tag{2}$$

Then using y as a level-k gateway (namely, including y as a gateway in x's level-k routing entry), it would guarantee the connectivity for node x to reach any node in $B_k(x)$ (see proof in [16]).

[1] Similar to Kademlia and other DHT routing algorithms.

Author's personal copy

4.3.2 Routing Table Computation Algorithm

VIRO employs a DHT-like publish-&-query procedure to build its routing table that satisfies the above invariant property. This procedure completely eliminates *network-wide* (control plane) flooding. Each node x discovers a level-k gateway that satisfies Eq. 2 to reach nodes at every bucket, $B_k(x)$ (e.g., $k = 1, 2, \ldots, L$), in the vid space. More generally, in VIRO a node x builds its routing table using the following steps:

1. Node x discovers its directly connected neighbors via HELO messages or any local broadcast protocol. Its physical neighbors may reside at any bucket in the vid space: $B_k(x), 1 \leq k \leq L$.
2. Node x computes its level-k routing entry using any $z \in B_k(x)$ physically connected to node it as the next-hop to reach Bucket k and itself as the gateway. Furthermore, node x publishes itself as a level-(k) gateway to a level-(k) *rendezvous point(s)*, rdv_k, thereby announcing to all the other nodes in $S_{k-1}(x)$ that its a level-(k) gateway.
3. For the remaining empty entries in its routing table, B_{k+1}, node x discovers and learns a level-$(k + 1)$ gateway by querying gateway information to one of the level-$(k + 1)$ rendezvous point.

The procedure described above is performed *periodically* by each VIRO node and its pseudo-code is given by Algorithm 2.

Algorithm 2 Constructing routing tables for node i

```
1:  $S_0(i) := i, B_0(i) := i$
2:  Input: $N_1(i) \leftarrow$ Physical neighbors for node $i$
3:  Output: Routing table for the node $i$
4:  for $k = 1$ to $L$ do
5:      if $x$ in $N_1(i)$ and $\delta(i, x)$ then
6:          $R_k(i) := (BucketDistance = k, Prefix = pfx_{L-k}(i))$
7:          $NextHop = x, Gateway = i$
8:          Publish $(rdv_k(i), edge(x \leftrightarrow i))$
9:      else
10:         $gw_k :=$ Query$(rdv_k(i), k, i)$
11:         if $gw_k$ is not $Nil$ then
12:             $d := \delta(gw_k, i)$
13:             $nexthop_k := R_d(i).nexthop$
14:             $R_k(i) := (BucketDistance = k, Prefix = pfx_{L-k}(i))$
15:             $NextHop = nexthop_k, Gateway = gw_k$
16:         end if
17:     end if
18: end for
```

Next, using Fig. 1 as an example, we illustrate how routing tables at node C can be constructed using the above procedure. Firstly, during the local physical neighbor discovery process, node C discovers its three direct neighbors: $D \in B_2(C), A \in B_3(C)$ and $M \in B_4(C)$. Then, node C constructs a *null* level-1 routing entry, using the information about its local physical neighbors. In addition, node C constructs

Author's personal copy

level-2, level-3 and level-4 routing entries by entering itself as the gateway and publishing itself as a level-2, level-3 and level-4 gateway to reach $B_2(C)$, $B_3(C)$ and $B_4(C)$ respectively. To build its level-5 routing entry, C queries a level-5 rendezvous point and discovers a level-5 gateway node B (which is connected to node $E \in B_5(C)$). Then, it installs node B as its level-5 gateway in its routing table and node A as the next-hop to reach B. We show the final routing table for node C in Table 1.

4.3.3 Scalability and Complexity of the Routing Algorithm

Given a L-bit vid space, the number of VIRO switches that can be connected to the network is up to 2^L, the size of the VIRO virtual id space. As stated in the previous section, the value of L can be configured at the bootstrapping phase and selected based on the size of the target network or other design considerations. Our implementation of VIRO in OVS (see Sect. 5) uses the 48-bit Ethernet source and destination MAC addresses for the vid's, and we choose $L = 32$ to represent the switch vid's (thus with up to 2^{32} switches in a VIRO network) and an additional 16 bits is used to represent host vid's (thus a total of 2^{16} hosts can be attached to a single VIRO switch). Compared to the conventional IP networks, VIRO is scalable in several aspects. Thanks to the inherent hierarchical structure of the vid space, each VIRO switch only needs to maintain $O(L)$ routing entries, one or a few entries (for load balancing and resilient routing) per level. This is in contrast to conventional IP routers where each router must maintain $O(N)$ routing entries, where N is the number of routers in a single IP network. Given the number of VIRO switches can be up to $N = 2^L$ in a VIRO network, this implies that in the worst case the routing table size of VIRO scales in the order of $O(\log_2 N)$ as opposed to $O(N)$. We remark that routing table size is a major constraining factor that limits the scalability of classical IP routers. The much smaller routing table size not only significantly reduces the memory requirement of VIRO switches, it also leads to much faster routing table look-up speed, thereby faster data packet processing and forwarding.

Furthermore, VIRO incurs significantly lower control overheads when compared to the standard link-state routing protocols such as OSPF used in today's IP networks. This is because *by design*, VIRO does not resort to any form of *network-wide* flooding of control packets, in constrast to OSPF which relies on network-wide flooding of link state adjacency (LSA) packets to learn about network topology and changes in the topology, nor does VIRO resort to any form of data packet flooding for forwarding, as in the case of an Ethernet switch which employs an adaptive "self-learning" algorithm to build the switch forwarding table; it broadcasts a data packet to its neighbors when it does not know where to forward the packet. Instead, VIRO employs a *query-&-reply* mechanism through the rendezvous points where a kth-level gateway registers with the kth-level rendezvous point (rdv_k) and other switches within the k-level subtree query rdv_k to build the k-level routing entries (see Algorithm 2). As a result, it takes $O(L)$ rounds to build the VIRO routing table with $O(L)$ control messages exchanged between a VIRO switch and

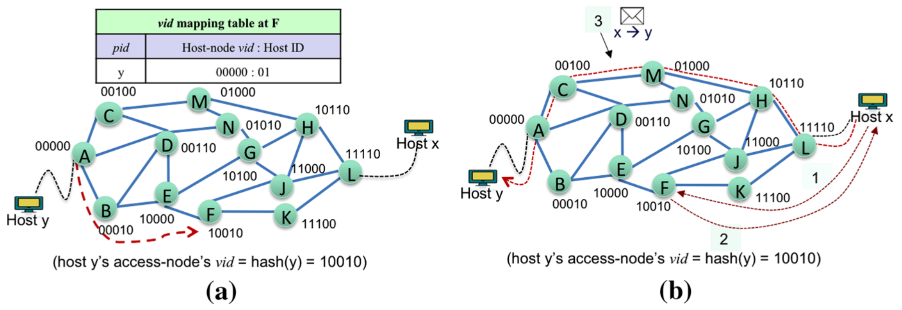
Author's personal copy

corresponding rendezvous points, where for each level k, it takes $O(2^k)$ steps to deliver these control messages in the worst case per kth-level subtree. This is in contrast to OSPF where it takes $O(N)$ steps to flood the LSA packets per router with a total $O(N^2)$ control messages exchanges; and moreover, it takes $O(|E|\log N)$ time complexity per router to compute the routing table using Dijkstra's algorithm, where $|E|$ is the number of edges. $|E|$ can be in the order of $O(N^2)$ in the worst case for a network with rich topology (e.g., full-meshed).

4.4 Virtual Id Lookup and Forwarding

In VIRO, a Kademlia-style DHTs is used to implement vid look-up mechanisms and address/name resolution (i.e., pid-vid mappings). Depending on the look-up speed and memory requirements/constraints, either one-hop or multi-hop DHTs may be used for these operations, as in SEATTLE [2]. Once the *vid* of an end-host is looked up using its *pid*, packet forwarding between VIRO nodes is performed using *vid* only. However, at the network edges the *vid's* are mapped to a 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. For more flexible and better support for mobility, geographically-scoped hash functions (see [17, 18]) may also be used for vid lookup and address/name resolution. Next, we discuss VIRO vid-lookup mechanism in details:

The *host-node* assigns *vid's* to any end-host h connected to it. In addition, the host-node stores end-hosts *pid-vid* mapping in its local cache. Furthermore, using $pid(h)$ as the key, it periodically publishes the $\langle pid(h), vid(h)\rangle$ mappings (e.g., MAC address, IP address, or a flat-id name to vid mappings) to the respective *access-node* in the network—recall from Sect. 3 that an access-node is a node whose *vid* is closest to the $hash_L(pid(h))$ based on the *XOR* distance, as in [12]. This *mechanism* is illustrated in Fig. 4a: when a host y joins the network by connecting to node A, the host-node A assigns a *vid* to host y and publishes the *pid-vid* mapping to the relevant access node, which is F in this example. Thus, when another node wants to look up the vid of host y, it uses the hashed key, $vid = hash_L(pid(y))$ to query the network. Then, the corresponding access-node (whose vid is closest to $hash_L(pid(y))$) responds with the stored $vid(y)$ of host y, if it exist; otherwise a error message is returned.



vid mapping table at F	
pid	Host-node *vid* : Host ID
y	00000 : 01

Fig. 4 VIRO vid-lookup and forwarding mechanism, **a** *vid* publish process, **b** steps in packet forwarding

Author's personal copy

As stated above, packet forwarding between VIRO nodes is performed using *vid* only. In Fig. 4b, we illustrate VIRO's packet forwarding mechanism, which consists of two steps:

(a) *Host vid lookup* when host x wants to send a packet to destination host y, it sends a *pid-vid* mapping request to its host-node (node L), which forwards the request to the corresponding access-node (node F) for host y (step 1 in Fig. 4b). Then, node F returns host y *vid* to host x (step 2 in Fig. 4b).

(b) *Packet forwarding using vid* routing in VIRO is done based on destination vid and gateway information. When host x sends a packet to host y, according to VIRO routing protocol, node L will compute the *logical* distance between its vid and A's vid, namely $\sigma(L, A) = 5$. Assuming its level 5 routing table is not empty, node L will look up its routing table for a level-5 gateway (node H) and forward the packet to the next-hop to reach its level-5 gateway (which is also node H). The next-hop follows similar process until the packet is delivered to the destination node A (step 3 in Fig. 4b).

A similar forwarding mechanism is used to process VIRO control packets, e.g., gateway *publish* or *query* packets, *pid-vid* mapping registration, etc. The main difference here is that the destination vid does not correspond to a physical node, instead it is a key that is meant to identify the VIRO node whose vid is closest to this key. In this case, for node x receiving a packet with vid = *dest*, if its level-k routing entry is empty, where $k = \sigma(x, dest)$, it does not drop the packet. Instead, it flips the $(L - k)$th bit (counting from the left) in the destination vid, and uses this updated vid to look up the routing table, to find a valid next-hop to reach the node closest to the vid. If the level-($k - 1$) routing entry is also empty, it flips the $(L - (k - 1))$th bit in the (updated) destination *vid*, and looks up its level-$(k - 2)$ routing entry. This process stops either when a next-hop is found or node x discovers that it is the closest node to this destination vid (see Algorithm 3).

Algorithm 3 Packet (msg) forwarding at node i

```
1:  $nexthop = Nil$
2:  $k = \sigma(i, msg.dest)$
3:  if $R_k(i)$ not $Nil$ then
4:      $nexthop = R_k(i).nexthop$
5:  end if
6:  if $msg.type = CONTROL$ then
7:      while $nexthop = Nil$ or $k = 0$ do
8:          $msg.dest = Flip\_Kth\_Bit(msg.dest)$
9:          $k := \sigma(i, msg.dest)$
10:         $nexthop = R_k(i).nexthop$
11:     end while
12: end if
13: if $msg.dest = i$ then
14:     processPacket($msg$)
15: else
16:     sendPacket($nexthop$, $msg$)
17: end if
```

Author's personal copy

4.5 Gateway Selection Strategy and Multi-Path Routing

We use a *gateway selection* strategy that is *consistent* in order to ensure that no routing loops is formed during the computation of the routing tables. Firstly, let assume that only one (default) gateway is installed in the routing table of each node. Then, when node x queries a level-k rendezvous point $rdv_k(x)$ to discover a level-k gateway, $rdv_k(x)$ always select one of those gateways whose *vid* is closest to the *vid* of the querying node x.

However, when *multiple* gateways are installed in the routing tables (e.g., for load-balancing or fast rerouting), a generalized consistent gateway selection rule is achieved by associating each level-k gateway node a special *forwarding directive*: a L-bit key, which is associated with the level-k gateway whose vid is closest to this key and its first $L-k$ bits are the same as those in the *vid* of the querying node. Hence, when a (level-k) gateway is selected to reach $B_k(x)$, its respective forwarding directive is also included in the packet header to direct subsequent packet forwarding towards this gateway.

Thus, if a source node x wants to use a specific gateway z to reach the destination bucket, it sets the forwarding directive field in the packet to z. Then, if an intermediary node y sees a packet with the forwarding directive field set, it forwards the packet towards the forwarding directive instead of the destination. However, if the forwarding directive field is empty, node y selects an appropriate gateway from its routing table to set the forwarding directive field. However, when the packet reaches node, say z, in the forwarding directive, node z can either reset the forwarding directive field or use the destination vid to forward the packet. The *forwarding directive* field is also used to re-route traffic in the event of node/link failures: if a link to the current next-hop for the destination (or forwarding directive) has failed, then the current node can forward the packet towards a gateway for which it has a working next-hop entry. We will further discuss the failure recovery mechanism in the next section.

By utilizing multiple gateways in the routing tables at each bucket level, VIRO provides built-in support for multi-path routing, load-balancing and fast re-routing. For example, using m gateways to reach each bucket enables m-way multipath routing. More precisely, this allows a node to selectively choose one of the m different paths to reach a destination bucket, either for load-balancing or for fast re-rerouting. By including the gateway node's *vid* as part of the forwarding directive in the packet header, we can guarantee that there are no forwarding loops (see [16] for a correctness proof of the VIRO routing algorithm under the consistent gateway selection strategy).

4.6 Handling Node/Link Failures

VIRO utilizes a *withdraw & update* mechanism to handle node/link failures, without resorting to flooding for failure notifications (as used in OSPF). A node adjacent to a failed node (e.g., a gateway node) or a failed link (to a gateway node) withdraws its previously published *connectivity information* from the appropriate rendezvous point(s). Thus, when a rendezvous point receives this withdraw notification, it sends

Author's personal copy

an *update* message containing the *withdrawal* of the current gateway, replacing it with a new gateway to all (or a subset of) nodes in a affected sub-tree – namely, those that are currently using the failed or no longer reachable gateway in their routing tables. Therefore, failures are *localized* because only the nodes that are affected by the failures, or in the same subtree as the failure node need to update their routing entries. When a rendezvous node fails, a neighboring node would then take over and serve as the new rendezvous node. This node would recover the *connectivity information* from one or a combination of the following methods: (a) the current gateway(s) stored in its routing table; (b) another rendezvous point, when multiple rendezvous points are used; and (c) through periodic publications by available gateways. In practice, we assume that for large networks, multiple rendezvous points will be used for enhanced robustness.

5 SDN Implementation OF VIRO

5.1 Design Overview and Implementatin Challenges

We have implemented VIRO using Open vSwitch (OVS). The architecture of our VIRO node[2] is illustrated in Fig. 5. It contains three main components [19]: data plane, control plane and management plane. VIRO nodes use OVS [20] in the data plane and POX controllers in both the control and management planes. To implement the VIRO data plane, we re-purpose the Ethernet MAC address to represent VIRO virtual id. Furthermore, we also modify and extend OVS OpenFlow actions (both within the user and kernel spaces) to realize VIRO packet forwarding functions. The OVS daemon (slow-path 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.

Open vSwitch implements the OpenFlow switch specifications and the 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 [1], and it has build-in multipath and fast failure (re)routing capabilities. Recall from Sects. 4.4 and 4.5, 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" functions in OpenFlow. In the following, we present our design and implementation framework (VIRO SDN

[2] We use the terms "node" and "switch" interchangeably.

Author's personal copy

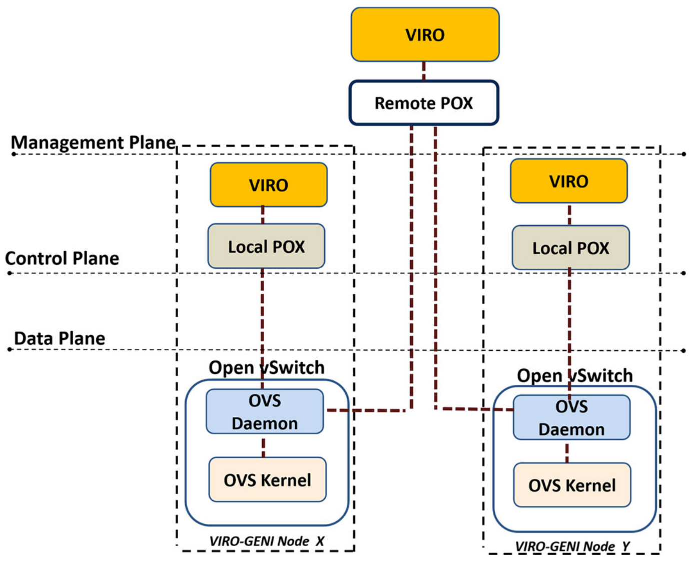


Fig. 5 Software stack in a VIRO node

Data Plane and VIRO SDN Control Plane) as well as solutions to overcome the challenges in adapting the OVS to implement a non-IP protocol such as VIRO.

5.2 VIRO SDN Data Plane

For the data plane implementation, we use OVS version 1.0 with Nicira's extensions. The OVS implementation consists of two components: a kernel (fast path) and a user space (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 [21]. 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 for reading the OpenFlow configuration from ovsdb-server[3] [22]. Its packet classifier supports efficient flow lookup with wildcards and checks datapath flow counters to handle flow expiration and statistics requests [22].

When a packet arrives 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 user space 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

[3] Database that holds switch-level configuration.

Author's personal copy

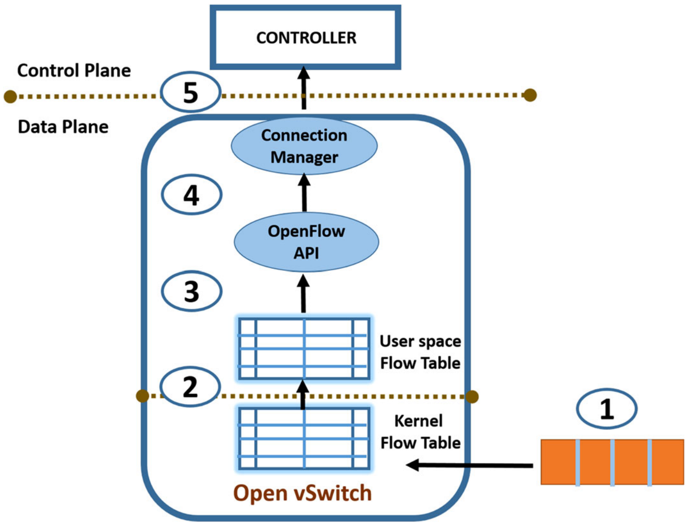


Fig. 6 Packet processing in Open Vswitch

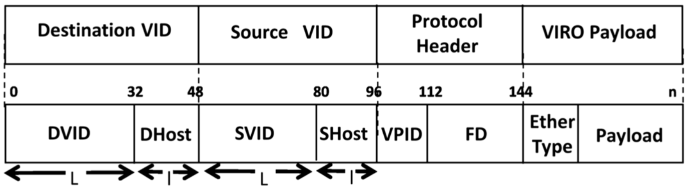


Fig. 7 Structure of a VIRO frame

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 [23]. Figure 6 illustrates this process (steps 1–5).

To implement VIRO using the OVS-SDN platform, we first define the structure of a VIRO frame. To achieve this, we re-purpose the standard Ethernet frame as follows (see Fig. 7): we reuse the 6-bytes of the source and destination MAC addresses (SMAC and DMAC) to represent VIRO virtual addresses (vids). More precisely, we use 4-bytes from the DMAC to set the destination switch's vid (DVID) and the remaining 2-bytes to set the destination host identifier (DHOST). Similarly for the SMAC, we re-use the 6 bytes to represent VIRO source switch's vid (SVID) and source host (SHOST) identifier respectively. In addition, we introduce new 6 bytes for the VIRO protocol header field, where 2-bytes are reserved for VIRO's protocol identifier (VPID) and the others 4-bytes are used to set VIRO's *forwarding directive* (FD) field. The remaining bytes are used for the payload of the VIRO frame, which is composed of the EtherType and the payload of

Author's personal copy

Table 2 List of the new actions added to our extended OVS

Actions	Description
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

the standard Ethernet frame. A VIRO frame has the EtherType 0x0802 which differentiate it from standard Ethernet protocols such as IP, LLDP and ARP. We also added a new EtherType 0x0803 for VIRO control packets (see Sect. 5.3). The VIRO frame header uses more bytes than the standard Ethernet frame. Hence, we use Path MTU Discovery at the end-hosts to reduce their frame size, in order to avoid encapsulation without using any fragmentation [24].

As discussed in the previous section, 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 behavior. Thus, VIRO's forwarding cannot be directly implemented using the standard "match-action" functions of OpenFlow. Therefore, in order to forward our VIRO frame in the OVS data-path, we modify and extend the match and actions of the OVS fast and slow paths with the following new actions (see Table 2): 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.

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

5.3 VIRO SDN Control Plane

VIRO's SDN control plane consists of two main components: *control plane* and *management plane*. Next, we describe the functionalities of each of these components:

Control plane (local controller) the local controller (LC) in the control plane implements VIRO's routing functions: neighbor discovery, routing table computation, failure recovery and all the rendezvous point functions. In addition, it also

Author's personal copy

handles VIRO's packet encapsulation/decapsulation (at the end points) and packet miss from the data-plane (see Sect. 5.4). Recall from Sect. 5.2 that VIRO control packets are identified by the protocol ID 0x0803 in the frame payload (EtherType) to differentiate them from VIRO data packets (e.g., IP packets). The LC handles all types of VIRO control packets:

- *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.

Management plane (remote controller) in the management plane, we have the VIRO remote controller (RC) and it is the single instance that all VIRO switches in the network connect to (see Fig. 5). This controller is responsible for all the network management functions that can be performed in a *centralized* fashion. It implements some of the *access-node* and *host-node* functions, such as: it assigns *vid* to end-hosts and maintains the host's *pid-vid* mapping. In addition, RC is also in charge of the following: network topology discovery and maintenance (host/switch added or removed), switch vid assignment, ARP and DHCP requests,[4] and to maintain a global view of the network. In the following section, we discuss in details the tasks of the RC.

5.4 VIRO Network Bootstrapping Events

In this section, we present the main events that occur during the bootstrapping phase of a VIRO network. Recall that the OVS in each VIRO node is connected to both a local controller (running the VIRO module) and to a single remote controller (running centralized management functions) in the network. Initially, the following events take place:

Connection up when a VIRO switch starts, it immediately connects to both the local controller(LC) and the remote controller (RC), using the standard OpenFlow protocol. Upon connection, the RC inserts rules to receive all the ARP and DHCP packets generated by host machines in the network.

Vid assignment we use a *centralized* approach to assign *vid's* to hosts and switches. The RC periodically sends *Controller_Echo* message to the LCs with the vid assignment to the respective switches[5] and it saves the switches DPID/VID mapping to its topology table. For end-hosts, the RC assigns vid's during the DHCP lease process and it also saves the mapping MAC/IP/VID/PORT for every end-host

[4] We reuse POX's ARP and DHCP modules.

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

Author's personal copy

in the network. In addition, the RC sends the host's *pid-vid* mapping information to the corresponding LC.

Neighbor and failure discovery the LC in a VIRO switch sends *Neighbor Echo Request* messages every t seconds to discover the (physically) directly connected neighbors. It timestamps, t_x, when a *Neighbor Echo Reply* message is received, and saves the neighbor's vid and t_x values in a table. We use these values to find the failed neighbors. For example, if an entry in the table is not updated after t' seconds, then we consider the correspondent switch as failed. In addition, we also use OpenFlow *Port Status* messages for neighbor failure discovery.

Routing table construction the VIRO module attached to each VIRO switch exchange VIRO control packets (*RD_Publish, RDV_Query, RDV_Reply* and *Neighbor Echo Request & Reply*), in order to build the routing table in each switch using VIRO's *publish-&-query* algorithm described in Sect. 4.3. These routing tables are later installed in the OVS flow-tables in the slow-path and fast-path.

End-host discovery during the DHCP lease process, the RC sends *Local_Host* messages with IP/MAC/VID/PORT mapping to the LC that the respective end-host is attached to. The LC stores the *pid-vid* mapping for future end-host name resolution, as well as, to build its local view of the network.

Pid-vid resolution as discussed in Sect. 4.4, VIRO uses a one-hop (or multi-hop) DHT for *pid-vid* look-up and resolution. However, for simplicity, in our current implementation, we use a *centralized* approach for *pid-vid* resolution. First, when an end-host 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. Second, when one end-host x wants to communicate with another end-host y in a VIRO network, it first issues an ARP request. The VIRO switch attached to it forwards the ARP packet to the RC. Then, RC returns host's y vid in the ARP reply by replacing DMAC with host's y vid (recall that RC has a global view of the network).

In summary, upon starting, a VIRO switch connects to the VIRO controllers (RC and LC), and it receives its vid from the RC. Then, it exchanges *Neighbor Echo Request & Reply messages* to discover its (physically) direct connect neighbor switches and uses VIRO's *publish-&-query* mechanism to build its routing table. Lastly, it discovers its attached hosts during the DHCP lease process (*Local_Host* messages from RC).

5.5 Packet Forwarding in a VIRO Network

In this section, we explain how the *pid-vid* mappings and packet forwarding is performed in a network composed with VIRO switches. To achieve this, we use the example illustrated in Fig. 8. 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 switch x forwards the ARP query to RC.

Author's personal copy

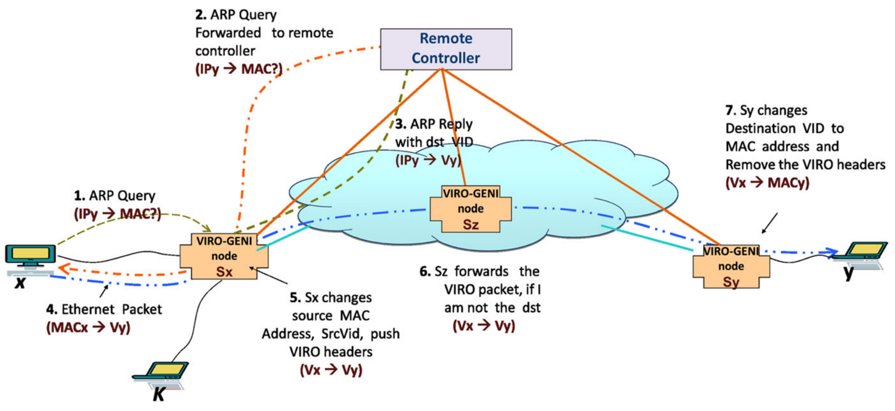

2. ARP Query Forwarded to remote controller (IPy → MAC?)
Remote Controller
3. ARP Reply with dst VID (IPy → Vy)
7. Sy changes Destination VID to MAC address and Remove the VIRO headers (Vx → MACy)
1. ARP Query (IPy → MAC?)
VIRO-GENI node Sx
VIRO-GENI node Sz
VIRO-GENI node Sy
x
y
K
4. Ethernet Packet (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)

Fig. 8 VIRO packet forwarding between two host machines

- RC returns the ARP reply packet and it replaces the DMAC with the vid of host y, which is composed of switch y vid prepended to host y *l*-bit identifier (recall that RC has a global view of the network).
- Host x receives the ARP reply and generates the first Ethernet frame, whose DMAC address is host y vid. This frame is forwarded to switch x.
- The Ethernet frame will be received by the source's access-node (switch x), and it will generate a miss in the OVS fast-path and slow-path. Then, the frame will be send to VIRO LC, and it will replace the SMAC with the SVID. In addition, it will push the VIRO headers into the Ethernet frame and forward the packet to the next destination, according to its routing table. Lastly, it will add OpenFlow rules to insert the VIRO packet header into packets received from host x and to set the SVID and SHOST appropriately. This will cause future packets to be forwarded by the fast path.
- The intermediary VIRO switches (e.g., switch z) will forward the VIRO packets to the next hop, according to their VIRO routing table (this process may include rewriting the FD).
- When the VIRO packet is received by the destination VIRO switch y, it will first generate a miss in the OVS fast and slow path. Then, the packet will be sent 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 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 switch x.

Author's personal copy

6 Experiments

In this section, we present the results of our evaluation of VIRO using simulations and a real testbed. We have developed our customized in-house simulator for VIRO, and carried out experiments to evaluate and compare VIRO with several existing routing protocols such as OSPF and SEATTLE [2], using various real and synthetic network topologies (see Table 3):

- *Router level AS topologies* we use the following router level AS topologies from the RocketFuel project [25]: (1) AS 1755, (2) AS 3967, and (3) AS 6461.
- *Data center topologies* we generated multiple of Fat-Tree [26] topologies by varying the number of nodes in them. Here, we provide results for the following three topologies: (1) DC125, (2) DC320, and (3) DC500. These topologies are arranged in three layers (ToR, Aggregation switches and Core switches). Thus, the maximum shortest distance between any two switches(nodes) is 4 hops.
- *Synthetic router level AS topologies using Brite* we use the *Barabasi* model in Brite [27] to generate router level AS topologies containing different number of nodes (200, 400 and 600).

Additionally, we have conducted a number of experiments in the GENI testbed to evaluate our OVS/SDN prototype of VIRO. We discuss our GENI experiments in Sect. 6.2.

6.1 VIRO Simulation Experiments

Through extensive simulations, in this section, we evaluate VIRO using the network topologies in Table 3. In these experiments, we compare VIRO with several existing link-state routing protocols such as OSPF and SEATTLE [2] using the following metrics:

Routing table size a key metric for evaluating the scalability of a routing protocol is the size of the routing table at each node. Recall that nodes in VIRO keep only one routing entry to reach each level-k Buckets. There are only $O(log_2(n))$ number of such buckets in a network of n nodes. In contrast, nodes in the *link-state* routing keep n routing entry to reach each node in the network. Figure 9 shows the size of the routing tables for VIRO and link-state routing protocols for different topologies. It shows that VIRO creates much smaller routing tables than link-state routing protocol. This is because VIRO stores only one routing entry for each logical

Table 3 Summary of the topologies used in our simulations

Router level ASs	Data center	BRITE
AS1755 (295, 543)	DC125 (125, 500)	BT200 (200, 790)
AS3967 (353, 820)	DC320 (320, 2048)	BT400 (400, 1590)
AS6461 (654, 1332)	DC500 (500, 4000)	BT600 (600, 2390)

Author's personal copy

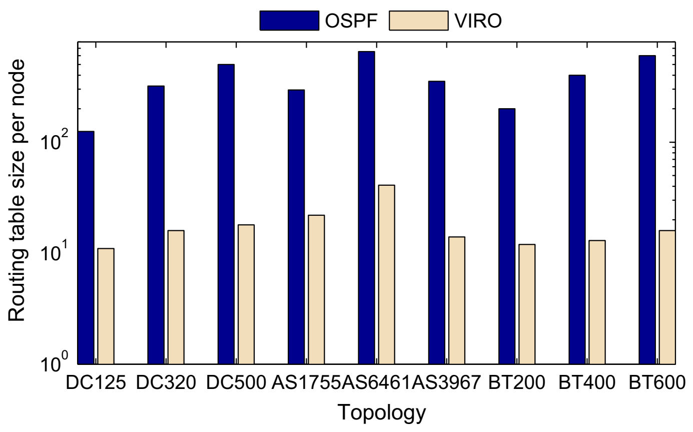


Fig. 9 Routing table size comparison

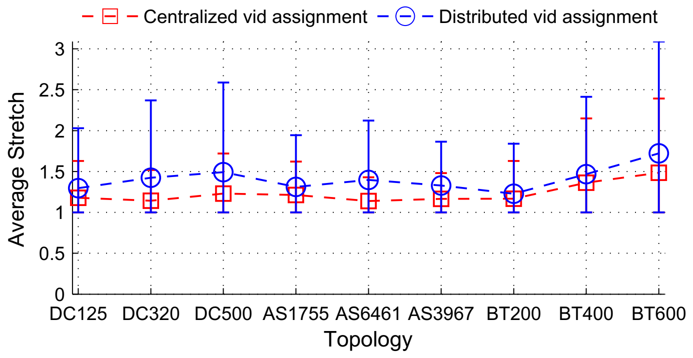


Fig. 10 Routing stretch distribution for VIRO

distance. On the other hand, in link-state protocols (e.g., OSPF and SEATTLE) nodes keep a routing entry for every node in the network, which grows linearly with the number of nodes in the network.

Routing stretch unlike algorithms such as OSPF, VIRO does not use shortest path routing. Thus, it induces a small overhead in terms of the routing optimality. This overhead is measured using *routing stretch* (RS). We define RS as the ratio of the length of the path taken using VIRO and the shortest path length between a source and destination pair. Our experimental results shows that the average RT remains close to 1 for most of the topologies (see Fig. 10). Additionally, the *bottom-up* approach (distributed) for vid assignment incurs much larger RT than *top-down* (centralized) vid assignment. It is because an optimal *vid* assignment is achieved using graph-partitioning algorithms.

Control overhead we estimate the control-overhead for a node by counting the number of control-messages processed by that node to build its routing table. We compare the overhead due to control-messages used by VIRO and link-state

Author's personal copy

protocols. For VIRO, we consider four different variants by allowing more than one rendezvous node at different level. Here VIRO-1, VIRO-2, VIRO-4 are different variants of VIRO with maximum of 1, 2 and 4 rendezvous nodes at each level respectively. Since the number of node pairs increases exponentially with logical distances, i.e., there are maximum of 2^k node-pairs at k logical distance. We also consider another variant of VIRO by allowing maximum of $log(k)$ number of rendezvous nodes at *kth* level. We refer to this variant of VIRO as VIRO-log. The results are shown in Fig. 12, the x-axis represents the different topologies and y-axis (plotted on log-scale) shows the average number of control-messages processed by each node in the network. As seen in this figure, control-overhead is much smaller for VIRO than link-state. It is because nodes using VIRO's *publish-&-query* mechanism exchange fewer control packets than using the "flooding" based mechanism used by the link-state routing protocol. Next, we compare the control overhead on rendezvous nodes at any level. This overhead is created by the rendezvous publish/query messages processed by rendezvous nodes. In our simulations, we measure this by counting the number of such publish/query message received by each rendezvous node. Figure 11 shows the distribution of control-overhead on rendezvous nodes at different levels. In this Figure, x-axis represents the level and the y-axis shows the number of publish/query messages received by the rendezvous node. It shows that control overhead increases with the level of rendezvous node. However, having more number of rendezvous nodes helps significantly in reducing the overhead on individual rendezvous nodes.

Vid lookup cost both SEATTLE and VIRO requires the look-up for the host location to send packets to them. In case of SEATTLE, switches store the host to switch mapping by constructing a 1-hop DHT. Similarly in VIRO, we store *pid-vid* mappings at switches by constructing a Kademlia style DHT. In Fig. 13, we plot the number of hops taken to resolve these mappings for VIRO and SEATTLE. It shows that lookup overhead for VIRO is slightly larger than SEATTLE, which is due to the greater than 1 routing stretch for VIRO. However, the difference is less than a hop for most of the topologies using the centralized vid assignment.

Failure control-overhead in Fig. 14a we compare the control overhead due to the failure notification messages for VIRO and OSPF. In this Figure, the y-axis (plotted using log-scale) shows the average number of control-messages processed by each

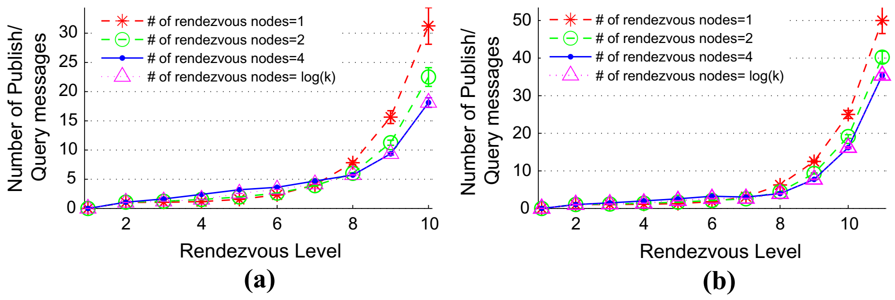


Fig. 11 Comparison of control overhead on rendezvous nodes for VIRO with different number of rendezvous points **a** DC125, **b** BT200

Author's personal copy

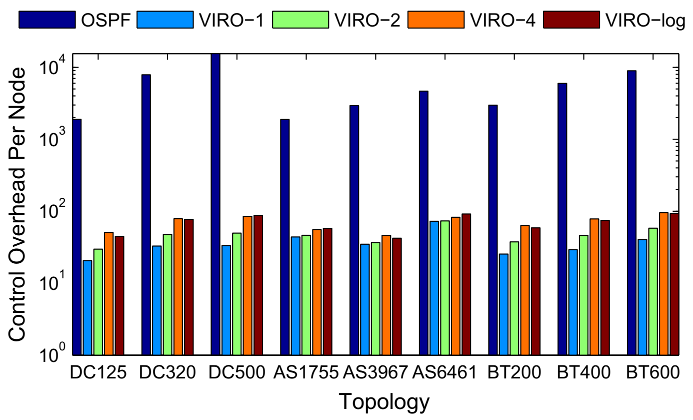


Fig. 12 Control-overhead comparison

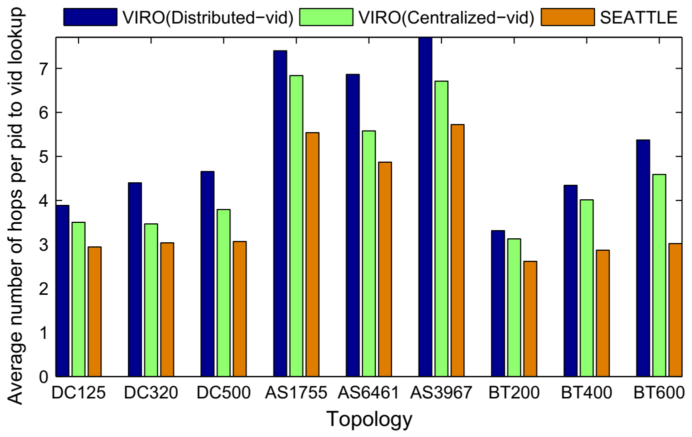


Fig. 13 Look-up costs for VIRO and SEATTLE

node for VIRO and OSPF for the corresponding topology shown in the x-axis. As seen in this Figure, the no flooding based mechanism used in VIRO helps in reducing the number of "failure notification" messages drastically, while the overhead of OSPF style routing protocols is much larger. In Fig. 14b, we evaluate the overhead of control messages due to the failures of the rendezvous nodes at different levels. We observe that the control-overhead to spread the failure notifications increases with the level of rendezvous node. However, this overboard remains very small for even higher levels, e.g., it is only 6 control messages per node for the failure of the rendezvous node at level 14. Lastly, in Fig. 14c, we investigate the effectiveness of VIRO to localize the effect of failures by comparing the control overhead on the nodes with the logical distance from the failed node. This figure shows that nodes which are logically far from the failure are less affected

Author's personal copy

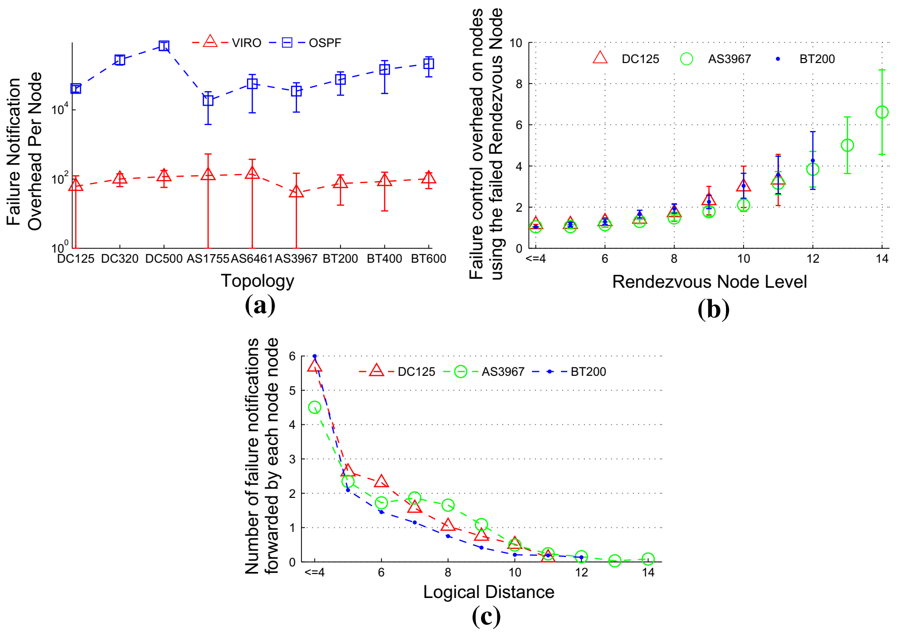


Fig. 14 Comparison of VIRO and link-state for failures (vertical bars show the 95 % confidence interval for the mean values), **a** Control overhead, **b** RDV failure, **c** failure localization

by the failure. However, failures are more likely to affect the nodes which are close to it. Hence, VIRO is very effective in localizing the effect of failures.

6.2 VIRO GENI Experiments

We have conducted a number of experiments in the GENI testbed to evaluate our initial prototype of VIRO. In this section, we describe two sets of experiments. In the first experiment, we investigate VIRO's packet encapsulation/decapsulation overhead at edges switches. In the second experiment, we evaluate and compare VIRO's failure recovery mechanism as discussed in Sect. 5.4 (*Neighbor Echo Request & Reply* and *Port Status*). The results of these experiments will help us to improve our initial prototype of VIRO, e.g., to select the best failure recovery mechanism.

Encapsulation/decapsulation overhead in this experiment, we are investigating the processing delay overhead imposed by VIRO's packets encapsulation/decapsulation at the edges switches. To achieve this, we deployed the topology illustrated in Fig. 15a in GENI using the Illinois GENI Aggregate Manager, 1 raw PC and 4 Xen VMs. We measure the processing delay of ping messages[6] from h1 to h2. We use *tcpdump* to obtain the timestamps of packets as they enter and leave the switches. The difference in the timestamps is the "packet processing delay time"—

[6] We generate 100 ping request packets.

Author's personal copy

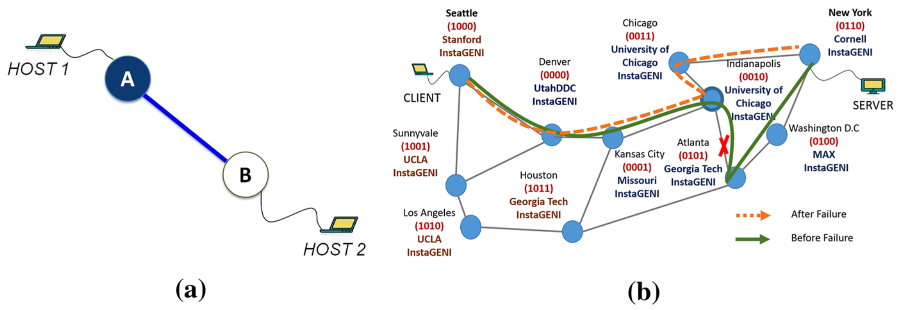

HOST 1
A
B
HOST 2
(a)
Seattle (1000) Stanford InstaGENI
CLIENT
Denver (0000) UtahDDC InstaGENI
Chicago (0011) University of Chicago InstaGENI
Indianapolis (0010) University of Chicago InstaGENI
New York (0110) Cornell InstaGENI
SERVER
Sunnyvale (1001) UCLA InstaGENI
Kansas City (0001) Missouri InstaGENI
Atlanta (0101) Georgia Tech InstaGENI
Washington D.C (0100) MAX InstaGENI
Houston (1011) Georgia Tech InstaGENI
Los Angeles (1010) UCLA InstaGENI
After Failure
Before Failure
(b)

Fig. 15 Network topologies of our experiments in GENI, **a** packet processing delay, **b** failure recovery

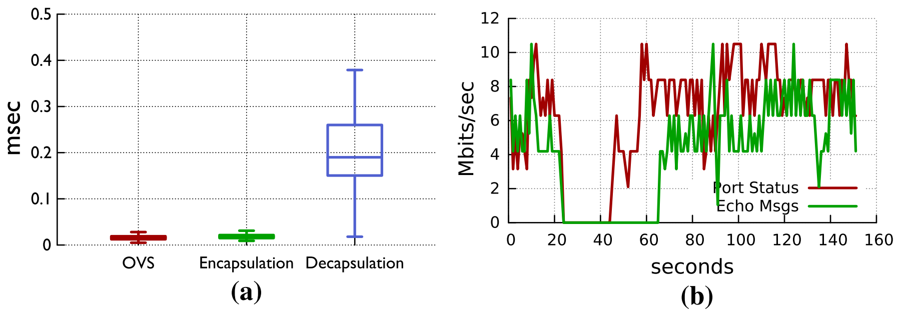


Fig. 16 GENI experimental results, **a** packet processing delay, **b** failure recovery

since we do not generate high amounts of traffic, we consider the queue delay negligible. We repeat this experiment using both a traditional OVS (with the standard IP forwarding) and our extended OVS.

Figures 16a shows the processing time in milliseconds for a traditional OVS, extended-OVS encapsulation (encap-OVS) and extended-OVS decapsulation (decap-OVS). We observe from our experimental results that the 95 percentile for packet's processing delay is 3.11×10^{-2}, 3.01×10^{-2} and 3.50×10^{-1} milliseconds for OVS, encap-OVS and decap-OVS. These results show that the packet's processing delay for OVS and encap-OVS are very close. Surprisingly, we observe an increase in the packet's processing time for decap-OVS (see Fig. 16a). In the future, we will investigate why the processing time for VIRO packet decapsulation is significantly larger than packet encapsulation.

Failure recovery 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 Fig. 15b. We deployed this topology in GENI using 8 GENI InstaGENI Aggregate Managers (AMs), 10 raw PCs, 14 Xen VMs and EGRE tunnels to connect the GENI AMs. For this experiment, a client in Seattle communicates with a server in New York. 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. 15b). During

Author's personal copy

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

We use the network tool *iperf* to generate traffic from the client to the server for 150 s. Figure 16b shows the results of our experiment. We observe that the failure of link *Indianapolis* → *Atlanta* happens at about 20 s. It takes 20 s for the network to recover using the port status method (see Fig. 16b). However, it takes about 60 s for the network to recover using VIRO echo-messages. These results shows that the *Port status* method outperforms *Neighbor Echo Request & Reply* method, as expected. We also observe that recovery time for both methods is significantly large. Hence, in the future we will improve the tuning of our experimental parameters in order to decrease the failure recovery time in our experiments.

7 Conclusion

In this paper, we proposed VIRO—a novel “plug-&-play”, scalable and robust non-IP routing paradigm for future large dynamics networks. The key idea behind VIRO is the introduction of a topology-aware, structured virtual id (vid) space onto which both physical identifiers (e.g., Ethernet MAC addresses) as well as higher layer addresses/names (e.g., IPv4/IPv6 addresses or flat-id names) are mapped. Taking advantage of such a topology-aware and structured vid space, VIRO employs a DHT-style routing algorithm to build routing tables, look up objects (names, addresses, vids, etc.) and forward packets. Hence, VIRO completely eliminates network-wide flooding in both the data and control planes. Furthermore, because of the structured vid space, VIRO effectively localizes the effect of failures, performs fast rerouting and support multiple (logical) topologies on top of the same physical network substrate to further enhance network robustness. VIRO also facilitates the support for virtualized networks and network services, as well as enables access control and isolation of services for security and performance.

We have developed our customized in-house simulator for VIRO and carried out experiments to evaluate and compare VIRO with several existing routing protocols such as OSPF and SEATTLE, using various real and synthetic network topologies. Our experimental results show that VIRO outperforms OSPF and SEATTLE, and it has immense scalability and robustness, while keeping the control overheads very low. Furthermore, we have also developed an initial prototype of VIRO using the OVS-SDN platform and deployed it in the GENI testbed. We modified OVS to implement VIRO switching functions and adapted the POX SDN controller to implement VIRO control and management plane functions. In addition, we have

Author's personal copy

carried out experiments to test our initial prototype of VIRO in the GENI testbed. Moreover, we plan to expand our current prototype of VIRO to include additional functionalities. These include further extensions to OVS 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.

Acknowledgments This research was supported in part by a Raytheon/NSF subcontract 9500012169/CNS-1346688, DTRA grants HDTRA1-09-1-0050 and HDTRA1-14-1-0040, DoD ARO MURI Award W911NF-12-1-0385, and NSF grants CNS-1117536, CRI-1305237 and CNS-1411636.

References

1. Jain, S., Chen, Y., Zhang Z.: VIRO: A scalable, robust and namespace independent virtual id routing for future networks. In: Proceedings of the IEEE INFOCOM, (2011). doi:10.1109/INFCOM.2011.5935058
2. Kim, C., Caesar, M., Rexford, J.: Floodless in Seattle: a scalable ethernet architecture for large enterprises. In: Proceedings of the ACM SIGCOMM, (2008). doi:10.1145/1402958.1402961
3. Caesar, M., Condie, T., Kannan, J., Lakshminarayanan, K., Stoica, I.: ROFL: routing on flat labels. In: Proceedings of the ACM SIGCOMM, (2006). doi:10.1145/1151659.1159955
4. Ford, B.: Unmanaged internet protocol: taming the edge network management crisis. In: Proceedings of the ACM SIGCOMM, (2004). doi:10.1145/972374.972391
5. Myers, A., Ng, E., Zhang, H.: Rethinking the service model: scaling Ethernet to a million nodes. In: Proceedings of the ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets), ACM (2004)
6. Sharma, S., Gopalan, K., Nanda, S., Chiueh, T.C: Viking: a multi-spanning-tree Ethernet architecture for metropolitan area and cluster networks. In: Proceedings of the IEEE INFOCOM, (2004). doi:10.1109/INFCOM.2004.1354651
7. Rodeheffer, T.L., Thekkath, C.A., Anderson, D.C.: Smartbridge: a scalable bridge architecture. In: Proceedings of the ACM SIGCOMM, (2000). doi:10.1145/347059.347546
8. Kim, C., Rexford J.: Revisiting Ethernet: plug-and-play made scalable and efficient. In: Proceedings of the 15th IEEE Workshop on Local and Metropolitan Area Networks, (2007). doi:10.1109/LANMAN.2007.4295993
9. Ray, S., Guerin, R., Sofia, R.: A distributed hash table based address resolution scheme for large-scale ethernet networks. In: Proceedings of the IEEE International Conference on Communications (ICC), IEEE (2007). doi:10.1109/ICC.2007.1066
10. Alaettinoglu, C., Shankar, A: Viewserver hierarchy: a new inter-domain routing protocol and its evaluation. In: Proceedings of the IEEE INFOCOM, (1993). doi:10.1109/INFCOM.1994.337589
11. GENI: Exploring networks of the future. https://www.geni.net/
12. Maymounkov, P., Mazieres, D.: Kademlia: a peer-to-peer information system based on the XOR metric. In: Proceedings of the IPTPS, (2002)
13. Caesar, M., Castro, M., Nightingale, E. B., O'Shea, G., Rowstron, A.: Virtual ring routing: network routing inspired by DHTs. In: Proceedings of the ACM SIGCOMM, (2006). doi:10.1145/1159913.1159954
14. Rao, A., Ratnasamy, S., Papadimitriou, C., Shenker, S., Stoica, I.: Geographic routing without location information. In: Proceedings of the ACM 9th Annual International Conference on Mobile Computing and Networking (MobiCom), ACM (2003). doi:10.1145/938985.938996
15. Ee, C., Ratnasamy, S., Shenker, S.: Practical data-centric storage. In: Proceedings of the 3rd Conference on Networked Systems Design and Implementation (NSDI), vol. 3, pp. 24–24, ACM (2006)
16. Jain, S., Chen, Y., Zhang Z.: VIRO: A plug & play, scalable, robust and namespace independent virtual id routing for future networks. In: Tech report. http://networking.cs.umn.edu/veil/viro
17. Lu, G.H., Jain, S., Chen, S., Zhang, Z.: Virtual id routing: a scalable routing framework with support for mobility and routing efficiency. In: Proceedings of the 3rd International Workshop on Mobility in the Evolving Internet Architecture (MobiArch), ACM (2008). doi:10.1145/1403007.1403025

Author's personal copy

18. Yu, Y., Lu, G., Zhang, Z.: Enhancing location service scalability with HIGH-GRADE. In: Proceedings of the IEEE International Conference on Mobile Ad-hoc and Sensor Systems, (2004). doi:10.1109/MAHSS.2004.1392102
19. Dumba, B., Mekky, H., Sun, G., Zhang, Z.: Experience in implementing and deploying a non-IP routing protocol VIRO in GENI. In: Proceeding of the IEEE International Workshop on Computer and Networking Experimental Research Using Testbeds, IEEE (2014). doi:10.1109/ICNP.2014.85
20. Open vSwitch. http://www.openvswitch.org/
21. Pettit, J.: A Whirlwind Tour (2011)
22. Pettit, J.: OVS Deep Dive, OpenStack Summit (2013)
23. Mekky, H., Hao, F., Mukherjee, S., Zhang, Z., Lakshman, T.: Application-aware Data Plane. In: Proceedings of the ACM SIGCOMM Workshop on Hot topics on Software Defined Networks (HotSDN), ACM (2014). doi:10.1145/2620728.2620735
24. Mekky, H., Jin, C., Zhang, Z.: VIRO-GENI: SDN-based approach for a non-IP protocol in GENI. In: Proceedings of the Third GENI Research and Educational Experiment Workshop (GREE), IEEE (2014). doi:10.1109/GREE.2014.14
25. Spring, N., Mahajan, R., Wetherall, D., Anderson, T.: Measuring ISP topologies with rocketfuel. In: Proceedings of the IEEE/ACM Transactions on Networking, IEEE (2004). doi:10.1109/TNET.2003.822655
26. Al-Fares, M., Loukissas, A., Vahdat, A.: A scalable, commodity data center network architecture. In: Proceedings of the ACM SIGCOMM, (2008). doi:10.1145/1402946.1402967
27. Medina, A., Matta, I., Byers, J.: BRITE: a flexible generator of Internet topologies. Technical Report, ACM (2000)

Braulio Dumba is currently a Ph.D. student at the Department of Computer Science and Engineering at University of Minnesota, Twin Cities. In 2011, he earned a B.A. in Computer Science and Physics, with a Mathematics minor, from Luther College. His research interests lie in Network Routing Protocols, Software Defined Networks and Online Social Networks.

Hesham Mekky received his B.Sc. in Computer Science from Alexandria University, Egypt in 2007, and M.Sc. in Computer Science from University of Minnesota in 2013. He is currently a Ph.D. candidate of Computer Science at University of Minessota. His research interests include networking, security, and privacy. He is a student member of IEEE.

Sourabh Jain graduated from University of Minnesota, twin cities, in 2011, with a Ph.D. in Computer Science. He finished his undergraduate studies in Computer Science and Engineering at Indian Institute of Technology Roorkee (IIT-Roorkee) (2002–2006). Currently, he is a Software Engineer at Google.

Guobao Sun is a software engineer at Facebook, Inc. He received his B.S. in Computer Science and Engineering from Shanghai Jiao Tong University in 2013, and M.S. in Computer Science and Engineering from University of Minnesota, Twin Cities in 2015. His research interests mainly lie in Software-Defined Networking (SDN) and wireless networking.

Zhi-Li Zhang is a full Professor at the Department of Computer Science and Engineering at University of Minnesota, Twin Cities. He is also a Qwest Chair Professor, McKnight Distinguished University Professor and a Fellow of IEEE. His research interests lie broadly in computer communication and networks, Internet technology, multimedia and emerging applications.