Creating Scalable, Efficient and Namespace Independent Routing Framework for Future Networks

A DISSERTATION
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Sourabh Jain

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
Doctor of Philosophy

Prof. Zhi-Li Zhang

May, 2011

ⓒ Sourabh Jain 2011
ALL RIGHTS RESERVED

Acknowledgements

My journey as a graduate student would not have been possible without the help of several people. First and foremost, I want to thank my advisor, Prof. Zhi-Li Zhang, for teaching me how to conduct research, for providing me with constant support and encouragement.

I thank my friends and lab mates for their assistance in my research. In particular, I enjoyed working with Vijay on several traffic analysis related projects. Working with Yingying on her research was an enriching experience. As a graduate student in the networking lab, I was priveleged to work with several talented colleagues including Inderpreet, Rohini, Guor-Huar, Shanzhen, Esam, Gyan, Pengkui, Yu, Ji, Jia, Yanhua and Hal.

I am grateful to Prof. Chandra, who guided me on log-analysis research project. He provided valuable suggestions with constructive feedback on my research. Similarly, Dr. Bronevetsky's help on supercomputing log-analysis project was exteremely valuable.

I thank Professors Du, Odlyzko and Chandra for serving as my PhD thesis committee member and spending their precious time and efforts during my PhD defense. I would like thank the Computer Science Department staff for their valuable services.

Finally, I am grateful to my parents for their unconditional love and support. There have been several friends who helped me at various instances during my PhD life. In particular, I would like to thank my friends Sunayana, Sudhir, Sunil, Pramod, Aravindan, Shruti, Peng-Fei, Ajay and Sujit for their support and help.

Dedication

This thesis is dedicated to several important people in my life. I dedicate it to my parents, my teachers and my friends.

Abstract

In this thesis we propose VIRO — a novel and paradigm-shifting approach to network routing and forwarding that is not only highly scalable and robust, but also is namespace-independent. VIRO provides several advantages over existing network routing architectures, including: i) VIRO directly and simultaneously addresses the challenges faced by IP networks as well as those associated with the traditional layer-2 technologies such as Ethernet — while retaining its "plug-&-play" feature. ii) VIRO provides a uniform convergence layer that integrates and unifies routing and forwarding performed by the traditional layer-2 (data link layer) and layer-3 (network layer), as prescribed by the conventional local-area/wide-area network dichotomy and layered architecture. iii) Perhaps more importantly, VIRO *decouples routing from addressing*, and thus is namespace-independent. Hence VIRO allows new (global or local) addressing and naming schemes (e.g., HIP or flat-id namespace) to be introduced into networks without the need to modify core router/switch functions, and can easily and flexibly support inter-operability between existing and new addressing schemes/namespaces.

In the second part of this thesis, we present *Virtual Ethernet Id Layer*, in short VEIL, a practical realization of VIRO routing protocol to create a large-scale Ethernet networks. VEIL is aimed at simplifying the management of large-scale enterprise networks by requiring minimal manual configuration overheads. It makes it tremendously easy to plug-in a new routing-node or a host-device in the network without requiring any manual configuration. It builds on top of a highly scalable and robust routing substrate provided by VIRO, and supports many advanced features such as seamless mobility support, built-in multi-path routing and fast-failure re-routing in case of link/node failures without requiring any specialized topologies. To demonstrate the feasibility of VEIL, we have built a prototype of VEIL, called *veil-click*, using Click Modular Router framework, which can be co-deployed with existing Ethernet switches, and does not require any changes to host-devices connecting to the network.

Contents

Acknowledgements **i**

Dedication **ii**

Abstract **iii**

List of Tables **viii**

List of Figures **ix**

1 Introduction **1**

1.1 Requirement for Higher Scalability 2
1.2 Support for Seamless Mobility 3
1.3 Higher Availability & Reliability 4
1.4 Manageability & Security 5
1.5 Summary 6
1.6 Bibliographic Notes 7

2 Background **8**

2.1 Ethernet Network Architecture 8
2.2 Interconnecting Ethernet LANs using IP 10
2.3 Link State Routing Protocols 12

3 VIRO: Virtual ID Routing **14**

3.1 Introduction 14
3.2 Overview and Related Work 16

3.2.1 Design of VIRO: An Overview 16
3.2.2 Notations & Definitions 20
3.2.3 Related Work 22
3.3 Virtual Id Assignment 23
3.3.1 vid Space Construction at Bootstrap 24
3.3.2 *vid* Assignment upon New Node Join 25
3.3.3 Host *vid* Assignment 26
3.4 VIRO Routing 26
3.4.1 Overview and the Routing Invariant Property 26
3.4.2 Routing Table Construction Algorithm 28
3.4.3 Bridge 30
3.4.4 Basic Forwarding Algorithm 33
3.4.5 Handling Node/Link Failures 33
3.5 Virtual ID Lookup and Forwarding 34
3.6 Additional Features & Discussion 36
3.7 Evaluation 38
3.7.1 vid-assignment evaluation 38
3.7.2 Routing Overheads 40
3.7.3 Failure Dynamics 43
3.8 Conclusion 45

4 VEIL: Virtual Ethernet ID Layer 55
4.1 Introduction 55
4.2 Overview and Related Work 57
4.2.1 Overview of VEIL 57
4.2.2 Related Work 59
4.3 The VEIL Layer 59
4.3.1 VEIL Id Assignment 59
4.3.2 Virtual Id Routing (VIRO) Protocol used by VEIL-Switches 60
4.3.3 Id Lookup/Address Resolution, and End-to-End Packet Delivery 61
4.3.4 Support for Mobility and Legacy Protocols 62
4.4 Conclusion 63

5 VEIL-Click: Prototype VEIL-Switch **64**
5.1 Introduction 64
5.2 Overview 65
5.2.1 Motivation 65
5.2.2 Related Work 67
5.2.3 VEIL-click: Design Overview 68
5.3 VEIL-click: Key Components 70
5.3.1 vid Management 70
5.3.2 Routing & Forwarding 73
5.3.3 Host-device Namespace Management 73
5.4 VEIL Features 74
5.4.1 No Manual Configuration 74
5.4.2 Policy Control using Custom Namespace Resolutions 74
5.4.3 Robust Host Mobility Support 75
5.4.4 Multi-path Routing & Fast Failure-Rerouting 75
5.5 VEIL-CLICK: The Prototype 76
5.5.1 Basic Design Modules 76
5.5.2 Initial Evaluation 82
5.6 Conclusion 85

6 Conclusion and Discussion **86**
6.1 Conclusion 86
6.2 Future Work 87

References **89**

Appendix A. VIRO: Properties and Proofs **97**
A.1 Basic Properties 97
A.1.1 Gateway Selection Strategy 99
A.1.2 Properties 100
A.2 Routing Algorithm Correctness Proof 101

List of Tables

2.1 Comparison of link-state advertisement cost. 12
3.1 Summary of notations 20
3.2 Routing table for a node in VIRO 22
3.3 Routing table for node A shown in Fig. 3.2 46
3.4 Routing tables (before bridge) for the nodes in Fig. 3.4 48
3.5 Routing tables (after bridges) for the nodes in Fig. 3.4 49
3.6 Summary of the topologies used in evaluation. 49
A.1 Routing tables at the end of round-4 98
A.2 Routing tables at the end of round-5 99

List of Figures

3.1 Overview of VIRO. 17
3.2 Virtual ID assignment process using bottom-up clustering method. 25
3.3 vid structure for the hosts. 26
3.4 Violation of *logical connectivity constratint* on vid assignment 30
3.5 vid publish process. 35
3.6 Steps in packet forwarding. 36
3.7 Distribution of $physical$(MinHop) distance with the $logical$ distance. 39
3.8 Distribution of routing stretch for VIRO. 40
3.9 Routing Table size comparison for VIRO and link-state routing. 50
3.10 Look-up costs for VIRO and SEATTLE. 50
3.11 Control-overhead for VIRO and the link-state based routing protocols. 51
3.12 Comparison of memory-overhead for rendezvous nodes at different levels . . . 51
3.13 Control overhead on rendezvous nodes. 52
3.14 Comparison of VIRO and Link-state for failures. 53
3.15 Localized affect of failures for VIRO 54
3.16 Comparison of disconnected node-pairs for VIRO and OSPF 54
4.1 vid field structure for VEIL. 59
4.2 vid lookup and address resolution process. 61
5.1 An example showing large-scale layer-2 network using VEIL-Click. 69
5.2 An overview of the design of Click based prototype. 76
5.3 An initial prototype of *veil-switch*. 82
5.4 Intial testbed created using prototype *veil-switches*. 83
5.5 Host-device mobility and the traffic bit rate. 84
5.6 TCP throughput comparison. 85

A.1 Multiple gateways to reach $B_5(A)$ for nodes in $S_4(A)$ 97
A.2 Illustration of the property 1 100
A.3 Illustration of the property 2 100
A.4 Illustration of the no routing loops proof 102

Chapter 1

Introduction

The Internet has fundamentally changed the way we access information, communicate and interact with each other, purchase goods and services and entertain ourselves. Past two decades have witnessed explosive growth in Internet driven services and devices. Today, not only do we use a variety of diverse set of devices to interconnect and communicate with each other, but also use a wide range of innovative and disruptive applications and services enabled through Internet. These growing number of applications and services (e.g., email, software, etc.) are usually served by huge data centers residing within the core of the Internet, where thousands or tens of thousands of servers are inter-connected with high-speed networks. Similarly, the edges of the Internet are also becoming diverse and mobile: more and more users are accessing the Internet via smart phones, and Internet-ready TVs and other devices being rolled out to the market. In addition, with advances in sensors and sensor network technologies, diverse devices are likely to be connected to the future Internet to enable sensing, actuation and other functionalities, thus integrating the cyber and physical worlds.

To accommodate the demands set by these trends, network providers are rapidly expanding and upgrading the networks. For instance, organizations are now building high speed Ethernet networks, which can transfer data at the rate of 10Gbps, while the work for deploying 40Gbps and 100Gbps Ethernet standard is in progress. Similarly, variety of other wireless communication technologies are being developed such as 802.11 a/b/g/n, 3G, 4G etc. In addition to deploying more advanced communication technologies, network providers are enhancing their existing network topologies by adding a large number of additional and redundant network links for better reliability and availability. Hence we have significantly faster network technologies

and richer network topologies. However, underlying network architecture is being increasingly strained to meet the demands and requirements of these Internet services and their users, including high availability (i.e., "always-on" services), robustness, mobility, manageability, and security. As the universal "glue" that pieces together various heterogeneous physical networks, the Internet Protocol (IP) suffers certain well known shortcomings, e.g., the need for careful and extensive network configurations — in particular, the need for address management, relatively poor support for mobility, and so forth. Despite the potential benefits offered by a larger address space, transition from IPv4 to IPv6 has been difficult and slow; among a variety of other factors, the tight coupling of addressing, routing and other network layer functions clearly make such transition not an easy task. A flurry of "fixes" [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11] have been proposed to address these limitations.

In the following, we provide a brief discussion of several challenges posed by the current networking trends.

1.1 Requirement for Higher Scalability

In order to accommodate the demand set by increasingly large number of Internet driven services (e.g., Email, chat, VoIP, social-networking applications) organizations are continually expanding their existing network infrastructures. The first and the foremost challenge set by these trends is the requirement of higher scalability. Current networks are required to accommodate thousands/Millions of devices. As a result, not only the size of forwarding/routing tables for the edge switches and core routers is exploding, but also the control overhead for underlying routing protocols is becoming increasingly unmanageable. Furthermore, the routing/forwarding table size at the routers and switches is limited by the amount of lookup memory available to them. Similarly, the processing power at routers and switches limits the number of control packets processed by them.

The current networking architecture consists of two key technologies. First is the layer-2 networking technologies, such as Wired/Wifi Ethernet LANs. Ethernet switches usually rely on flooding based forwarding method if the outgoing port for the destination Ethernet address is not already known. They also use a self learning algorithm to avoid this flooding by passively learning the port of connection for each Ethernet address in the network. Clearly, the flooding based forwarding scheme is not scalable as the network size grows. Furthermore, as more and

more number of host-devices connect to the network the forwarding table size at the switch also grows linearly. In particular, each switch tries to learn and store the location of all the host in the network (See Chapter 2). In addition, other Ethernet based protocols such as ARP [12] etc., also rely on flooding, which further leads to scalability concerns and severely limit the size of network both in terms of the number of host-devices and as well as number of Ethernet switches and redundant links used in the network.

The second key technology that plays an integral role in connecting the existing networks is the IP layer (or layer-3) technology. Since, the size of layer-2 networks is limited, usually these layer-2 networks are interconnected using layer-3 networks to create a larger network. Layer-3 uses shortest path based routing protocols, such as OSPF [13], IS-IS [14] etc. to avoid flooding in the data plane, while the control plane used by these routing protocols still relies on the flooding of control packets. A more detailed description of the basic concepts and key mechanisms used by these networks is provided in Chapter 2. These trends and the limitations of existing networking protocols calls for significant improvements in the network architecture which can ensure scalable support to accommodate the large number host-devices and routing nodes.

1.2 Support for Seamless Mobility

The next big challenge facing current networking architectures is the requirement for seamless mobility. The number of mobile and portable devices are rapidly increasing. We use these mobile devices to connect to several Internet driven services “on-the-go”, such as video calling, video-streaming, social-networking etc. Because of these trends, the requirement of seamless mobility support to ensure uninterrupted network connectivity is constantly growing. Furthermore, the on-going trends suggests the increasing use of *virtual computation*, by using *virtual machines* for enabling the cloud driven services. These virtualization based computing technologies not only provide amazing flexibility in managing the physical computing resources, but also enable several attractive features such as *live migration* of virtual machines from one physical host to another. However, the live migration of virtual machines is limited by the mobility support provided by the underlying network connecting the physical host machines. Existing Ethernet/IP technologies can ensure continuous *physical connectivity*, but they do not ensure against the interruption in the network applications running on the host-device. The

main reason behind this interruption is the requirement of reconfiguring the host-devices with the updated IP address, subnet prefix etc. On the other hand, transport and networking layers, usually use the IP address as the host identity in the connection. Therefore, the reconfiguration caused by the network mobility often breaks the on-going connectivity for the applications.

As seen from above, the basic cause for the application/transport level mobility disruption is the *dual use* of IP addresses as the location of the host-device and as well as the end-point identifier. So, when host moves it needs to be re-configured with an appropriate IP address to ensure the connectivity with the network, however, it creates confusion for the applications, as the IP address change is considered as the change in the host identity, and as a result on-going application sessions are terminated. Although, this problem has been known for long [15], recent trends in mobile computing and virtualization are pressing to find the alternate network architectures to sustain the future growth and innovations. Therefore, several solutions have been proposed to address this problem, such as [16, 17, 2, 18, 19, 20, 21].

Clearly, in order to support the growing trends toward mobile computing and cloud enabled services, networks must provide inherent support for seamless support for host-device mobility. The key approach to achieve this would be to enable Location/Identifier split using the underlying networking architecture, as advocated in several recent proposals [16, 17].

1.3 Higher Availability & Reliability

As our reliance on network driven services is growing, the requirement for higher availability and reliability is also increasing. Existing networks may become completely unusable due to failures. The failures may be of several types, for instance, a failure of DHCP server may leave a network unusable, even though the network is physically connected. Similarly, a link or switch failure in an Ethernet network requires the re-computation of the forwarding spanning tree, which in some cases may take more than 50 seconds [22]. On the other hand, intra-domain layer-3 routing protocols such as OSPF may take more than couple of seconds to recover [23]. This time includes the flooding of updated LSA (link state advertisements) packets to all the routers in the network, and finally re-computation of routing tables based on the updated network topology. Similarly, a failure in inter-domain link may cause the interruptions that may last up to several minutes [24, 25].

Most of these problems related to intra-domain networking have their root in the shortest-path-based routing paradigm used in IP (intra-domain) routing. The use of shortest-paths only for routing limits the ability of IP networks to exploit path diversity inherent in the network topology to perform load-balancing, and proactively fast reroute traffic under failures. To perform traffic engineering, one has to resort to sub-optimal workarounds or fixes, e.g., via IGP weight optimization [26,27,28]. Likewise, various IP fast rerouting mechanisms have been proposed [29,30,31,32], which partly circumvent the slow convergence problem [33,34,35,31,36] plaguing the traditional reactive IP routing protocols (e.g., OSPF or IS-IS). Unfortunately most of these fast rerouting mechanisms are fairly complex, and as "add-ons" to existing routing protocols, require additional configurations and further complicate the operations of IP networks [37].

There are several ways to achieve higher availability and reliability. However, the key here is to proactively prepare for the failures, rather than simply reacting to failures. Therefore, a natural support for multi-path routing and fast failure re-routing on top of any arbitrary network topologies using the underlying routing architecture is essential.

1.4 Manageability & Security

Networks are becoming increasingly larger and complicated, therefore their manageability is a big concern. Existing IP networks require careful and often manual configurations and management. For instance, the need for address management is cumbersome and problematic: while an end host joining a network can dynamically obtain its IP address via DHCP, adding a router or subnet to expand an existing network often requires allocation of one or more new IP address blocks. This is because IP address management is link-based (or subnet-based): each link (between two routers), either via a (wired/wireless) point-to-point or broadcast connection — and each subnet must be assigned a distinct IP address block. These address blocks must then be configured manually into routers, and injected into intra-domain routing protocols. When routers are moved from one place to another, their IP addresses must be reassigned. All these tasks must be manually performed by network operators, thus it is often time-consuming and error-prone. Despite the potential benefits offered by a larger address space, transition from IPv4 to IPv6 has been difficult and slow; among a variety of other factors, the tight coupling of addressing and routing as well as other network layer functions clearly make such transition not

an easy task.

Similarly, often organization networks need to build detailed firewalls and security mechanisms to block any malware from entering the network. Again, these firewall rules are closely tied to the IP addresses, and therefore, any IP configuration or device movement leaves these firewalls in an inconsistent state. Any inconsistency in the firewall state may either deny a legitimate device from accessing the network, or introduce a backdoor for potentially harmful traffic to enter in the network.

As suggested in several recent proposals [38,39,40,5,41], the key to achieve better manageability is to create a *logically* centralized control plane to manage the network-wide policy and firewall configuration. Also, assigning persistent identifiers to each host-device in the network, which can be securely verified, would be the key to reduce the dynamic changes in the policy configuration. It would, in turn ensure minimal manual intervention by not requiring operators to constantly update the network configuration whenever any changes occur to the network, and would immensely simplify the network management.

1.5 Summary

While Internet driven applications and devices are growing at a rapid pace, they are continuously stressing the underlying network architecture to meet the demands, which are limited by the scalability of the underlying network technologies. These trends are posing daunting challenges to network designers and operators, who often resort to complicated workarounds to expand their networks and accommodate the demand, which is usually achieved by paying a huge cost in terms of the additional manual overhead in managing and maintaining the infrastructure.

There are several reasons for this stress. On one hand, layer-2 network technologies such as Ethernet are largely plug-&-play and require minimal manual configuration. However, they can not scale to create large and dynamic (layer-2) networks such as, Metro Ethernets, huge enterprise networks, single ISP network, since they rely on flooding based packet forwarding and address resolutions. On the other hand, Internet Protocol (IP layer) which acts as a glue to interconnect several layer-2 networks, require careful and extensive network configurations including the IP address assignments to routers and end-host devices which complicates the network management tasks. They offer relatively poor support for mobility and have limited scalability for the underlying routing protocols due to the state-size explosion and so forth.

To address these challenges we propose a novel network architecture, which not only simplifies the management for large-scale enterprise networks, but also improves the scalability and host mobility of the network to incorporate a large number of routing-nodes (e.g. routers and switches) and host-devices. To enable scalable and robust routing substrate we propose a *Virtual ID ROuting* (VIRO in short) routing framework (See Chapter 3). VIRO takes a paradigm-shifting approach to network routing by completely eliminating the need for flooding from both control and data planes. As we will see in this thesis, VIRO not only enables a highly scalable routing substrate for future networks, but it also provides natural support for many advanced features such as multi-path routing and fast-failure routing. To realize our goals of enabling a scalable and robust network architecture, we present VEIL (Virtual Ethernet Id Layer) in Chapter 4, which essentially addresses the several deficiencies of existing Ethernet networks using VIRO as the core routing protocol. Next, we present VEIL-Click, a real prototype implementation of VEIL using the Click modular router framework [42] in Chapter 5.

Please note that in this thesis we present VIRO as a solution for most of the intra-domain networking challenges, while addressing the inter-domain networking issues is the focus of future work. We propose to use VEIL as an scalable Ethernet architecture for enabling large-scale enterprise networks, metro-ethernets and data center networks.

1.6 Bibliographic Notes

Part of the content of Chapter 3 is from the conference paper, titled as *VIRO: A Scalable, Robust and Namespace Independent Virtual Id ROuting for Future Networks*, which appeared in the proceedings of 30th IEEE International Conference on Computer Communications (IEEE INFOCOM 2011) [43]. The design of VEIL is introduced in the paper, titled as *VEIL: A "Plug-&-Play" Virtual (Ethernet) Id Layer for Below IP Networking*, and appeared in the First workshop on Below IP Networking (BIPN 2009) held in conjunction with Globecom'09 [44], it constitutes a part of Chapter 4, which describes VEIL in detail. Part of the content in Chapter 5 is taken from the paper, titled as *Simplifying Manageability, Scalability and Host Mobility in Large-Scale Enterprise Networks using VEIL-click*, it appeared in the proceedings of the workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE 2011), co-located with USENIX NSDI 2011 [45].

Chapter 2

Background

In this chapter we provide the background for the remainder of the thesis and discuss the motivation behind our work. We explain how existing Ethernet networks work and the key reasons behind their limited scalability. Next, we describe the Ethernet/IP based network architecture commonly used in Enterprise networks. Finally, we provide a quick overview of how link-state routing works, which forms the basis for a variety of intra-domain routing protocols such as OSPF [13] and IS-IS [14].

2.1 Ethernet Network Architecture

Ethernet networks are composed of segments, where each segment comprises of a single physical layer. In case of modern Ethernet networks this segment essentially represents the link connecting a host-device to a port of an Ethernet Switch. Next, Ethernet switches (or Bridges) are used to interconnect these segments together, which creates a layer-2 Ethernet network, commonly known as, LANs (Local Area Network). The complete network is considered as a *single broadcast domain* for packet forwarding and address resolutions.

Basic Setup and Packet Forwarding: Each host-device in an Ethernet network is assigned a unique 48-bit long identifier, known as, MAC address or Ethernet Address. Ethernet switches employ a self-learning mechanism to create a forwarding table, which consists of the mapping between a MAC address and the corresponding output port on the switch. The basic idea behind this approach is the following. If a switch sees an incoming Ethernet frame with source Ethernet address (say, $sourcemac$) as non broadcast address (FF:FF:FF:FF:FF:FF) at a port (say, p) then

it inserts a mapping $\langle sourcemac, p\rangle$ in its forwarding table. This table is populated over time, while the entry expires for the hosts if the switch does not see any packets coming from it in last T seconds. This table is then used for forwarding the packets to destination hosts based on their MAC addresses. So for example, if a switch receives a packet destined to mac, and forwarding table has an entry $\langle mac, p\rangle$, then packet is forwarded through port p at the switch. On the other hand, if no mapping entry is found the packet is flooded through remaining ports on the switch except the incoming port. The same mechanism is used to flood the packets destined to MAC address FF:FF:FF:FF:FF:FF. Similarly, in case of a large network this flooding may lead to significant inefficiencies in the data plane, and a large portion of the link bandwidth may be wasted in carrying redundant Ethernet frames. In addition, each switch tries to learn the mapping for every host in the network, which may lead to extremely large forwarding tables at the switches.

Avoiding Loops using Spanning Tree Protocol: Since Ethernet frames do not carry a TTL (Time-To-Live) value, any loops in the physical topology may allow a frame to be replicated repeatedly, and it may eventually bring down the whole network. To ensure against such scenarios, Ethernet topologies are either required to be completely loop free, or some ports on specific switches are ignored so as to avoid any possible loops in the data plane. The later mechanism is known as the Spanning Tree Protocol [22]. Usually, it works by selecting a root node in the network topology, and computing a spanning tree based on distances to the root. The links that are not present in the tree are ignores and not used for carrying the traffic. If an Ethernet network consists of several redundant links for back-up purposes, then only one of them is used. It leads to a significant waste of resources for richer topologies. Furthermore, any failure in the forwarding spanning tree requires it to be re-computed, which may usually take up to 5-30 seconds.

Although spanning tree helps in avoiding forwarding loops in Ethernet networks, it offers no flexibility in route selection. In particular, Ethernet frame do not necessarily travel through shortest paths, which further wastes the link bandwidth because of the sub-optimal route selection. In addition, it may also lead to skewed load distribution on different links in the topology, where some links may be extremely congested while other links may by under utilized.

Host Address Configuration and Address Resolution Protocol: Each host in a typical Ethernet network use IP addresses to identify each other. Therefore, each host needs to be configured

with appropriate IP address and subnet prefix before it can connect to other hosts in the network. Each host must share the same subnet prefix and mask to communicate with each other. Hosts can either use *static* configuration to achieve this, or may use Dynamic Host Configuration Protocol (DHCP) [46] to perform this *dynamically*. The DHCP based dynamic address configuration often relies on the Ethernet flooding, where configuration request packets are sent to broadcasting address. When, a DHCP server receives this request packet it replies with the appropriate IP address configuration for the requesting host.

While end hosts use IP addresses to identify each other, they need to learn each other's MAC address to actually communicate with each other. Host relies on Address Resolution Protocol (ARP) [12] for this. To achieve this, say, if a host $host_1$ with IP address ip_1 and MAC address mac_1 wants to communicate with another host $host_2$ with IP address ip_2 and MAC address mac_2, then $host_1$ creates an Ethernet frame containing the ARP request packet and sends it to the broadcast Ethernet address. When $host_2$ receives the packet, it checks if the IP address on the ARP request packet is same as its own IP address, and replies back with ARP reply packet destined to $host_1$.

Though DHCP based address configuration and ARP based address resolution play a key role in the simplicity of Ethernet networks, they severely limit the scalability of the Ethernet networks, mainly because of the underlying flooding based mechanisms.

2.2 Interconnecting Ethernet LANs using IP

As seen above, it is not possible to create a large scale Ethernet networks. Therefore, organizations often resort to IP based routers to workaround this deficiency. The basic idea behind this workaround is to create multiple subnets using Ethernet based switches, which are then connected to several Layer-3 devices or IP routers. These routers then limit the broadcast domain for each subnet to within themselves, and allow the networks to grow. In order to avoid any conflicts between individual Ethernet networks or subnets, they are assigned a unique subnet IP address prefix and gateway IP address to connect to other subnets. Each host in a given subnet, must be assigned a unique IP address based on the subnet prefix. Assigning subnet prefixes and associating them with the appropriate interfaces on the routers is usually a manual (or semi-manual) process, where operators must carefully assign subnet prefixes to: a. avoid any conflict across different subnets. b. minimize any future reassignments if more hosts are added to each

subnet over time.

There are several implications of using this hybrid Ethernet/IP architecture in terms of the pros and cons. In the following we provide a brief discussion of these.

Better Data Plane Scalability: Since, IP routers do not use flooding based mechanisms to forward the packets, they are more scalable. Usually, IP routers run more sophisticated routing protocols, such as OSPF, IS-IS etc. These routing protocols essentially use either Link-state or Distance Vector based routing to construct the routing/forwarding tables. This further avoids the inefficiencies caused by the arbitrary path selection made by Ethernet switches.

Control Plane Relies on Flooding: As we will see soon, OSPF and IS-IS routing protocols run link state routing protocol. In case of link state routing, each node first learns its directly connected neighbors by using periodic exchange of HELO packets. After this each node flood their physical neighbor information to all the other nodes in the network in the form of Link State Advertisement (LSA) packets. This flooding of LSA packets further limits the scalability of these networks. As another workaround, operators need to carefully define *OSPF areas*, which limit the control plane flooding domain for LSA packets. However, it adds an additional management overhead, and creates complexities in the network configuration.

Extensive Careful Manual Configurations: As mentioned already, each subnet in the hybrid Ethernet/IP network must be configured carefully, which is done using a manual (or semi-manual) configuration process. Furthermore, any change in the network configuration must be done carefully to avoid any conflicts. In addition, any reconfiguration task requires massive manual overhead, which includes reconfiguring the server and host machines with updated IP address and subnet prefixes, reassignments for the router interfaces, updating network firewalls and QoS rules and updating the DNS mappings to reflect the IP address changes. According to a study [47], 70% of the cost for enterprise network is attributed to the maintenance and configuration related to tasks as opposed to the physical infrastructure cost.

Limited Mobility Support for End Hosts: Since IP networks divide the network into multiple subnets, each end host must be configured with the appropriate IP address based upon the subnet that it is connected to. Because of this, when a host moves from one subnet to another it must be assigned a new IP address based on its new location in the network. This disrupts the on-going network connections for the host, since host level protocols use IP address to identify each other in the network. Furthermore, typically operators use IP address as one of the several tuples to create network wide policies to ensure Quality of Service (QoS), or to

restrict the communication between certain hosts for the security reasons. However, subnet based IP address configuration creates additional complications in the network management. For instance, consider a scenario where a web-server instance runs as a virtual machine on a physical host machine. Now, if the physical host machine needs to be upgraded it may require the virtual machine running web-server instance to be migrated on live to another physical host machine. In case, the new physical host machine is in different subnet, then it will require several changes including: a. the virtual machine needs to be reconfigured with new IP address and subnet prefix, b. DNS mappings should be updated immediately after the move to reflect the new IP address settings. Also, any on-going HTTP session during the move is reset because of the IP address change.

2.3 Link State Routing Protocols

As mentioned earlier, most of the modern intra-domain routing protocols such as, OSPF and IS-IS, are based on Link State Routing. It is one of the two classical approaches for packet switching based network routing. The second approach is based on Distance Vector Routing algorithm. In this section we briefly introduce the basic concepts behind the Link State Routing Protocols [48].

The basic idea behind link-state routing is the creation of network wide link state using the flooding of link state messages. Using this each node constructs (and maintains) a map of the complete network connectivity graph. After this each node independently computes the shortest path to reach all other nodes in the network using the Dijkstra's algorithm [49]. Based on the computed paths a node creates the routing table by using the nexthop node to reach the destination node. This contrasts with distance-vector routing protocols, which work by having each node share its routing table with its neighbors. In a link-state protocol the only information passed between nodes is connectivity related.

Topology	Number of nodes	Number of edges	Average LSAs per node
DC125	125	500	1,892
DC320	320	2,048	7,898
DC500	500	4,000	15,533

Table 2.1: Comparison of link-state advertisement cost.

The computation of shortest paths in link-state routing comes at the cost of network wide flooding in the network, which is essential for each node to construct the complete network connectivity graph. However, the cost of flooding increases exponentially as the network size grows. To illustrate this, let us consider the cost of flooding LSA (Link-State Advertisement) in a fat-tree topology with different number of nodes. In this experiment we consider the fat-tree topology using 125, 320 and 500 nodes. As seen in the Table 2.1 as the network size increases the number of LSAs processed per node also increases sharply.

In order to reduce the overhead caused by the flooding of LSAs, network operators use several workarounds. One such workaround is to define *OSPF Areas*, which limit the LSA flooding domain to given set of nodes, while each area is configured with designated routers to talk to other neighboring areas and directly exchange information with them. However, this results in a manual or semi-manual process with significant planning overhead. Also, depending upon the network topology and areas, it may also result into suboptimal routing by not using shortest paths for source and destination pairs.

Chapter 3

VIRO: Virtual ID Routing

In this chapter we introduce VIRO a novel, virtual identier (Id) routing paradigm for future networks. The key idea in the design of VIRO is to introduce a topology-aware, structured virtual id (vid) space onto which both physical identiers as well as higher layer addresses/names are mapped. VIRO completely eliminates network-wide ooding in both the data and control planes, and thus is highly scalable and robust. Furthermore, VIRO effectively localizes failures, and possesses built-in mechanisms for fast rerouting and load-balancing.

3.1 Introduction

The explosive growth of the Internet has enabled a wide range of diverse devices to interconnect and communicate with each other through a variety of disparate technologies. While serving as the universal "glue" that pieces together various heterogeneous physical networks, the Internet Protocol (IP) suffers certain well-known shortcomings, e.g., the need for careful and extensive network configurations, relatively poor support for mobility, and so forth. In addition, despite the potential benefits offered by a larger address space, transition from IPv4 to IPv6 has been difficult and slow; among a variety of other factors, the tight coupling of addressing, routing and other network layer functions clearly make such transition an uneasy task.

In contrast, layer-2 technologies such as Ethernet are largely *plug-&-play*: hosts are equipped with persistent MAC ad- dresses, and Ethernet switches automatically learn about host addresses and location, adapt to changes in network topology as well as host mobility, and perform packet forwarding seamlessly with minimal operator configuration and intervention.

On the other hand, as it was originally developed for small, local area networks, the traditional layer-2 Ethernet technology can hardly meet the scale as well as the demanding efficiency and robustness requirements imposed on today's large, dynamic networks. For instance, the *network-wide flooding* – often resorted by Ethernet switches to locate end-hosts and forward packets whose locations are yet to be learned – significantly reduces the network capacity. Further, the spanning tree algorithm used to avoid forwarding loops not only results in sub-optimal forwarding paths, but is also slow to adapt to changes in the network topology. To address some of these challenges, several solutions have been proposed, e.g., SEATTLE [2] which employs the OSPF-style shortest routing in layer 2 to build switching tables in Ethernet switches. 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 this chapter we introduce VIRO — a novel, "plug-&-play" *virtual identifier (Id)* routing paradigm for future networks. The objective is three-fold. First, VIRO directly addresses the challenges faced by the traditional layer-2 technologies such as Ethernet, while retaining its "plug-&-play" feature. Second, it provides a uniform *convergence* layer that integrates and unifies routing and forwarding performed by the traditional layer-2 (data link layer) and layer 3 (network layer), as prescribed by the traditional local-area/wide-area network dichotomy. Third and perhaps more importantly, VIRO *decouples routing from addressing*, and thus is *namespace-independent*. This is because under VIRO routing and forwarding are done purely based on *virtual ids* (except possibly at the edge of the network). Hence, VIRO allows new (global or local) addressing and naming schemes (e.g., a flat-id namespace [50]) to be introduced into networks without the need to modify core router/switch functions, and can easily and flexibly support inter-operability between existing and new address schemes/namespaces (e.g., IPv4/IPv6 addresses or flat- id names).

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, structured *vid* space, VIRO employs a DHT[1] -style routing algorithm to build routing tables, look up objects (names, addresses, *vid*'s, etc.) and forward packets. Hence VIRO completely eliminates *network-wide flooding* in both the *data* and

[1] DHT stands for Distributed Hash Table.

control planes. As a result, VIRO is highly scalable and robust, while at the same time offers flexible support for multi-homing, mobility as well as 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 more *diverse* physical devices (of differing layer-2 capabilities) – with relative ease and minimal configuration – to form a large, dynamic and heterogeneous network (as a single autonomous system or domain).

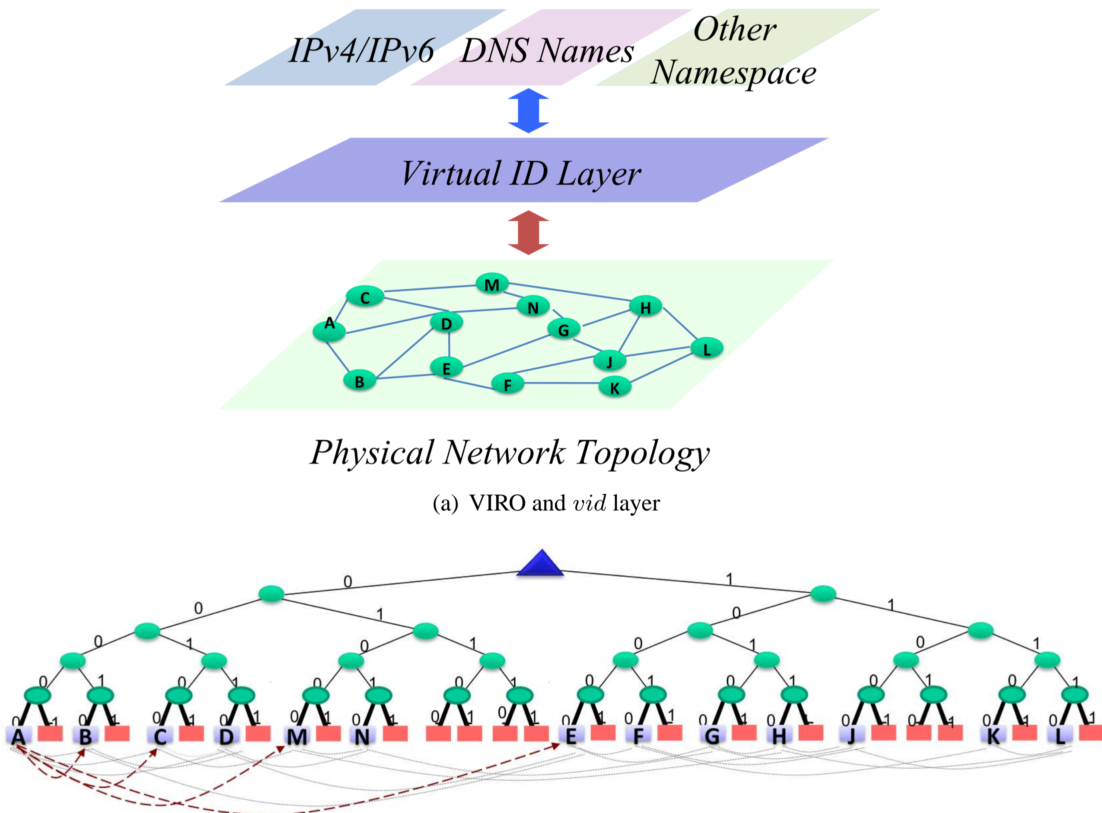
The remainder of the chapter is organized as follows. In Section 3.2 we will provide an overview of VIRO, especially its three key components: *vid space construction and vid assignment*, *VIRO routing*, and *vid -lookup and forwarding*, and then briefly discuss the related work. These key components of VIRO and their basic operations are presented in more details in Section 3.3, Section 3.4 and Section 3.5, respectively. Additional features of VIRO are briefly discussed in Section 3.6. In Section 3.7 we provide detailed simulation based evaluation of VIRO and compare it with other existing protocols. The chapter is concluded in Section 3.8.

3.2 Overview and Related Work

In this section I provide an overview of VIRO, in particular its three key components: *vid space construction and vid assignment*, *VIRO routing*, and *vidlookup and forwarding*. I also include a summary of the terminologies and notations to be used in the subsequent sections. The section is concluded with a brief discussion of the related work.

3.2.1 Design of VIRO: An Overview

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, see Fig.3.1(a) for an illustration. By *topology-aware*, I mean that the physical network topology, as formed



(a) VIRO and *vid* layer

(b) *vid* space as a *virtual binary tree*: the grey dotted lines denote physical connectivity, the red dashed lines indicate the logical neighbors at *different* levels of node A; the red boxes represent the *unused* *vid*'s.

Figure 3.1: Overview of VIRO.

by the connections among "routing-nodes"[2] or VIRO switches, is *embedded* into a *structured* space, e.g., a Kademlia-like virtual tree [51], a hypercube, a d-dimensional Euclidean space, in such a manner that *physical proximity among VIRO switches are approximately preserved.* For the physical network topology shown in Fig.3.1(a), Fig. 3.1(b) shows such an embedding using a Kademlia-like [51] *virtual binary tree*, where only the *leaf* nodes correspond to physical (switching/routing) devices, i.e., *VIRO switches* (or (VIRO) nodes), whereas all intermediate

[2] In this thesis I refer to a "routing node" (such as router/switch/ wireless access points) simply as "node" or a VIRO switch. We use the term *end-host* or simply *host* to represent the end host-devices that connect to the network. E.g. Laptops, smartphones etc.

nodes in the virtual binary tree are logical (thus the term *virtual* binary tree!), representing all the VIRO switches residing within its subtree. Please note that, as such, it does not make sense to talk about the failure of an intermediate node! Hence unlike a physical tree (e.g., as used in Ethernet spanning tree), failure of any node in the network only affects a leaf node in the (virtual) tree. Namely, there is no single (*intermediate*) point of failure.

The topology-aware, structured *vid* space is constructed at the network *bootstrapping* phase, i.e., when the network is being set up. Let $\mathcal{L}$ denote the number of bits used to represent the vid space (or the depth of the virtual binary tree). The *vid* of a (leaf) node is the $\mathcal{L}$-bit long binary string along the path from the root to the corresponding leaf node. The *logical distance* between a pair of vids in this vid space is defined as $\mathcal{L}$ *minus the length of the longest common prefix for the pair* (see Eq.(3.1) for a formal definition). Hence it is clear that all routing nodes within the same sub-tree (thus with logically closer vid's) are also physically closer. In Sec. 3.3 I will describe how the vid space construction and *vid* assignment can be performed in either a centralized or distributed fashion during the bootstrapping phase. Once the *vid* space has been constructed, when a new node (VIRO) switch joins the network, its *vid* will be assigned based on the subtree (and its neighbors in the subtree) that it is attached to. For end-hosts, their *vid*'s are always (dynamically) assigned at the time when they join the network: when an end-host is attached (either via wired or wireless link) to a node (VIRO switch) in the network, it is assigned an (*extended*) vid consisting of the ($\mathcal{L}$-bit) vid of the switch plus a (randomly assigned) l-bit local id (see Sec. 3.3 for detail). The VIRO node/switch that an end-host is attached to will be referred to as its *host-node*. Hence the *vid*'s for all end-hosts attached to the same host-node share an $\mathcal{L}$-bit prefix, and thus they are at 0 logical distance from each other.

Taking advantage of the topology-aware, structured vid space, *VIRO routing* employs a Kademlia-like, DHT-style routing algorithm to build routing tables at each node so as to maintain network-wide connectivity and perform end-to-end data delivery. However, unlike the peer-to-peer (P2P) Kademlia routing protocol which operates on top of the IP network layer and thus the end-to-end connectivity is assumed (and maintained by the underlying network layer), VIRO has to build network-wide connectivity in a "bottom-up" manner based on the (native) link layer connections. In VIRO, routing tables are constructed *piece-meal-wise* using the vid logical distance instead of physical distance (e.g., hop counts). As in any DHT routing algorithm, a "pub-sub" (or "publish-&-query") mechanism is used by each node to publish and query (the so-called *rendezvous points*) relevant routing information to build routing entries at

each level using a "round-by-round" bottom-up procedure (see Section 3.4). As a result, VIRO completely eliminates *network-wide flooding* in both the data plane (unlike Ethernet switching algorithm) and *control plane* (unlike OSPF and other shortest path routing algorithms). Furthermore, because of the natural *hierarchical* structure of the vid space, routing information regarding far-away part of the network is automatically aggregated using the vid prefixes. Hence the routing table size is on the order of $O(\log N)$, where N is the number of nodes in the network, as opposed to $O(N)$ (as in the case of OSPF). Unlike OSPF, in VIRO no *network-wide* full topology needs to be maintained by any switch, thanks to the structured vid space, and hence changes in network topology do not need to be flooded globally. Due to the aggregate routing information maintained by switches, failure of a link or switch node can be *localized*, without affecting nodes in far-away parts of the network. Furthermore, path and topology diversity can be easily exploited in VIRO by using multiple gateways; hence failure of one gateway does not affect network-wide reachability.

The third major component of VIRO is *vidlookup and forwarding*. VIRO maps both lower-layer (native network) physical addresses (e.g., MAC addresses) and higher-layer addresses/names (e.g., IPv4/IPv6, flat-id names, etc.) onto the *vid* space. Routing and forwarding *between VIRO nodes/switches* is performed using *vid*'s; only at the network *edge* are the physical addresses/logical addresses/names needed (between a VIRO switch and an end-host) to locate individual end-hosts, or data/services to be delivered. VIRO performs address/name resolution and *vid* look-up by building the (standard) DHT look-up mechanisms on top of the same *vid* space. For example, either one-hop or multi-hop DHT may be used for this purpose, depending on the speed, memory or other design/performance considerations. Section 3.5 provides more details regarding how name/address resolution, *vid* look-up and forwarding operations are performed.

By *decoupling routing from addressing* and confining address/name resolutions to the edge of networks, VIRO is *namespace-independent*. It is in a sense "future-proof": VIRO allows new (global or local) addressing/naming schemes, e.g., a flat-id namespace [50], to be introduced into networks without having to modify core network routing/forwarding functions. Furthermore, data can be addressed, accessed and delivered *across multiple namespaces*, for example, from a user with an IPv4 address to another user with an IPv6 address, while the information or service may be named using a DNS-based URL or a "flat-name" as defined

in [50]. In other words, VIRO allows multiple namespaces to co-exist within the same network substrate, and route/forward data across multiple namespaces by mapping these service-specific names/addresses (and "native" physical network addresses, e.g., MAC addresses) to *vid*'s. Hence inter-operability between existing and new addressing schemes/namespaces can be flexibly supported under VIRO. VIRO also provides seamless and efficient support for multi-homing and mobility. It allows an end-host to be connected with multiple VIRO nodes without causing loops and other complexities. Further, mobility of an end-host can be easily supported through scalable and efficient address/name resolution and *vid* lookup. By controlling the name/address resolution operations, VIRO enables access control with ease. Some of these and other additional features and capabilities of VIRO are briefly discussed in Section 3.6. In summary, 1) VIRO is highly scalable, robust; 2) it decouples routing from addressing, and thus is namespace-independent; 3) it provides seamless and efficient support for multi-homing, mobility and access control; 4) VIRO localizes failures, and provides *built-in* mechanisms for fast rerouting and load-balancing; and 5) 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 or service isolation.

3.2.2 Notations & Definitions

In the following I define and list the key notations and terminologies that will be used to describe VIRO in the remaining sections. These notations are summarized in the Table 3.1

Notation	Description
$vid(x)$	Virtual id (vid) of a node x
$pid(x)$	Physical/persistent address/name or other lower/higher layer identifier of a node x
$d(x,y)$	Shortest physical hop distance between nodes x and y
$\delta(x,y)$	Logical distance between nodes x and y
$vid_k(v_x)$	First k bits(from left) of $vid(x)$
$lcp(v_x,v_y)$	Length of the longest common prefix for v_x and v_y
$S_k(x)$	$\{y : \delta(x,y) \leq k\}$ kth sub-tree of node x
$B_k(x)$	$\{y : \delta(x,y) = k\}$ kth bucket of node x
$rdv_k(x)$	kth level Rendezvous points for node i
$R_k(x)$	Reachability information for node x's bucket $B_k(x)$
$hash_k(val)$	Hash function to get k bit hash value for val

Table 3.1: Summary of notations

Logical Distance ($\delta(x,y)$): The logical distance between any two nodes say x and y in an $\mathcal{L}$-bit

vid space is defined as:

$$\delta(x, y) = \mathcal{L} - lcp(vid(x), vid(y)). \tag{3.1}$$

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 $\mathcal{L} = 4$ then $\delta(x, y) = 4 - lcp(vid(x), vid(y)) = 4 - 1 = 3$

Bucket $B_k(x)$: For a given node x, the kth *bucket*, $B_k(x)$, is the set of nodes which are at 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 a level k, $rdv_k(x)$, is a node in the sub-tree $S_{k-1}(x)$, which stores the connectivity information to reach its k-th bucket $B_k(x)$. It is the node which is closest[3] 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 sub-trees 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 I 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/IPv4 addresses, persistent name (e.g., a flat-id name) or other addresses/names that 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

[3] In the context of rendezvous point and access-node I define the closeness using the xor distance [51] between the hashed key and actual node vid, this is to ensure the uniqueness of Rendezvous points and Access-nodes.

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 following information about the reachability to a given bucket $B_k(i)$ for node i. We denote this by $R_k(x)$, and it consists of following values:

a) Bucket level k

b) vid prefix in $B_k(i)$ that is reachable using this entry.

c) *Nexthop* to reach any node in this bucket.

d) Logically closest gateway to reach this $B_k(i)$ prefix.

Routing Table: The routing table for a node x consists of the reachability information for all the buckets $B_k(x)$ for x. It is shown in Table 3.2.

Bucket	Prefix	Nexthop	Gateway
1	pfx_1	nh_1	gw_1
2	pfx_2	nh_2	gw_2
...	...	...	...
$\mathcal{L}$	$pfx_{\mathcal{L}}$	$nh_{\mathcal{L}}$	$gw_{\mathcal{L}}$

Table 3.2: Routing table for a node in VIRO

3.2.3 Related Work

Closely related to our work, SEATTLE [2] focuses primarily on addressing the scalability issues of Ethernet. As pointed out earlier, while SEATTLE eliminates data plane flooding, it employs OSPF-like shortest path routing, which requires *network-wide flooding* of link state advertisements (LSAs) in maintaining network topology and tracking its changes. SEATTLE thus suffers the same limitations plaguing OSPF-based IP routing: for example, it is limited to the use of shortest paths; load-balancing and fast rerouting can be messy to implement. In contrast, VIRO avoids these inherent problems in shortest-path routing. It is far more scalable and robust (e.g., with $O(\log N)$ routing table sizes instead of $O(N)$ in OSPF).

Our work is also substantially different from the "flat-id" based routing schemes such as VRR [52], UIP [20] and ROFL [19], which advocate a *flat* universal id space to replace the current global IP address space. These schemes employ a DHT-style randomly and consistently hashed id assignment–which produces an id-space completely independent of the underlying network topology–and perform routing based on logical distance to the id of the destination,

incurring a stretch penalty (which is unbounded in the worst case). In addition, link/node failures and node dynamics (node joining, leaving or moving around in the network) often induce a network-wide effect, as two logically close nodes may be far away in the underlying physical network. By introducing a *topology-aware*, structured vid space, VIRO circumvents these problems: it incurs fairly small routing stretches, and effectively localizes the effect of failures. More importantly, VIRO is *namespace-independent*, allowing any namespace to be used, be it hierarchical or flat, and supporting inter-operability across namespaces.

3.3 Virtual Id Assignment

In this section we describe the mechanisms to create topology-aware and structured *vid* space. In particular, we describe how routing-nodes are assigned *vids* and how the *host-devices* that connect to the network get their *vids* assigned.

The initial *vid* space construction and *vid* assignment is performed at the network bootstrapping process when the network is being set up. As mentioned earlier, the *vid* space is structured in a Kademlia-like virtual binary tree (with a depth of $\mathcal{L}$), where each node (VIRO switch) resides at a leaf node of the tree, and is assigned an $\mathcal{L}$-bit vid corresponding to the bit-string from the root to this leaf node. More precisely, each interface of a VIRO switch/node is assigned a *vid*; hence a VIRO node may have multiple *vid*'s, as in today's IP networks. The parameter $\mathcal{L}$ can be configured at the bootstrapping phase, or a default value (say, $\mathcal{L} = 32, 48, 64, 128$) may be used, depending on the target network size or other design considerations. In general I assume that $2^{\mathcal{L}}$ is far larger than the actual size of the network, and thus many leaf nodes will be un-occupied; furthermore, some subtrees may be entirely empty. In particular, during the initial *vid* construction and assignment phase, it is ensured that each (non-empty) subtree has sufficient empty leaf nodes to accommodate new node joins in the future. After the network is set up (and the initial *vid* assignment completed), the *vid* assignment for a new VIRO node (switch) that subsequently joins the network is done based on its location and the *vid*'s of its physical neighbors, and the (extended) *vid* for an end-host that joins the network is assigned by its host-node (to which it is attached). We describe these operations in more detail below.

3.3.1 vid Space Construction at Bootstrap

During the vid construction (and subsequent addition/deletion of nodes), two key *invariant* properties are always maintained: i) The *closeness* property: if two nodes are close in the vid space, then they are also close in the physical topology; in particular, if $\delta(x, y) = 1$ (i.e., nodes x and y are *logical* neighbors, then they must be (physically) directly connected. ii) The *connectivity* property: any two *logically adjacent* (i.e., sharing a common prefix) (and *non-empty*) sub-trees must be physically connected, i.e., there must be at least one physical (wired/wireless) link connecting one node in a sub-tree to another node in the other sub-tree. We have designed two modes of vid space construction and vid assignment for bootstrapping a VIRO network: a *centralized* algorithm and a *distributed* algorithm. Both algorithms guarantee that the constructed *vid* space satisfies these two properties.

The centralized mode is designed for networking environments (e.g., ISP, large campus/enterprise or data center networks) where the (at least the initial) topology is *pre-planned* and thus known *a priori*. Given the topology, the centralized vid assignment algorithm employs a *top-down* approach to assign vid's by starting from the root of the virtual binary tree: it recursively partitions the network topology into two (*connected*, with possibly overlapping boundary nodes) subgraphs, and appends 1-bit to the (already assigned) vid prefixes of the nodes in each subgraph. (See Algorithm 1)

Algorithm 1 vid assignment using top-down graph-partitioning approach

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**

The distributed mode is more suitable for networking environments (e.g., home or small office networks, wireless ad hoc networks) where networks are set up in a *piecemeal*, *unplanned* or *ad hoc* fashion. The distributed *vid* assignment algorithm employs a *bottom-up* approach to assign *vid*'s by starting from the leaf nodes (namely, the lower-level *vid* bits (a suffix) are

determined first, and higher-level bits are recursively assigned). As illustrated in Fig. 3.2, VIRO nodes first discover their physical neighbors (e.g., via an OSPF-like adjacency discover protocol, or *local* broadcast), and collaboratively run a recursive clustering algorithm (similar to those used in wireless ad hoc networks, see, e.g., [53]) to construct a virtual binary tree.

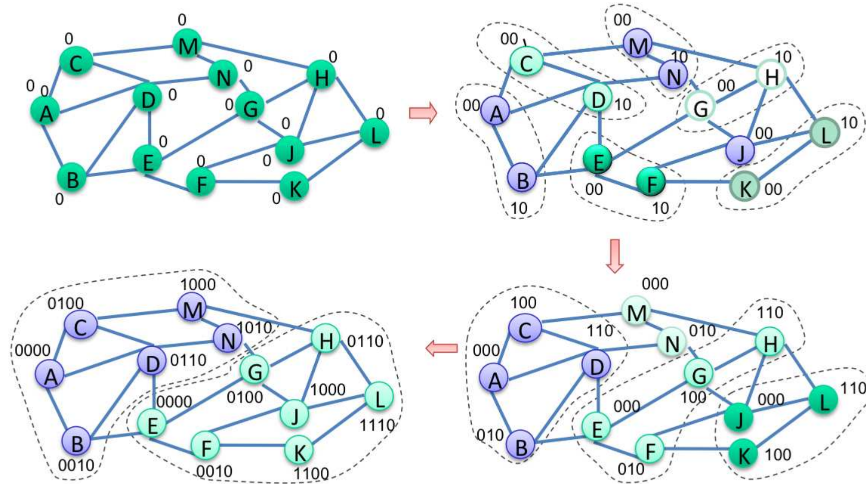


Figure 3.2: Virtual ID assignment process using bottom-up clustering method.

3.3.2 *vid* Assignment upon New Node Join

After the network is set up (and the initial *vid* assignment completed), when a new VIRO node (switch) joins the network, its *vid* is assigned based on its location and the *vid*'s of its physical neighbors. More specifically, it picks one of the *unused vid*'s that are closest to one of its physical neighbors, thus joining a subtree (of an appropriate level) of this neighbor. Using Fig. 3.2 as an example, suppose a new node joins the network and is connected to both node A and node B. It can pick one of the two unused vid's: $\{00001, 00011\}$, joining either the level-1 subtree of node A or node B (see Fig. 3.1(b)). As mentioned in the beginning of this section, we assume that in general $\mathcal{L}$ is configured in such a manner that there will be sufficient empty slots under each subtree to accommodate new node joins. In the (extremely rare) case where $\mathcal{L}$ is ill-chosen initially, one could possibly expand $\mathcal{L}$ by appending additional bits (as long as there will be header space). In our design, I reserve a 128-bit header space for $\mathcal{L}$, while allowing $\mathcal{L}$ to be configured with a smaller value (i.e., only the first $\mathcal{L}$ bits are used). This

provides the advantage of using a smaller $\mathcal{L}$ (thus fewer routing table entries per node) as well as the flexibility to expand $\mathcal{L}$ if needed.

3.3.3 Host *vid* Assignment

Figure 3.3: vid structure for the hosts.

In VIRO, the *vid* for end-hosts comprises two parts (see Fig. 3.3): the $\mathcal{L}$-bit *host-node vid* part identifies the host-node (VIRO switch) that an end-host is directly connected to, i.e., the *vid* of its host-node; the second l-bit *host vid* part identifies the end-host, and distinguishes it from other end-hosts attached to the same host-node. This l-bit host *vid* part is selected randomly by the host-node, e.g., via a (l-bit) random hash function, $hash_l(pid(h))$, using a $pid(h)$ (e.g., MAC or IP address) of the end-host, to ensure local uniqueness. The ($\mathcal{L} + l$)-bit (expanded) *vid* thus uniquely identifies each end-host in the network. When an end-host moves from one part of the network to another (thus attaches itself to a different VIRO node), its *vid* is also re-assigned. The host-node publishes and maintains pid-vid mappings of end-hosts attached to it, see Section 3.5.

3.4 VIRO Routing

Routing in VIRO is inspired by Kademlia-like [51] DHTs. The key difference is that DHTs assume end-to-end connectivity, and utilize the underlying IP-layer for routing and look-up operations, whereas VIRO must build end-to-end connectivity by itself, using the (local) physical connectivity among VIRO nodes. In the following we describe how this is done in VIRO, and once the routing tables are constructed, how forwarding is performed. We will also briefly discuss how failures are handled by VIRO.

3.4.1 Overview and the Routing Invariant Property

Similar to Kademlia (and other DHT routing algorithms), each node in VIRO maintains a routing table with (at most) $\mathcal{L}$ entries, one per level of the (virtual binary) tree. Given an arbitrary

(VIRO) node x in the network, all other nodes in the network fall within one of $\mathcal{L}$ buckets, $B_k(x)$, $1 \le k \le \mathcal{L}$. For some k, $B_k(x)$ may be empty, i.e., there are no VIRO switches residing at the corresponding leaf nodes. The VIRO routing tables (as in any DHTs) are constructed in such a manner that for any non-empty $B_k(x)$, $1 \le k \le \mathcal{L}$, 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 (e.g., via y) *any* node within $B_k(x)$. Hence each node x in VIRO only needs to maintain $\mathcal{L}$ route entries, where the *level-k routing entry* maintains some routing information regarding how to reach a node in $B_k(x)$, $1 \le k \le \mathcal{L}$; if $B_k(x)$ is empty, then the level-k routing entry is null.

Take the network in Fig. 3.1(b) as an example. Representing each level of routing entries by the *vid* prefix (of nodes within the corresponding bucket), the routing table at node A (with $vid_A = 00000$) contains 5 routing entries: the level-1 ($\{00011\}$) routing entry is empty (no node in $B_1(A)$); the level-2 ($\{0001*\}$) routing entry would contain B, the only node in $B_2(A)$; the level-3 ($\{001**\}$) routing entry would contain either C or D or both ; the level-4 ($\{01***\}$) routing entry would contain either M or N, or both; and the level-5 ($\{1****\}$) routing entry would contain (at least) one of the 7 nodes in $B_5(A)$ (i.e. E, F, G, H, J, K, L).

Unlike standard DHTs where end-to-end connectivity is assumed, VIRO must build up end-to-end connectivity by itself. This is done through a *bottom-up* procedure, where for $k = 2, \ldots, \mathcal{L}$, the first $k-1$ routing entries are constructed before the level-k routing entry is built. If there is a node, say, y, resides within $B_1(x)$, then by the *closeness* property of the *vid* space, node x and node y must be directly connected (as $\delta(x, y) = 1$). Hence the level-1 can be trivially built. More generally, if $B_k(x)$ is not empty, then by the *connectivity* property there must exist another node, say y, within $S_{k-1}(x)$ that is physically connected to a node, say z, in $B_k(x)$. Hence as long as x can reach y, then via y it can reach $z \in B_k(x)$, thus any node in $B_k(x)$. Node y is thus referred to as a (level-k) *gateway* to $B_k(x)$. Since $y \in S_{k-1}(x)$ (thus $\delta(x, y) = k-1$), y is contained in one of $k-1$ buckets of x, i.e., $y \in B_{k'}(x), k' < k$. In other words, once the first $k-1$ routing entries have been constructed, the gateway node y to $B_k(x)$ can be reached using these first $k-1$ routing entries. Using y, we can build the level-k routing entry to reach any node in $B_k(x)$. This leads us to the following *invariant* property, referred as the *routing Invariant Property* that must be satisfied by any VIRO routing table construction algorithm. In the next subsection we present one such algorithm for constructing VIRO routing tables.

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 $1 \leq k \leq \mathcal{L}$,

$$\exists z \in B_k(x) \wedge \exists y \in S_{k-1}(x) \text{ such that } (y, z) \in E. \tag{3.2}$$

Then using y as a level-k gateway (namely, including y as a gateway in x's k-level routing entry) would guarantee node x to reach any node in $B_k(x)$.

3.4.2 Routing Table Construction Algorithm

VIRO employs a *bottom-up, round-by-round* procedure, starting with round $k = 1$ to $\mathcal{L}$, to build routing tables that satisfy the above invariant property. At each round k (for building level-k routing entry), each node x needs to discover a level-k gateway that satisfies Eq.(3.2) so as to reach nodes in $B_k(x)$. We employ a *publish-&-query* based method for this discovery process, thereby completely eliminating *network-wide (control plane) flooding*. Before this round-by-round procedure is invoked, each node x first runs a *local* physical neighbor discovery protocol (e.g., via HELO messages, or local broadcast) to discover its (physically) directly connected neighbors. (Note that these (physical) neighbors may reside in any $B_k(x)$, $1 \leq k \leq \mathcal{L}$.) At the first round ($k = 1$), if $B_1(x)$ is not empty, then any node $y \in B_1(x)$ must be a direct (physical) neighbor of node x. Hence the level-1 routing entry is constructed trivially based on node x's locally discovered physical neighbors.

More generally, suppose the first k routing entries have already been constructed at node x. In round $k + 1$, if node x is directly connected to node $z \in B_{k+1}(x)$ (as discovered during the local physical neighbor discovery process), node x is then a level-($k + 1$) gateway (for nodes within $S_k(x)$). Besides installing itself as a gateway in its own level-($k + 1$) routing entry, it declares (to other nodes within $S_k(x)$) that it is a level-($k + 1$) gateway by *publishing* this information (i.e., $\langle$ level-($k + 1$) gateway, $vid(x)\rangle$) to the level-($k + 1$) *rendezvous point(s)* within $S_k(x)$. Depending on the level k and robustness requirement, one or multiple rendezvous points (say, k rendezvous points per level k) may be used, and they are selected based on certain *consistent* rules. We use $rdv_{k+1}(x)$ to denote a level-($k + 1$) rendezvous point which is responsible for maintaining the level-($k + 1$) gateway information. If node x is *not* directly connected to a node in $B_{k+1}(x)$, it discovers and learns about a level-($k+1$) gateway by sending a query to one of the rendezvous points, $rdv_{k+1}(x)$. The rendezvous point then replies with one of the (published) level-($k + 1$) gateways. Node x thus constructs its level-($k + 1$) routing

entry, and moves on to the next round until all $\mathcal{L}$ routing entries have been built. Using a level-k gateway, each node x can recursively look up its routing table (using first $(k-1)$ routing entries only) and discover the *next-hop* (a directly connected neighbor) to reach the gateway. The basic algorithm is described in pseudo-code in Algorithm 2. We note that the local physical neighbor discovery process as well as the publish-query-based routing process are performed *periodically* by each node. In other words, each node periodically updates its routing entries by periodically publishing and querying gateway information.

Algorithm 2 Constructing routing tables for node i

```
1:  S_0(i) := i, B_0(i) := i;
2:  Input: N_1(i) ← 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 δ(i,x) = k then
6:        R_k(i) := (BucketDistance = k, Prefix = pfx_{L−k}(i), NextHop = x, Gateway = i)
7:        Publish (rdv_k(i), edge(x ↔ i))
8:     else
9:        gw_k := Query (rdv_k(i), k,i)
10:       if gw_k is not Nil then
11:          d := δ(gw_k, i)
12:          nexthop_k := R_d(i).nexthop
13:          R_k(i) := (BucketDistance = k, Prefix = pfx_{L−k}(i), NextHop =
             nexthop_k, Gateway = gw_k)
14:       end if
15:    end if
16: end for
```

Again using the network in Fig. 3.1(b) as an example, in the following we illustrate how routing tables at node A can be constructed using Alg. 2. First, during the local physical neighbor discovery process, node A discovers its three physical neighbors, $B \in B_2(A)$, $C \in B_3(A)$ and $D \in B_3(A)$. Using the information about its local physical neighbors, in round 1, A constructs a *null* level-1 routing entry. In round 2 and round 3, A constructs its level-2 and level-3 routing entries by entering itself as the gateway, and also publishes itself as the level-2 and level-3 gateways (to reach $B_2(A)$ and $B_3(A)$ respectively). To build its level-4 routing entry, A queries a level-4 rendezvous point and discovers a level-4 gateway, say, node C (which is connected to node $M \in B_4(A)$). It installs C as its level-4 gateway (which is also the next-hop to reach C itself). Similarly in round 5, A queries and discovers a level-5 gateway, node B (which is connected to node $E \in B_5(A)$), and installs it in its level-5 routing entry. We show

the routing-table for node A at the end of each round in Table 3.3.

Last but not the least, we note that in order to ensure no routing loop is formed during the construction of routing tables, the *gateway selection* process must be *consistent*. In other words, when a node x queries a level-k rendezvous point $rdv_k(x)$ to discover a level-k gateway, the rendezvous point $rdv_k(x)$ – which may have learned multiple gateways, say, four level-5 gateways, B, D, M, and N, to reach $B_5(A)$ – cannot select an *arbitrary* gateway and return it to node x (see Appendix A for an example where such a strategy for causing a routing loop). Assuming that only one (default) gateway is installed in the routing table of each node, a simple consistent strategy is to always select one of those gateways whose vid is closest to the vid of the querying node. Under this selection rule, in the case of node A, node B will be selected as the level-5 gateway for node A to reach $B_5(A)$. More generally, when multiple gateways may be installed in the routing tables, and used, say, for load-balancing or fast rerouting (see Section 3.6), a generalized consistent gateway selection rule can be devised by associating with each level-k gateway a special *forwarding directive* (a $\mathcal{L}$-bit key, whose first $\mathcal{L} - k$ bits are the same as those in the vid of the querying node, is associated with the level-k gateway whose *vid* is closest to this key). When this (level-k) gateway is selected to reach $B_k(x)$, the associated forwarding directive is also included in the packet header to direct subsequent packet forwarding toward this gateway. We formally prove that *when either the simple or the generalized consistent gateway selection rule is used, loop-free routing/forwarding is guaranteed.* in Appendix A.

3.4.3 Bridge

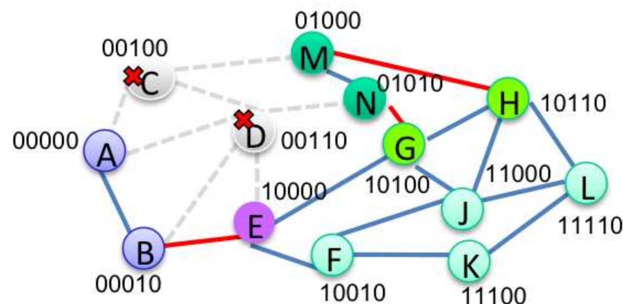


Figure 3.4: Violation of *logical connectivity constratint* on vid assignment

Furthermore, we provide an extended algorithm for constructing routing tables in the presence of *logical connectivity constraint* violations on vid-assignment constraint (3.2). In such cases two nodes which are at a logical distance of k may be connected through some nodes which are at a logical distance of $k' > k$ from both of them. To deal with these scenario we introduce the concept of *Bridge*. A bridge at kth level is defined as follows:

$$\exists p_k \in B_k(x) \wedge \exists p_{k'} \in S_{k'}(x) \wedge (p_{k'}, p_k) \in E$$

Here $1 \leq k' \leq k + r, 0 < r < l$, $p_{k'}$ is the bridged-gateway and $(p_{k'}, p_k)$ is the bridge.

For example consider the scenario shown in Fig. 3.4, in this case due to the failures of nodes C and D, nodes in $S_4(A)$ are not physically connected through nodes in $S_4(A)$ only. E.g. nodes A and M ($\delta(A, M) = 4$) are now connected through nodes E and G, which are at the logical distance of 5 from A and M. In this case nodes $G \in B_5(A)$ and $H \in B_5(A)$ will act as *bridged-gateway*s to reach bucket $B_4(A)$ for node A.

In order to learn bridge information, we modify the Algorithm 2. Algorithm 3 shows the modified algorithm. Similarly, Algorithm 4 shows the modified query and packet forwarding functions using bridges. Now, during the kth round nodes perform following two tasks:

1. Learn the connectivity information to reach nodes in $B_k x$ by querying the Rendezvous Point $rdv_k(x)$.

2. For all the Buckets $B_{k_1}(x)$ that were unreachable in round k(here $k_1 < k$), search for a bridge by querying the Rendezvous Points in $B_{k_2}(x)$ to which it is already connected for the connectivity to reach $B_{k_1}(x)$ ($k_1 < k_2 \leq k$).

In the case of bridges if some node in $B_k(x)$ provides connectivity for the locally disconnected sub-tree[4] in the set $S_{k-1}(x)$ then all the nodes in $B_k(x)$ store separate routing entries to reach each of the sub-tree.

To illustrate the routing table construction using the modified algorithm consider the topology shown in the Fig. 3.4. As seen in this figure, nodes $\{A, B\}$ and $\{M, N\}$ need bridges to reach each other. Next we demonstrate how A, M and J update their routing tables to include the bridge information. At the end of round-5 all the nodes will learn the connectivity information to reach nodes upto logical distance of 5, except for the *logically disconnected subtrees*

[4] Locally disconnected buckets is a pair of buckets say $S_k(x)$ and $S_k(y)$ such that $\delta(x, y) = k + 1$ and there is no path between $x \rightsquigarrow y$ through nodes in $S_k(x) \bigcup S_k(y)$

Algorithm 3 Constructing routing tables using bridges 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 $l - 1$ **do**
5:   **if** x in $N_1(i)$ and $\delta(i, x) = k$ **then**
6:     $R_k(i) := (BucketDistance = k, Prefix = pfx_{l-k}(i), NextHop = x, Gateway = i)$
7:     **Publish** $(rdv_k(i)$, edge$(x \leftrightarrow i))$
8:     $List(gw_k, prefix) :=$ **Query** $(rdv_k(i)$, k,$i)$
9:     **for** $(gw_k, prefix)$ in $List(gw_k, prefix)$ **do**
10:       $d := \delta(gw_k, i)$
11:       $nexthop_k := R_d(i).bestmatch(gw_k).nexthop$
12:       $R_k(i) := R_k(i) \bigcup (BucketDistance = k, Prefix = prefix, NextHop = nexthop_k, Gateway = gw_k)$
13:     **end for**
14:   **end if**
15:   **for** $k' = 1$ to k **do**
16:     **if** $R_{k'}(i)$ is Empty **then**
17:       **for** $r = k' + 1$ to k **do**
18:         $y \in B_{k'}(i)$ and $z \in B_r(i)$
19:         Query r^{th} level rendezvous point in $B_r(i)$ for an edge $y \leftrightarrow z$
20:         **if** $\exists z$ **then**
21:           $R_{k'}(i) := (BucketDistance = k', Prefix = pfx_{l-k'}(i), NextHop = nexthop(z), (bridged)Gateway = z)$
22:           Break
23:         **end if**
24:       **end for**
25:     **end if**
26:   **end for**
27: **end for**

($\{A, B\}$ and $\{M, N\}$) We show the routing tables for nodes A, M, J at the end of round-5 in Table 3.4.

In the next round nodes in sub-trees ($\{A, B\}$ and $\{M, N\}$) will try to form bridges to reach each other by querying level-5 rendezvous point (say node F) for nodes in their level-5 bucket (i.e. $\{E, F, G, H, J, K, L\}$). F will have following edges in its rendezvous table to reach level-5 bucket: i) $E \leftrightarrow B$, ii) $H \leftrightarrow M$, iii) $G \leftrightarrow N$ When node A query F to reach its logically disconnected sub-tree (01***), F will search for an edge connecting nodes $y \leftrightarrow z$ such that $y \in S_4(F)$, $z \in B_5(F)$ and $vid_2(z) = 01$. It finds that there are two edges i) $H \leftrightarrow M$ and ii) $G \leftrightarrow N$, and returns H(randomly selected) as *bridged-gateway* to A. Furthermore, F will also update nodes in its level-4 sub-tree ($S_4(F)$) to use different gateways for each prefix 01***

(gateways: G, H) and 00**** (gateway: E). Similarly when M queries for bridge to reach its logically disconnected sub-tree (00***) F returns E as a gateway. We show the final routing tables for A, M and J in Table 3.5.

3.4.4 Basic Forwarding Algorithm

Given the routing tables constructed above, forwarding using vid is fairly straightforward. Consider a node x which wants to send a (*normal data*) packet to a node with $vid = dest$. Node x forwards the packet directly to $dest$ if it is one of its physical neighbors. Otherwise, it computes the logical distance $k = \delta(x, dest)$, and sends the packet to the next-hop as indicated in its level-k routing entry. (If its level-k routing entry is null, it implies that $B_k(x)$ is empty, hence node x simply drops the packet). Upon receiving this packet, the next-hop performs the same operation to determine its next-hop for forwarding the packet. In case multiple gateways are used, the forwarding operation is slightly more complex, where both the *vid* of the destination as well as the forwarding directive are used to select (the gateways and their corresponding) next-hops. See Section 3.6 for details. When the packet is a control packet, e.g., *publish*, *query* packets, or pid-vid mapping registration/lookup packets, a similar process is used: the key difference here is that the destination *vid* does not correspond to a physical node, but instead 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 $k = \delta(x, dest)$, it does *not* drop the packet. Instead it flips the ($\mathcal{L} - k$)th bit (counting from the left) in the destination vid, and uses this *updated vid* to look up the routing table, trying to find a valid nexthop to reach the node closest to vid. If the level-$(k-1)$ routing entry is also empty, it flips the ($\mathcal{L} - (k-1)$)th bit in the (updated) destination vid, and looks up its level-($k-2$) routing entry. This process stops either when a nexthop, or node x discovers that it is the closest node to this destination vid. (See Algorithm 5)

3.4.5 Handling Node/Link Failures

VIRO handles node/link failures, without resorting to flooding of failure notifications (as used in OSPF). Instead, it utilizes a *withdraw & update* mechanism: Upon discovering the failure, a node adjacent to a failed node (say, a gateway node) or a failed link (to a gateway node) notifies the appropriate rendezvous point(s) by withdrawing its previously published connectivity

information. When a rendezvous point receives this withdraw notification, it notifies all (or a subset of) nodes in an affected sub-tree, namely, those that are currently using the failed or no longer reachable gateway in their routing tables. Various optimizations are possible to limit the number of notification messages. For instance, when a level-k gateway x fails, only the nodes in $S_k(x)$ that lie within the vicinity of this gateway – and thus are on the critical path to this failed gateway – may be notified, with a new gateway installed. These nodes then can re-route packets sending toward the failed gateway, and notify the sender regarding the failed gateway with an updated new gateway. In this manner, all other nodes either discover a new gateway through periodic level-k queries, or via piggybacked notifications. More specifically, the rendezvous point sends a *update* message containing the *withdrawal* of the current gateway, replacing it with a new gateway. Hence only these nodes that are affected by the failures need to update their routing entries. When multiple gateways are used, fast rerouting can be invoked at the node detecting the failure, see Section 3.6 for a brief discussion. Hence under VIRO, failures are *localized*, as only those nodes within the same subtree of the failed node/link are likely to be affected. Nodes outside this subtree are in general not affected, thus no routing table updates are needed at all. When a rendezvous node fails, a neighboring node would then take over and serve as the new rendezvous node. The gateway information would be either (partially) recovered from the current gateway(s) stored in its routing table, recovered (from another rendezvous point, when multiple rendezvous points are used), or in the worst case, learned later through periodic publications by (still available) gateways. In practice, we assume that for large networks, multiple rendezvous points will be used for enhanced robustness.

3.5 Virtual ID Lookup and Forwarding

In VIRO, address/name resolution (i.e., pid-vid mappings) and *vid* look-up mechanisms are implemented on top of the *vid* space using Kademlia-style DHTs. 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 a host is looked up using its pid, packet forwarding between VIRO nodes is done solely based on vid. For more flexible and better support for mobility, *geographically-scoped hash* functions (see [54, 55]) may also be used for *vid* lookup and address/name resolution. Due to space limitation, in the following we describe only the *basic vid* lookup and forwarding operations in VIRO.

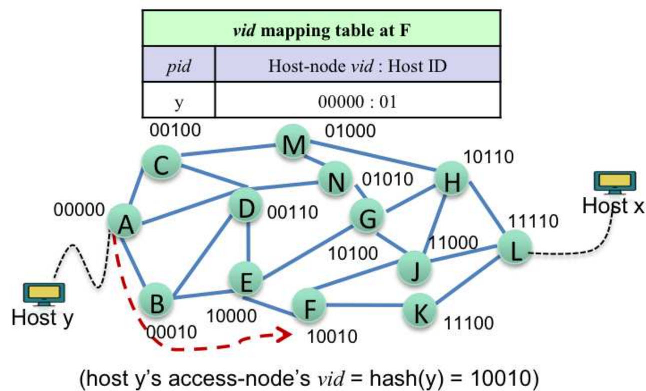


Figure 3.5: vid publish process.

When an end-host h is attached to a VIRO switch (which becomes the host's *host-node*), in addition to assigning a *vid* to the end-host, the host-node also *publishes* one or several pid-vid mappings (*key-value* pairs), e.g., MAC address, IP address, or a flat-id name to *vid* mappings. Let $\langle pid(h), vid(h) \rangle$ denote such a mapping. Using $pid(h)$ as the key, the mapping is stored at the node – referred to as the *access-node*[5] of pid or the host h) – whose vid is closest to $hash_{\mathcal{L}}(pid(h))$ based on the XOR distance, as in [51]. The host-node also stores these pid-vid mappings in its local cache. As long as the end-host is attached to its host-node, the host-node will periodically publish its pid-vid mappings. Fig. 3.5 illustrates the vid publish mechanism: when a host (y) joins the network by connecting to node A, the host-node A for y publishes the mapping at the access node, which is F in the example.

When another node wants to look up the $vid(h)$ for a host h identified by its $pid(h)$ in some namespace (e.g., the MAC address, IPv4 address or flat-id name of host h), it uses $pid(h)$ as the key, or more precisely, the hashed key, *vid* $= hash_{\mathcal{L}}(pid(h))$, and queries the network. The corresponding access-node (whose *vid* is closest to $hash_{\mathcal{L}}(pid(h))$ then responds with the stored $vid(h)$, if host h exists in the network; otherwise, an error message is returned.

We illustrate the packet forwarding mechanism in Fig. 3.6, which consists of two steps: a) *Host vid lookup:* when a host x wants to send a packet to a destination host y, it first performs

[5] In practice multiple access-nodes may be used for robustness, where each is determined by using a different random hash function, $hash'_{\mathcal{L}}$. The periodic publication of pid-vid mappings also ensures that such mappings will not be permanently lost when an access-node fails.

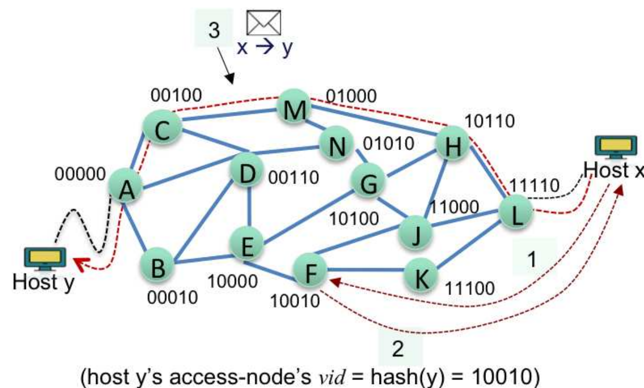


Figure 3.6: Steps in packet forwarding.

the vid resolution for host y, as described earlier. Namely, host x sends a pid-vid mapping request to its host-node (node L), which forwards it to the corresponding access-node for host y, namely, node F whose vid is closest to $hash(pid(y))$ (step 1 in Fig. 3.6); upon receiving this request, node F looks up its mapping table, and returns y's vid (00000:01) to x (step 2). b) Packet forwarding using vid: host x includes y's vid as the destination vid in the packet, sends it to its host-node L. Using y's vid, the packet is then forwarded from node L to node A (y's host-node), using the VIRO routing algorithm. Node A then delivers it to host y (step 3). In VIRO, end-hosts may also be *VIRO-agnostic*, e.g., for backward-compatibility, in that the *first-hop* and *last-hop* packet forwarding (between an end-host and its host-node) is done using the "native" physical network forwarding mechanism (e.g., MAC address and Ethernet), and only host-nodes understand VIRO and perform the necessary *vid* look-up, address/name resolution and pid-*vid* translation (at the first- and last-hop). See [44] for an example of such implementation with VIRO-agnostic end-hosts.

3.6 Additional Features & Discussion

• **Handling Link Weights.** For clarity in presenting VIRO, we have assumed each link has a unit weight, and thus physical distances are defined in terms of *hop counts*. As in OSPF, we can easily accommodate link weights in VIRO by defining physical distances in terms of link weights, and incorporating them in the (distributed or centralized) vid space construction and

assignment algorithms.

•**Fast Failure-Rerouting & Multi-path Routing.** Topologies for data-center networks and large-scale enterprise networks in general have rich path diversity. These topologies are designed to allow multiple paths to connect any pair of nodes for load-balancing or robustness. By utilizing multiple gateways in the routing table at each bucket level, VIRO provides *built-in* support for multi-path routing, load-balancing and fast re-routing, with any additional complex mechanisms. For example, VIRO enables m-way multi-path routing by simply learning and installing $m - 1$ additional gateways (in addition to the default (logically closest) gateway) to reach each bucket. This allows a node to choose one of m different paths to reach a destination bucket, either for load-balancing or for fast-rerouting. By including the gateway node's vid as part of the forwarding directive in the packet header, we can guarantee there is no forwarding loop (see Appendix A for details).

More precisely, in case of fast failure-rerouting and Multi-path routing using VIRO, the forwarding directive in the packet header has one more field as *intermediate gateway* in addition to source and destination of the packet. If a node wants to use a specific gateway to reach the destination bucket, and if intermediate gateway is not used in the forwarding directive, then it can choose any non-default gateway by putting the vid of the gateway in the intermediate gateway field. If a node sees that packet has intermediate gateway set on the packet then it forwards the packet towards the intermediate gateway instead of the final destination, when packet reaches the intermediate gateway it can either reset the intermediate gateway field or choose a new intermediate gateway to reach the destination. Furthermore, *intermediate gateway* field is used to re-route the traffic in the event of node/link failures. To use this, if a link to current nexthop for the destination (or intermediate gateway) has failed then current node can forward the packet towards a gateway for which it has a working nexthop entry.

• **Multiple Virtual Topologies.** Similar to VLANs, VIRO can easily support multiple virtual topologies or virtualized networks on top of the same physical network substrate to enhance security, robustness, performance or network isolation. For example, we can construct multiple *vid* spaces and build routing tables for each *vid* space. Comparing existing layer-2 VLANs or layer-3 virtual topology routing, VIRO can potentially support far larger number of virtual topologies or virtualized networks, since VIRO routing tables are small and their computation incurs relatively little overheads.

3.7 Evaluation

Through extensive simulations, in this section we evaluate VIRO using various real and synthetic network topologies. We also compare VIRO with several existing routing protocols such as OSPF [13], SEATTLE [2], ROFL [19] and VRR [52]. Both OSPF and SEATTLE use link-state routing protocol [48] to construct the forwarding tables for the nodes. ROFL and VRR also use the link-state routing algorithm for constructing the *v-set* paths. Therefore, in the following when comparing the control overheads of VIRO with these protocols, the link-state routing algorithm (OSPF) is used as the representative example.

We have developed our customized in-house simulator for VIRO so as to extensively simulate VIRO on large network topologies using the available computing resources. The following topologies are used in our evaluation, which is also summarized in Table 3.6.

Router Level AS topologies: These are the router level AS topologies collected by Rocket-Fuel [56]. We provide simulations results for following three AS's topologies: i) AS 1755, ii) AS 3967 and iii) AS 6461.

Data Center Topologies: We generated a large number of data-center topologies using the Fat-Tree [57] based designs to evaluate VIRO. Here we provide results for following three topologies: i) DC125, ii) DC320, iii) DC500. Each of these topology was created using 125, 320 and 500 commodity switches respectively, which were arranged to form 3 layers: i) ToR (top of rack switches), ii) Aggregation switches and iii) Core switches. Hence, maximum shortest distance between any two switches(nodes) in these topologies is 4 hops.

Synthetic Router Level AS topologies using Brite: We used Brite [58] network topology generator to generate the router level AS topologies containing different number of nodes. In this section we include the results for topologies generated using 200, 400 and 600 nodes using the Barabasi model in Brite.

3.7.1 vid-assignment evaluation

We evaluate the efficacy of the *vid* assignment in generating a topology-aware *vid* space using both the distributed and centralized *vid* assignment methods. For this evaluation we considered all the topologies mentioned earlier. However, due to space limitations we present results for only one topology in each category i.e. BT200, DC500 and AS3967.

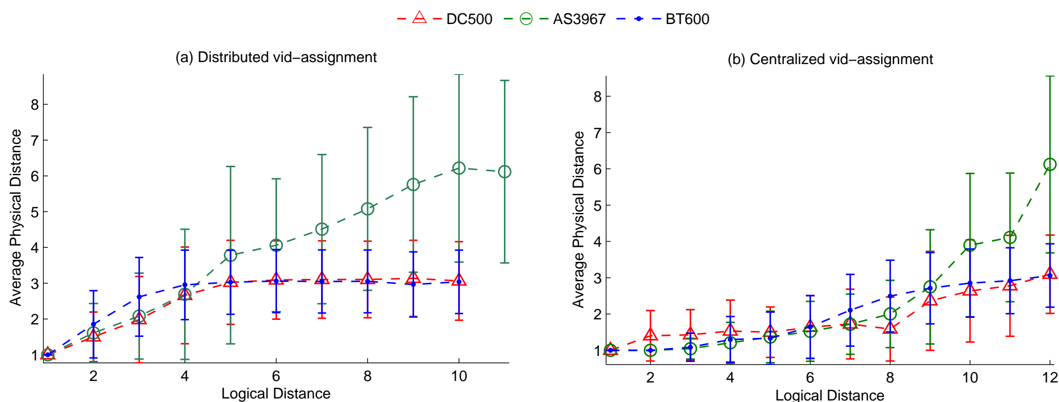
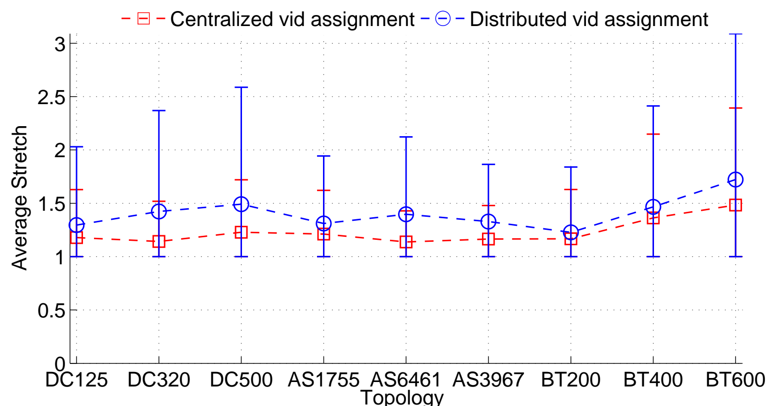


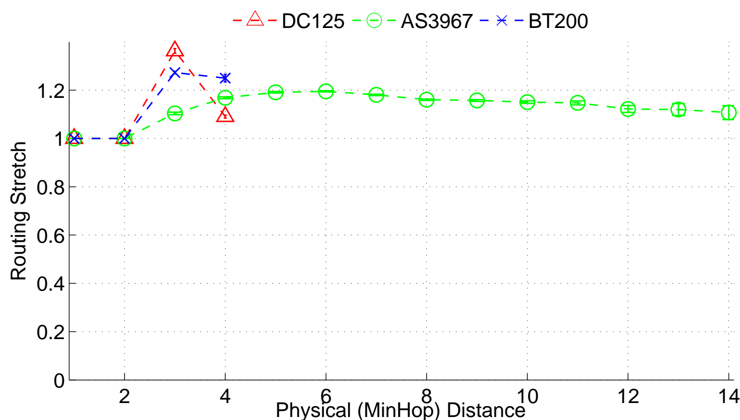
Figure 3.7: Distribution of $physical$(MinHop) distance with the $logical$ distance.

We plot average physical (minhop) distance for each pair of nodes at different logical distances in Fig. 3.7. In this figure, y-axis shows the average physical distance for all the pairs of nodes, at a given logical distance shown on the x-axis. Vertical bars at each data point represent the 95% confidence interval for the mean value. As seen in this figure, the average physical distance for the pairs of nodes increases with the logical distance for both distributed and centralized *vid* assignments. Therefore, the pairs of nodes which are logically closer to each other are likely to be physically close too. These results show that both the centralized and distributed *vid* assignments embed physical topology and hence provide *topology-aware vid*-layer.

Furthermore, as seen in Fig. 3.7 centralized and distributed *vid* assignments show slightly different physical distance distributions with respect to the logical distance. This is because of the bottom-up and top-down approaches used by the distributed and centralized *vid* assignment methods. Bottom-up approach provides more balanced clusters at the bottom by putting physically closer nodes together in the beginning, which is reflected as a proportional growth in the physical distance for the smaller logical distances in Fig. 3.7. On the other hand, top-down approach provides partitions containing almost equal number of nodes at the top by separating the nodes which are physically farthest from each other into different clusters, and hence proportional growth in the physical distance for the larger logical distances.



(a) overall



(b) with physical distance

Figure 3.8: Distribution of routing stretch for VIRO.

3.7.2 Routing Overheads

Routing Stretch: VIRO does not use shortest path routing, therefore it incurs a marginal overhead in terms of the routing optimality. We measure this overhead using *routing stretch*. We define routing stretch as the ratio of the length of path taken using VIRO and physical distance between a source and destination pair. Fig. 3.8 plots the average routing stretch for different topologies. Fig. 3.8(a) shows that average routing stretch remains close to 1 for most of the topologies. Furthermore, centralized *vid* assignment incurs much smaller stretch than

distributed *vid* assignment, it is because an optimal *vid* assignment is achieved using graph-partitioning algorithms. Fig. 3.8(b) plots distribution of the routing stretch with physical distance(minhop distance). In this figure x-axis is the physical distance, and y-axis shows the average routing stretch for the pairs of nodes at a given physical distance from each other. As seen in this figure, the routing stretch is close to one for pairs of nodes at different physical distances. However, it increases slightly in the middle and then decreases as the physical distance increases. This is because the routing stretch is measured as the ratio of number of hops taken by VIRO and the physical distance. Therefore, an extra hop increases this ratio more for smaller physical distances than for larger physical distances. SEATTLE and OSPF use the shortest path routing and therefore do not incur any routing stretch.

Routing Table Size: A key metric for evaluating the scalability of routing protocol is the size of routing table at each node. Nodes in VIRO keep only one routing entry to reach each of its buckets. There are $O(log_2(n))$ number of such buckets in a network of n nodes. On the other hand, nodes in the Link-state routing keep one routing entry to reach each node in the network, therefore each node in the Link-state routing protocol has to keep n entries in its routing table. Fig. 3.9 plots the size of routing tables for VIRO and link-state routing protocols for different topologies. This figure 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 distance, on the other hand link-state routing treats every node equally and has to keep routing entry for each node in the network.

Since OSPF and SEATTLE use link-state routing protocol to create similar routing tables, these protocols also keep similar routing tables, which grows linearly with the number of nodes in the network.

Control Overhead: Another metric that we used to evaluate the scalability of the routing protocol is the control overhead. We estimate the control-overhead for a node by counting the number of control-messages processed by that node to build the complete routing table. In case of VIRO this control-overhead is created by the control-messages corresponding to *Rendezvous publish* and *query* packets.

In this evaluation we also consider four different variants of VIRO 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 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.

In case of Link-state routing protocol each node floods the connectivity information (LSA: Links State Advertisements) through its neighbors, which are used by other nodes in the network to build the topology of the whole network and construct the routing tables. We compare the overhead due to control-messages used by VIRO and Linkstate in Fig. 3.11. In this figure, 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 larger for the link-state than VIRO. It is because of the "flooding" based mechanism used by the link-State routing protocol. In case of VIRO it is much lesser due to the *publish-&-query* mechanism used by it. These results show that VIRO has better scalability than the link-state based protocols in terms of routing- table size and control overhead.

Furthermore, there is a marginal growth in per node control overhead in VIRO when we increase the number of rendezvous nodes in the network. This is because the number of *rendezvous-publish* messages increases linearly with number of rendezvous nodes in the network, and therefore slight increase in the overhead.

Next we compare the memory overhead at rendezvous nodes at different levels. We measure the memory overhead by counting the number of *edges* stored at a given rendezvous node to reach the corresponding bucket. Fig. 3.12 plots average memory overhead for rendezvous nodes with respect to level. It shows that memory overhead on rendezvous nodes increases with level. However this memory overhead need not to be stored on "faster-memory" on the routers. Also, for all the practical topologies this overhead is limited by the number of physical connections between different buckets at any level which can not grow infinitely.

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. Fig. 3.13 shows the distribution of control-overhead on rendezvous nodes at different levels. In these figure x-axis represents the level and the y-axis shows the number of publish/query messages received by the rendezvous node. These figures show that control overhead increases with the level of rendezvous node, however having more number of rendezvous nodes helps in significantly 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* to *vid* mappings at switches by constructing a Kademlia style DHT. In Fig. 3.10 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.

3.7.3 Failure Dynamics

In VIRO, nodes store less routing information for the nodes which are far from it and more information is stored to reach physically closer nodes. Because of this any node in VIRO is less likely to be affected by the failures which occurred at a distance node. This helps in localizing the effect of failure in VIRO. On the other hand link-state protocol are sensitive to failures in the network irrespective of location, where a failure at any location is likely to affect all the nodes in the network.

In this section we compare the effect of failure dynamics for VIRO and SEATTLE by simulating random node failures. A recent study [18] has shown that 50% of the network device failures in a data-center are caused by the failures of less than 4 network equipments and 95% device failures are due to the failure of less than 20 network equipments. This shows that most of the network failures are caused by the failure of a very small number of devices. Therefore, we simulate a large number of failure scenarios by randomly removing small number of nodes from the topology.

Failure Control-Overhead: In Fig. 3.14(a) we compare the control overhead due to the failure notification messages for VIRO and OSPF. In this figure y-axis(plotted using log-scale) shows the average number of control-messages processed by each node for VIRO and OSPF for the corresponding topology shown in x-axis. In case of Link-state routing protocols, whenever a link/node fails, the neighboring nodes are required to flood the link failure information in the network so that other nodes can learn the failure and recompute their routing tables. It causes huge number of failure notifications in the network. On the other hand, in case of VIRO it requires the withdrawal of the gateway information published by the nodes, once this withdrawal is received by the corresponding rendezvous node it notifies the nodes which were using the failed link/node, so that they can update their routing tables. As seen in Fig. 3.14(a), this no

flooding based mechanism used in VIRO helps in reducing the number of failure notification messages drastically.

Next we evaluate the effect of rendezvous node failures for VIRO. Here, we consider the scenario with only one rendezvous node per level. It is because the failure of a rendezvous node in case of multiple rendezvous nodes does not create any overhead during failures, as nodes can easily switch to other replicas of the rendezvous nodes in the event of failures. Fig. 3.14(b) plots the overhead due to the failure of the rendezvous nodes at different levels. In this figure y-axis shows the control overhead on each node in the same sub-tree as failed rendezvous node level, which is shown on the x-axis. This figure shows that control-overhead to spread the failure notifications increases with the level of rendezvous node. However it stays 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.

We measure the effectiveness of VIRO to localized the effect of failures by comparing the control overhead on the nodes with the logical distance from the failed node (see Fig. 3.14(c)). In this figure y-axis shows the control overhead on a node with respect to logical distance from the failed node, which is shown on the x-axis. It shows that nodes which are logically far from the failure are less affected by the failures. On the other hand failures are more likely to affect the nodes which are close to it. Therefore, VIRO is very effective in localizing the affect of failures.

Fast Failure Re-routing: VIRO supports the fast re-routing in case of failures on the current forwarding paths. It is enabled by allowing more than one gateway per destination bucket, therefore when a nexthop to current gateway fails, packets are re-routed through a different gateway to the destination bucket. We consider three variants of VIRO as VIRO-GW1, VIRO-GW2, and VIRO-GW4 which allow nodes to keep maximum of 1, 2, and 4 gateways per destination bucket at each bucket level respectively. Fig. 3.16 shows the number of source destination pairs disconnected during the time interval between the occurrence of the failure and the re-computation of the routing tables for OSPF and VIRO. These figures show that having multiple gateways for each bucket greatly enhances the connectivity during the failures. Furthermore, it shows that having a maximum of 4 gateways per bucket was sufficient to re-route the packets for all the source-destination node pairs.

3.8 Conclusion

In this chapter we presented *VIRO*—a novel routing architecture for large-scale networks. The key idea in our design is to introduce a *topology-aware, structured* virtual id (vid) space onto which both physical identifiers as well as higher layer addresses/names are mapped. VIRO completely eliminates *network-wide flooding* in both the *data* and *control* planes, and thus is highly scalable and robust. 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. Our evaluation of VIRO using many synthetic and real topologies shows the immense scalability and robustness of VIRO, while keeping the overheads very low. In the next chapter we will see how VIRO can be used to enable large-scale layer-2 networks, while maintaining its simplicity and manageability.

Round-1: A learns it's physical neighbors			
Bucket	Prefix	NextHop	Gateway
1	00001	-	-
2	0001*	B	A
3	001**	C,D	A

Round-2: A publishes reachability to $B_2(A)$			
Bucket	Prefix	NextHop	Gateway
1	00001	-	-
2	0001*	B	A
3	001**	C,D	A

Round-3: A publishes reachability to $B_3(A)$			
Bucket	Prefix	NextHop	Gateway
1	00001	-	-
2	0001*	B	A
3	001**	C,D	A

Round-4: A queries reachability to $B_4(A)$ Rendezvous point returns C as gateway to reach $B_4(A)$			
Bucket	Prefix	NextHop	Gateway
1	00001	-	-
2	0001*	B	A
3	001**	C,D	A
4	01***	C	C

Round-5: A queries reachability to $B_5(A)$ Rendezvous point returns B as gateway to reach $B_5(A)$			
Bucket	Prefix	NextHop	Gateway
1	00001	-	-
2	0001*	B	A
3	001**	C,D	A
4	01***	C	C
5	1****	B	B

Table 3.3: Routing table for node A shown in Fig. 3.2

Algorithm 4 Modified *query* and *forwarding* functions using bridges

1: **Query** ($rdv_k(i)$, k,i))
2: CP: Control Packet, $dist$: k
3: $CP.src := v_i$
4: $CP.dest := rdv_k(i)$
5: $CP.payload := dist$
6: $CP.type := RDV_QUERY$
7: **forward**(CP)
8: $List(gw, prefix)$:= QueryReply()
9: return $List(gw, prefix)$

1: **Packet** (msg) **Forwarding at node** i
2: $nexthop = Nil$
3: $k := \delta(i, msg.dest)$
4: **if** $R_k(i)$ not Nil **then**
5: $nexthop = R_k(i).bestmatch(msg.dest).nexthop$
6: **end if**
7: **if** $msg.type = RDV_QUERY$ or $msg.type = RDV_PUBLISH$ **then**
8: **while** $nexthop = Nil$ or $k = 0$ **do**
9: $msg.dest = Flip_Kth_Bit(msg.dest)$
10: $k := \delta(i, msg.dest)$
11: $nexthop = R_k(i).bestmatch(msg.dest).nexthop$
12: **end while**
13: **end if**
14: **if** $msg.dest = i$ **then**
15: ProcessPacket(msg)
16: **else**
17: sendPacket($nexthop, msg$)
18: **end if**

1: **Bridge Query Handling at Rendezvous Node** i
2: src : Node which made the Bridge Query.
3: k' : Level for the bucket src is trying to reach
4: $k = \delta(src, i)$
5: Divide edges to $B_k(i)$ into prefix specific sets.
6: Learn that $S_{k'-1}(src)$ is not connected to $B_{k'}(src)$
7: Look for an edge of type $y \leftrightarrow z$ where $y \in prefix, z \in S_{k-1}(i)$ for each prefix in $B_{k'}(src)$
8: **if** $\exists z$ **then**
9: Send z as bridge reply to src
10: **else**
11: Send Nil as bridge reply to src
12: **end if**

Routing table for node A			
Bucket	Prefix	NextHop	Gateway
1	00001	-	-
2	0001*	B	A
3	001**	-	-
4	01***	-	-
5	1****	B	B

Routing table for node M			
Bucket	Prefix	NextHop	Gateway
1	01001	-	-
2	0101*	N	M
3	011**	-	-
4	00***	-	-
5	1****	H	M

Routing table for node J			
Bucket	Prefix	NextHop	Gateway
1	11001	-	-
2	1101*	-	-
3	111**	L	J
4	10***	F, G, H	J
5	0****	H	H

Table 3.4: Routing tables (before bridge) for the nodes in Fig. 3.4

Algorithm 5 Packet (msg) forwarding at node i

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

Routing table for node A			
Bucket	Prefix	NextHop	Gateway
1	00001	-	-
2	0001*	B	A
3	001**	-	-
4	01***	B	H(bridged-gateway)
5	1****	B	B

Routing table for node M			
Bucket	Prefix	NextHop	Gateway
1	01001	-	-
2	0101*	N	M
3	011**	-	-
4	00***	N	E(bridged-gateway)
5	1****	H	M

Routing table for node J			
Bucket	Prefix	NextHop	Gateway
1	11001	-	-
2	1101*	-	-
3	111**	L	J
4	10***	F, G, H	J
5	00***	F	E
5	01***	H	H

Table 3.5: Routing tables (after bridges) for the nodes in Fig. 3.4

Router Level AS Topologies
AS1755 (295 nodes, 543 edges)
AS3967 (353 nodes, 820 edges)
AS6461 (654 nodes, 1332 edges)
Data Center Topologies
DC125 (125 nodes, 500 edges)
DC320 (320 nodes, 2048 edges)
DC500 (500 nodes, 4000 edges)
Synthetic topologies using BRITE
BT200 (200 nodes, 790 edges)
BT400 (400 nodes, 1590 edges)
BT600 (600 nodes, 2390 edges)

Table 3.6: Summary of the topologies used in evaluation.

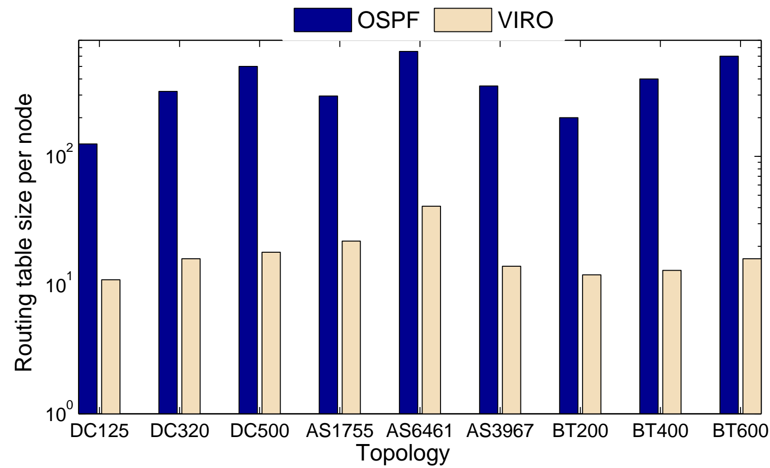


Figure 3.9: Routing Table size comparison for VIRO and link-state routing.

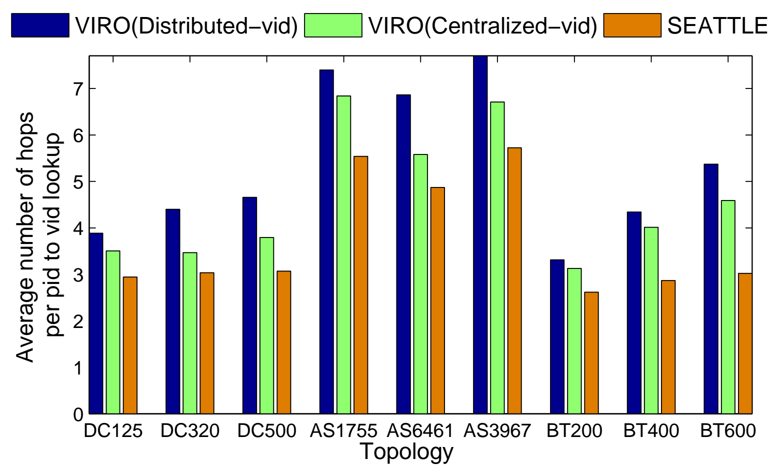


Figure 3.10: Look-up costs for VIRO and SEATTLE.

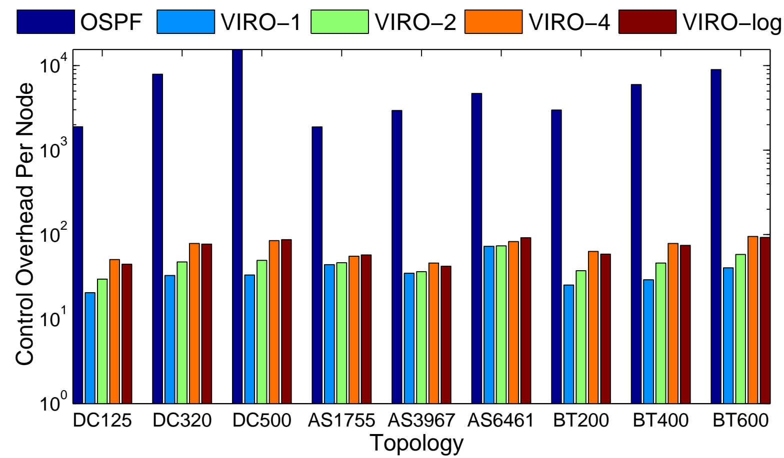


Figure 3.11: Control-overhead for VIRO and the link-state based routing protocols.

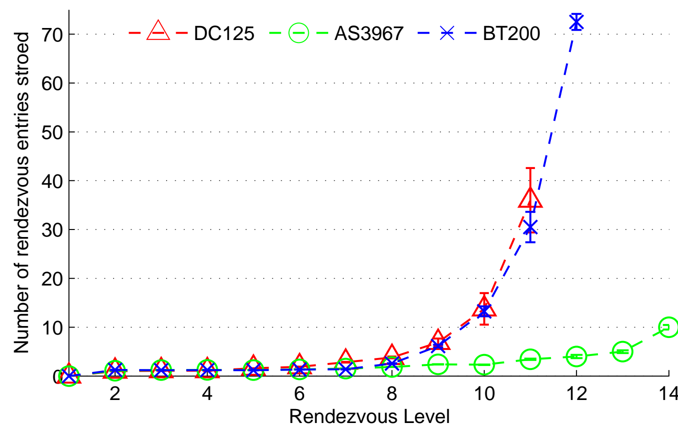
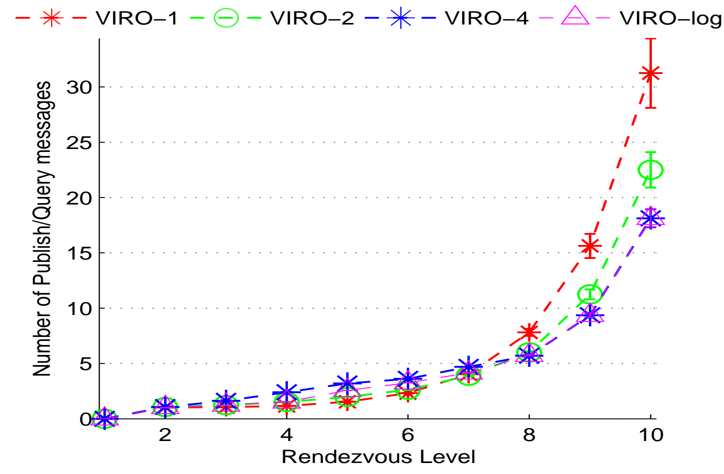
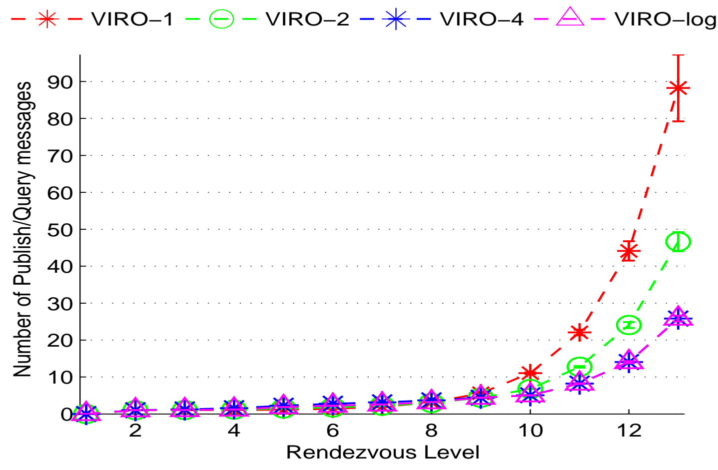


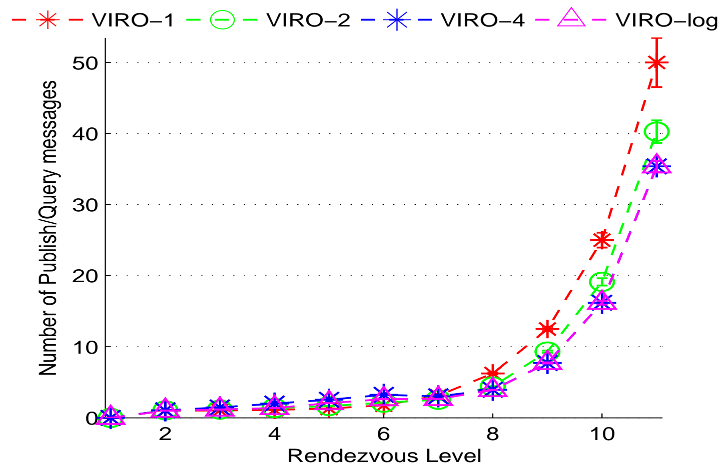
Figure 3.12: Comparison of memory-overhead for rendezvous nodes at different levels



(a) DC125

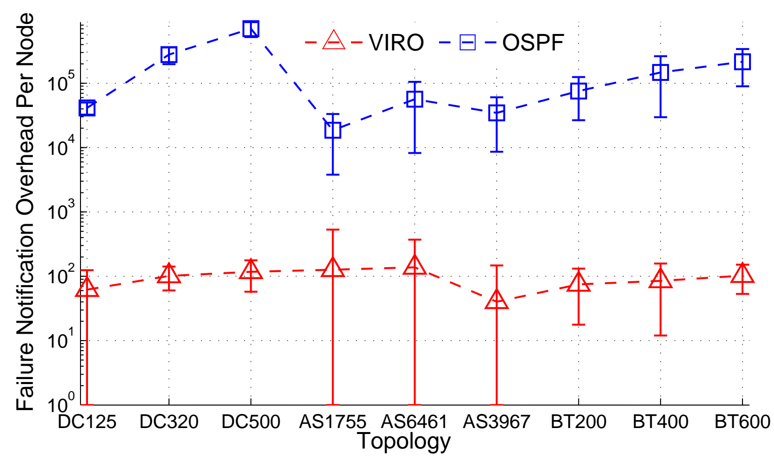


(b) AS3967

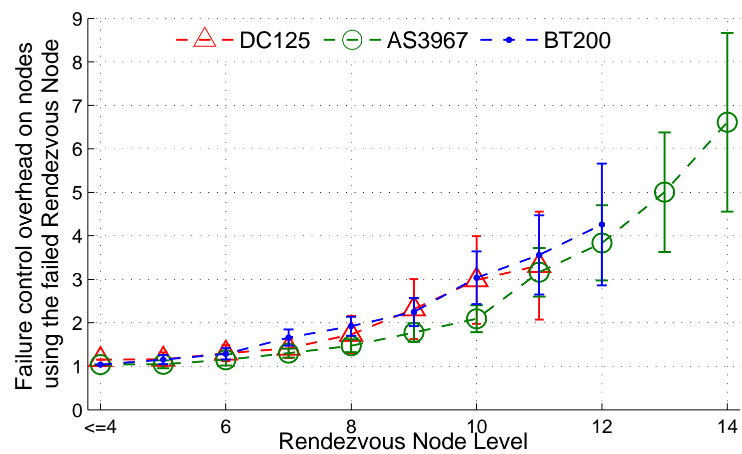


(c) BT200

Figure 3.13: Control overhead on rendezvous nodes.



(a) Control overhead due to the failure notification messages



(b) Failure of Rendezvous nodes

Figure 3.14: Comparison of VIRO and Link-state for failures.

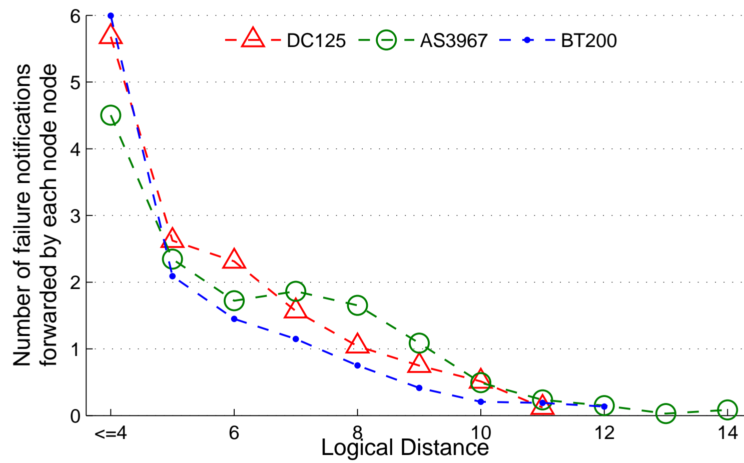


Figure 3.15: Localized affect of failures for VIRO

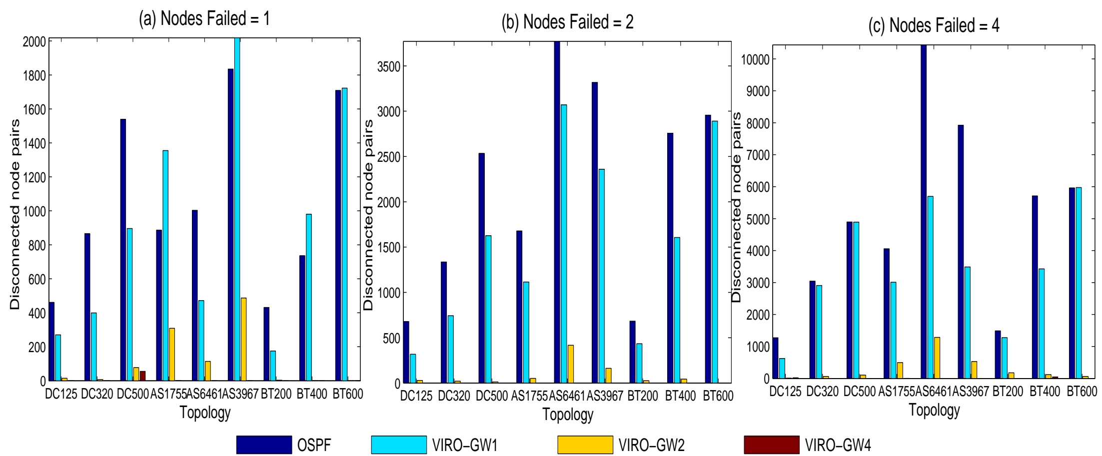


Figure 3.16: Comparison of disconnected node-pairs for VIRO and OSPF

Chapter 4

VEIL: Virtual Ethernet ID Layer

This chapter describes VEILa novel, “plug-&-play” Virtual Ethernet Identier Id Layer for below IP networking. It is a realization of VIRO routing framework to enable large-scale layer-2 networks. The objective is two-fold: i) VEIL directly addresses the scalability, efciency and reliability challenges facing the traditional Ethernet, while retaining its “plug-&-play” feature; ii) but perhaps more importantly, VEIL provides a uniform (below IP) convergence layer to support a large, dynamic and heterogeneous (layer-2) network that is capable of connecting hundreds of thousands or more diverse physical devices. The key idea in our design is to use the “Ethernet addresses” as the place-holder for VIRO based *vids*, which are exactly 48-bit long to ensure the backward compatibility. VEIL completely eliminates network-wide ooding in both the data and control planes, and thus is highly scalable and robust.

4.1 Introduction

The explosive growth of the Internet has enabled a wide range of diverse devices to be inter-connected and communicate with each other through a variety of disparate technologies. While serving as the universal “glue” that pieces together various heterogeneous physical networks, the Internet Protocol (IP) suffers certain well-known shortcomings, e.g., in terms of need for careful and extensive network configurations, relatively poor support for mobility, and so forth. In contrast, layer-2 technologies such as Ethernet are largely “plug-&-play” in that hosts are equipped with persistent MAC addresses, and Ethernet switches automatically learn about host addresses and location, adapt to changes in network topology as well as host mobility, perform

packet forwarding seamlessly with minimal operator configuration and intervention. Because of this simple "plug-&-play" semantics, today's *switched* Ethernet technology (where the collision domain is no longer a size-limiting factor) has been rapidly expanded to *large, dynamic* networks, such as large data centers and Metro Ethernet, with up to tens of thousands switches and millions of hosts.

On the other hand, the unprecedented scale as well as the demanding efficiency and robustness requirements of these new large, dynamic (layer-2) networks also pose revolutionary challenges on the Ethernet technology that was originally developed for small, local area networks. For instance, the *network-wide flooding*–often resorted by Ethernet switches to locate end hosts and forward packets whose locations are yet to be learned–not only significantly reduce the network capacity. The spanning tree algorithm used to avoid forwarding loops not only results in sub-optimal forwarding paths, but also is slow to adapt to changes in the network topology. To address these challenges, several solutions [7, 6, 59, 60, 2] have been proposed, of which SEATTLE [2] is closely in spirit to our work, in that both utilize DHT (distributed hash table) techniques for scalable and efficient address look-up and resolution. However, SEATTLE employs the OSPF-style shortest routing in layer 2. It therefore not only requires network-wide flooding in the control plane for building routing tables, but also suffers the same scalability and robustness limitations plaguing shortest-path routing. (We refer the reader to Sec.4.2 for further discussion of these and other related works.)

In this chapter we introduce *VEIL*—a novel, "plug-&-play" *Virtual (Ethernet) Identifier (Id) Layer* for below IP networking. The objective is two-fold: i) like SEATTLE, VEIL directly addresses the challenges facing the traditional Ethernet, while retaining its "plug-&-play" feature; ii) but perhaps more importantly, VEIL provides a uniform *convergence* layer (or "logical link layer" using the ISO OSI parlance) to support a large, dynamic *and heterogeneous* (layer-2) network that is capable of connecting hundreds of thousands or more *diverse* physical devices–not only Ethernet-equipped devices, but also non-Ethernet devices such as 802.16-based sensors, blue-tooth devices–in a *scalable* and *robust* fashion. The proposed VEIL architecture is a shim layer that operates under the (traditional) network layer (e.g., IPv4/IPv6) and above the ("native") link layer/physical layer such as Ethernet, 802.11 Wireless LANs, etc. The key idea in our design is to introduce a *topology-aware, structured* virtual id (vid) space onto which both physical identifiers (e.g., Ethernet MAC addresses), pid's in short, as well as higher layer addresses/names (e.g., IPv4/IPv6 addresses) are mapped. Using such a topology-aware, structured

vid space as the basis for efficient and scalable (DHT-style) object look-up/address resolution, routing and forwarding, VEIL completely eliminates *network-wide flooding* in both the *data* and *control* planes. As a result, VEIL is highly scalable and robust while at the same time offers better support for multi-homing and mobility.

The organization of this chapter is as follows. In Sec. 4.2 we provide an overview of the proposed VEIL architecture, and in Sec. 4.3 we describe the key mechanisms of VEIL. We conclude the chapter Sec. 4.4.

4.2 Overview and Related Work

4.2.1 Overview of VEIL

The proposed *VEIL* architecture is a shim layer that operates under the (traditional) network layer (e.g., IPv4/IPv6) and above the ("native") link layer/physical layer such as Ethernet, 802.11 Wireless LANs, etc. We assume that each physical device has a 48-bit Ethernet MAC address. The proposed VEIL architecture allows *heterogeneous* physical end devices to be plugged into the same layer-2 network. In lieu of Ethernet switches or wireless access points (APs), we have *VEIL switches* to which end devices (either through wired or wireless channels) are connected: they "speak" the *native* MAC/physical protocols to deliver data to/from these connected physical end devices; among themselves, they communicate using the VEIL protocol and perform VEIL operations such as vid assignment, mapping, routing to provide scalable and robust end-to-end connectivity and data delivery within a VEIL network.

The key idea in our design is to introduce an Ethernet compatible *topology-aware, structured* virtual id (vid) space onto which both physical identifiers (or Ethernet MAC addresses), pid's in short, as well as higher layer addresses/names (i.e., IPv4/IPv6 addresses) are mapped (see Fig. 3.1(a)). VEIL switches are assigned vid's in a manner such that if they are *logically* close in the vid space are also *physically* close to each other. Using a (binary) Kademlia virtual tree as an example, Fig. 3.1(b) shows such an embedding: the leaf nodes correspond to VEIL switches (*not* physical devices connecting to them!), the vid of a VEIL switch is the binary strings along the path from the root to the corresponding leaf node. The logical distance between a pair of vids in this vid space is defined as the difference of number of bits used to represent a vid and the length of the longest common prefix for the pair. In Sec. 4.3 we will briefly discuss how the vid assignment can be performed in either a centralized or distributed

fashion. End devices connecting to a VEIL switch inherit an *extended* vid consisting of the (32-bit) vid of the switch plus a (randomly assigned) 16-bit local vid (see Sec. 4.3 for detail).

Taking advantage of this topology-aware, structured vid space, VEIL switches run VIRO routing protocol to collaboratively build routing tables, maintain network-wide connectivity and perform end-to-end data delivery across a VEIL network. As explained earlier, in VIRO, routing tables are constructed piece-meal based on the vid logical distance instead of physical distance (e.g., hop counts), and packets from a source VEIL switch are forwarded towards their destination along a logical path with decreasing logical distance (to the vid of the destination VEIL switch). In Sec. 4.3 we briefly outline the basic operations of VIRO, and describe how packets are forwarded by VEIL switches using VIRO.

While at the expense of incurring additional routing stretches–albeit fairly small in general thanks to the topology-aware construction of the vid space (see Sec. 3.7)–when compared to the shortest path (using the physical distance) routing, the logical distance-based VIRO routing affords several important advantages. i) *Scalability.* By taking the *structured* vid space and constructing routing tables in a piecemeal, bottom-up fashion, VIRO completely eliminates *network-wide flooding* in both the data plane (unlike Ethernet switching algorithm) and *control plane* (unlike OSPF and other shortest path routing algorithms). Furthermore, because of the natural *hierarchical* structure of the vid space, routing information regarding far-away part of the network is automatically aggregated using the vid prefixes. As a result, the routing table size is in the order of $O(\log N)$, where N is the number of VEIL switches in the network, as opposed to $O(N)$ (as in the case of OSPF). ii) *Robustness.* Unlike OSPF, no *network-wide* full topology needs to be maintained by any switch, thanks to the structured vid space, and hence changes in network topology do not need to be flooded globally. Due to the aggregate routing information maintained by switches, failure of a link or switch node can be *localized*, without affecting nodes in far-away parts of the network. Furthermore, path and topology diversity can be easily exploited in VIRO by using multiple forwarders; hence failure of one forwarder does not affect network-wide reachability. iii) *Multi-Homing and Mobility.* VIRO also provides seamless and better support for multi-homing and mobility. It allows an end device to be connected with multiple VEIL switches without causing loops and other complexities; mobility of an end device can be easily supported through scalable and efficient pid (or higher layer names/addresses) to vid mapping and lookup.

4.2.2 Related Work

There are several proposals, e.g., RBriges [7], CMU-Ethernet [6], Viking [59], SmartBrigdes [60], that attempt to address the scalability limitations in scaling Ethernet to *large, dynamic* networks (see Sec. 1.1 of [2] for more detailed discussion on these proposals). Our work is closely related to SEATTLE [2], with similar goals but two *key differences*. As pointed out earlier, while SEATTLE eliminates data plane flooding, it employs OSPF-like shortest path routing, which requires *network-wide flooding* of link state advertisements (LSAs) in maintaining network topology and tracking its changes. SEATTLE thus suffers the same limitations plaguing OSPF-based IP routing. For example, Node and link failures therefore require network-wide flooding of LSAs and re-computation of routing tables at all nodes. For scalability, hierarchical, area-based routing must be introduced, which in turn introduces additional management complexity as well as routing stretch penalty. In contrast, with the introduction of vid's and a structured vid space, VEIL can support a large, dynamic (below IP) network with heterogeneous devices, not simply Ethernet-enabled devices, in a more self-organizing and scalable fashion (e.g., with $O(\log N)$ routing table sizes instead of $O(N)$).

4.3 The VEIL Layer

In this section we outline the key mechanisms of VEIL: vid assignment, the VIRO routing protocol, vid lookup/address resolution, and end-to-end packet forwarding. We will use the notation $v(x)$ or v_x to denote the virtual id of a host x (or VEIL switch), and $hash_k(val)$ represents a k-bit hash value for val using a consistent hash function. We will refer to *end-host devices* attached to switches as *hosts*, and the terms *node* and *switch* are used interchangeably to denote a VEIL switch.

4.3.1 VEIL Id Assignment

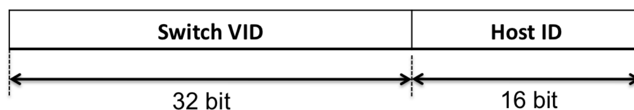


Figure 4.1: vid field structure for VEIL.

To be backward compatible with Ethernet MAC addresses, VEIL assigns a 48-bit long vids to each switch/host in a VEIL network. As shown in Fig. 4.1, a vid comprises of two parts: The first part, the $switch\text{-}vid$ field, is a 32-bit identifier that uniquely identifies a VEIL switch in the network. The second part, the $host\text{-}id$ field, is a 16-bit identifier that uniquely identifies a host attached (through either a wired or wireless link) to a VEIL switch. We refer to the switch that a host is currently attached to as the *host-switch* of the host. The vid of each host is assigned as follows: the first field is set to be the same $switch\text{-}vid$ as its host-switch, and the second field is randomly assigned, e.g., by using a 16-bit hash of its MAC address (pid), $host\text{-}vid := hash_{16}(pid)$, and is locally unique with respect to the host switch. *host-switches* periodically publish the *vid* mappings for its host to the corresponding *access-switches*, which is used to perform the *vid* lookup. For a switch, its $host\text{-}vid$ part of its vid is assigned in a similar fashion. In the following we will discuss how the $switch\text{-}vid$ part of a switch vid is assigned. For conciseness, unless otherwise stated, the term vid simply refers to the $switch\text{-}vid$ part of switch vid's, as they are the ones that form a *topology-aware*, *structured* vid space and are used for routing and forwarding.

In order to perform the *vid*-assignment for the *veil-switches* VEIL can use either of the distributed or centralized mechanisms described in Chapter 3. Using these mechanisms each *veil-switch* is assigned a unique 32-bit long *vid*. On the other hand, the end host device that connect to the network the receive their *vid*-assignment from the *veil-switch* they directly connect to. However, the *vid*-assignment to *host-devices* is completely transparent to them, i.e. *host-devices* are not aware if they are assigned a *vid* at all; as will be clear in Sec. 4.3.3, the vid of an end host is used only by its *host-switch* to look up and translate between its vid and pid (MAC address), thus not used by the end host itself at all.

4.3.2 Virtual Id Routing (VIRO) Protocol used by VEIL-Switches

As described in Chapter 3, *veil-switches* use a bottom up round-by-round protocol to construct the routing tables, where after the kth round, each node[1] x has built a routing entry to reach nodes in its bucket B^x_k. During round 0, each x discovers its directly connected neighbors, and all nodes in B^x_0. During round k, $1 \leq k < L$, x uses a publish/query based mechanism to i) either publish itself as a gateway to B^x_{k+1} if it has a direct connection to a node in B^x_{k+1}, or ii) query to discover such a gateway so as to install a routing entry for bucket B^x_{k+1}. Because of

[1] In this chapter we use the terms node and *veil-switch* interchangeably to represent a routing-node.

the closeness and connectivity properties of the vid assignment, this process is guaranteed to converge and generate the correct routing table.

Once the routing table is constructed, the forwarding process used by *veil-switches* is simple. In order to forward a packet from switch s with vid v_s to a destination d with vid v_d, s first computes the logical distance $k = \delta(v_s, v_d)$. If $k > 0$, s forwards the packet to a nexthop corresponding to Bucket B_k^s in the routing table. If $k = 0$ and v_d is a host directly connected to s, then s directly delivers it to d. In case d is not connected to s anymore, then s encapsulates the packet with destination as the *vid* of the d's *access-switch*, and routes it toward it. When *access-switch* receives this packet, and does not have the updated mapping for the host d it drops the packet. Otherwise, it decapsulates the original packet and updates the *vid* of the host and again forwards the packet to this new *vid*.

4.3.3 Id Lookup/Address Resolution, and End-to-End Packet Delivery

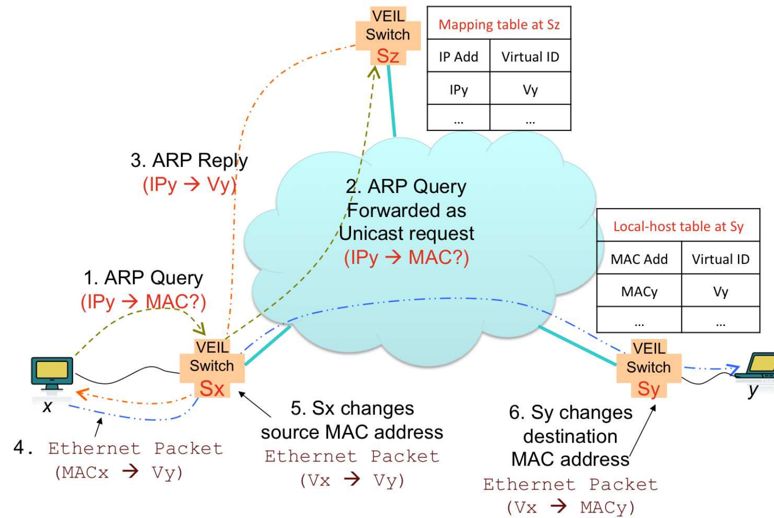


Figure 4.2: vid lookup and address resolution process.

Like SEATTLE [2], we employ a one-hop DHT for id lookup and address resolution. Recall that each host-switch assigns vids to all the hosts connected to it. A host switch discovers the MAC/IP addresses (denoted by pid and IP respectively) of an end host that is directly connected to it by listening to (or "snooping") all traffic on the link. It assigns a vid to each host, and store the $\langle pid, vid\rangle$ mapping as well as the $\langle IP, vid\rangle$ into a local cache. Furthermore,

the host switch periodically publishes the mapping $\langle IP, vid\rangle$ (as long as the end host is still connected to it) to a specific switch, called the *access switch* (of the said IP address), the $switch\text{-}vid$ of which is closest to the 32-bit hash, $hash_{32}(IP)$. The access switch stores this $\langle IP, vid\rangle$ mapping into its cache, and is responsible to answering queries for the said IP address from other nodes.

Fig. 4.2 depicts the address resolution, id lookup and packet delivery processes. When host x wants to send a packet to host y with IP address IP_y, it first sends an ARP query to resolve IP_y to MAC address of y (as in the standard IP operation). This ARP packet is intercepted by the host-switch S_x. If host y is not directly connected to S_x (otherwise, everything operates as the usual Ethernet), the host switch S_x sends a unicast request to the access switch S_z whose $switch\text{-}vid$ is closest to $hash_{32}(IP_y)$. Upon receiving the request, S_z replies with the corresponding $\langle IP, v_y\rangle$ mapping. S_x caches this mapping in its local cache for future packet forwarding, and sends an ARP reply to x with v_y as the MAC address. Host x encapsulates the IP packet in an Ethernet packet with v_y as the destination MAC address. When switch S_x receives this packet, it replaces the source MAC address (host x) by its vid, and forwards the packet toward the host switch, S_y, of the destination host y, using the VIRO packet forwarding described earlier. Upon receiving the packet, the host switch S_y uses v_y to look up the actual MAC address of host y in its local cache, rewrite the destination MAC address field, and forward the packet to host y.

4.3.4 Support for Mobility and Legacy Protocols

Host mobility with a VEIL network can be easily supported by updating the IP to vid (or MAC/ pid to vid) mappings. The host-switches are responsible for updating these mappings and publishing them to the corresponding access-switches, whenever changes take place. VEIL is also completely compatible with the current Ethernet semantics such as ARP/RARP, DHCP and VLAN. As explained in Sec. 4.3.3, the ARP protocol is supported with a flooding-free lookup/query mechanism. Similarly, RARP and DHCP can also be supported: a host-switch simply intercepts the appropriate packets, and then forwards them via unicast to appropriate destinations. Moreover, VLAN can also be accommodated by, say, setting aside the first 8-bit of the $switch\text{-}vid$ as the VLAN number. For each VLAN, there will be one corresponding (virtual) VEIL network, each with its own logical topology and vid sub-space. In this way, VEIL can inherit all the advantages of VLANs, while at the same time improving scalability,

security, and network management.

4.4 Conclusion

In this chapter we have presented *VEIL*—a novel, "plug-&-play" *Virtual (Ethernet) Identifier (Id) Layer* for below IP networking. The key idea in our design is to introduce a *topology-aware, structured* virtual id (vid) space such that it is compatible with the existing layer-2 identifiers used by the *host-devices*. These *vids* then replace the actual MAC addresses on the Ethernet packets when forwarding is done between the *veil-switches*, while the destination *vid* is replaced by the actual MAC address of the destination host at the edges. Similarly, the *host-switch* for the source host replaces the actual MAC address on the destination Ethernet address field in the Ethernet header by the *vid* of the host before forwarding the packets to other *veil-switches*. VEIL completely eliminates *network-wide flooding* in both the *data* and *control* planes, and thus is highly scalable and robust. In the next chapter, we will describe a real prototype implementation of VEIL using the Click modular router framework.

Chapter 5

VEIL-Click: Prototype VEIL-Switch

As explained in the Chapter 4, we can use the VIRO routing framework to design a large-scale Ethernet(layer-2) network. In this chapter, we present *veil-click*, which is a real prototype design of VEIL architecture. It is aimed at simplifying the management of large-scale enterprise networks by requiring minimal manual configuration overheads. It makes it tremendously easy to plug-in a new routing-node or a host-device in the network without requiring any manual configuration. It employs VIRO as a "core" routing protocol, while supports many advanced features such as seamless mobility support, built-in multi-path routing and fast-failure re-routing in case of link/node failures. The current prototype of *veil-click* is built using Click Modular Router framework, and is deployed in our lab for the evaluation.

5.1 Introduction

In this chapter, we describe *veil-click*— a realization of VIRO routing framework, which is tailored towards creating large-scale advanced layer-2 networks. There are several mechanisms that we employ to achieve this. First, we introduce a centralized controller for the network, which collects the network topology and performs the *vid* assignment for the routing nodes in the network. Second, we choose 48-bit long *vid* for both *host-devices* and as well as the routing nodes, so that, we can use the existing Ethernet semantics, such as ARP protocol for name resolutions and so on. Third, we do not introduce any new additional networking header for the basic packet forwarding. This is achieved by re-using Ethernet address fields in the layer-2 headers.

In order to demonstrate the practical feasibility of our routing architecture, we implement the key components of our proposed networking architecture using an initial prototype based on Click Modular Router framework [42], which is not only *completely backward compatible* to work along with the legacy hardware (e.g., Ethernet based switches) and software (e.g., host networking protocols such as ARP, IP based addressing etc.), but also leverages the VIRO routing framework to provide built-in mechanisms for load-balancing, fast rerouting, seamless mobility support and other key features needed to support future networking and application needs. Our initial evaluation using the Click based prototype switch shows that *veil-click* can support seamless mobility support for the *host-devices*, and it does not interrupt the on-going TCP connections for the hosts during the mobility. In addition, our experiments showed that TCP connections for the *host-devices* take only around 2-5 seconds to recover during the *host-device* mobility.

The remainder of the chapter is organized as follows. We provide an overview of VIRO, the key ideas behind *veil-click* and related work in Sec. 5.2. Sec. 5.3 provides the description of basic components, Sec. 5.5 presents the basic design of our Click based prototype and in Sec. 5.4 we describe the advanced features. Finally we conclude the chapter in Sec. 5.6.

5.2 Overview

In this section we discuss the motivation behind the design of *veil-click*, provide a brief description of the related work and an overview of the key ideas behind *veil-click*.

5.2.1 Motivation

The design of *veil-click* is motivated by the several challenges faced by existing enterprise networks. In the following, we highlight some of these challenges.

• **Requirement of extensive network configuration & address management.** Current IP networks require careful (and often manual) configurations and management. The need for *address management* is particularly cumbersome and problematic: while an end host joining a network can dynamically obtain its IP address via DHCP, adding a router or subnet to expand an existing network often requires allocation of one or more new IP address blocks. This is because IP address management is *link-based*: each link – either via a point-to-point connection or wired/wireless broadcast media – (or each subnet) must be assigned a distinct IP address

block. Such address blocks must then be configured into routers, and injected into intra-domain routing protocols. Similarly, the assignment and manual configuration of OSPF areas for IP based intra-domain routing adds to the complexity.

• **Limited ability to exploit the richness in the network topologies.** IP routing within a single network domain (i.e., intra-domain routing) also suffers several major problems. These problems have their root in the *shortest-path* based routing paradigm used in IP (intra-domain) routing. The use of shortest-paths limits the ability of IP networks to exploit path diversity inherent in the network topology to perform load-balancing and fast-rerouting of traffic under failures. To perform load-balancing and traffic engineering, one has to resort to sub-optimal work-arounds or fixes, e.g., via IGP weight optimization. Likewise, various IP-based fast rerouting mechanisms have been proposed, which partly circumvent the slow convergence problem plaguing the traditional *reactive* IP routing protocols (e.g., OSPF or IS-IS). Unfortunately most of these fast rerouting mechanisms are fairly complex, and as "add-ons" to existing routing protocols require additional configurations, which further complicates the operations of IP networks.

• **Poor scalability of existing Ethernet based layer-2 networking protocols.** Unlike IP networks, layer-2 networks such as Ethernet are largely *plug-&-play*: hosts are equipped with persistent MAC addresses, and Ethernet switches automatically learn about host addresses and location, and perform packet forwarding seamlessly with minimal operator configuration and intervention. On the other hand, as it was originally developed for small, local area networks, it relies on *network-wide flooding* for packet forwarding and address resolution, which severely limits its scalability and efficiency.

• **No support for the host mobility.** Current networking protocols consider the IP address of a host as a proxy for the node's identity. While IP address is also used to route the packet to the destination, therefore, it acts as an address for the node as well. When a node moves from one subnet to another subnet in the network, its IP address has to be reconfigured either using the DHCP or by performing static manual configuration. This dual use of IP address as an identity and as well as an address, causes the reseting of the existing network connection between the hosts whenever their IP address is changed due to the mobility.

5.2.2 Related Work

We organize the related work into three categories: routing scalability, multi-path routing & fast failure re-routing and support for host mobility.

a. Routing Scalability. To address the challenges faced by traditional layer-2 routing protocols, several solutions [60, 1, 2] have been proposed, of which, SEATTLE [2] is closest in spirit to our work, in that, both utilize DHT (distributed hash table) techniques for scalable and efficient address look-up and resolution. However, SEATTLE employs the OSPF-style shortest path routing in layer 2. It therefore not only requires network-wide flooding in the control plane for building routing tables, but also suffers from the same scalability and robustness limitations plaguing shortest-path routing: for example, it is limited to the use of shortest paths only; load-balancing and fast rerouting can be complicated to implement. In contrast, VIRO routing framework used in VEIL avoids these inherent problems in shortest-path routing. It is far more scalable and robust (e.g., with $O(\log N)$ routing table sizes instead of $O(N)$ in OSPF). Similarly, to circumvent these problems in a data-center environment, several "customer-made" networking solutions have been proposed, see, e.g., [18,57,61]. However, these solution are tied to a specific network topology, and therefore, can not work for any general network topology, which is mostly the case for large-scale enterprise networks.

Our work is also substantially different from the "flat-id" based routing schemes such as UIP [20] and ROFL [19], which advocate a *flat* universal id space to replace the current global IP address space. These schemes employ a DHT-style randomly and consistently hashed id assignment–which produces an id-space completely independent of the underlying network topology–and perform routing based on logical distance to the id of the destination, incurring a stretch penalty (which is unbounded in the worst case). We circumvent these problems by introducing a *topology-aware* structured vid space. It incurs fairly small routing stretches, and effectively localizes the effect of failures.

b. Multi-path Routing & Rapid Re-routing during Failures. In order to exploit the rich network topologies, and to meet the increasing demand for better availability and reliability several solutions are proposed, e.g., [62, 63, 64, 65]. Most of these works rely on computing multiple routing tables using the OSPF style routing protocol, and use them as the back up routes. For instance, Path Splicing [64] is one such recent work, which attempts to build multiple virtual topologies by randomly perturbing the link weights, and computes the shortest path on each of

these virtual topologies to perform the multi-path forwarding and fast failure re-routing. However, it suffers from the same scalability challenges faced by the shortest path routing, and therefore, may not be a suitable solution for large scale enterprise networks. On the other hand, VEIL leverages the VIRO routing framework to employ simple and scalable multi-path routing schemes, and offers a much more scalable solution. Similarly, compare to several other mechanisms common in large ISP networks rely on 'end-to-end' tunnels between the routers to deploy multiple backup paths, the mechanism used in VEIL offers a much simpler solution and does not require any manual overhead for setting up the backup paths. Similarly, it is different from ECMP [66], which enables multi-path forwarding by using multiple forwarding paths. However, it limits the forwarding to equal cost paths only to avoid the forwarding loops. Whereas, the mechanism deployed in VEIL does not have any such restriction, and therefore, it can more effectively exploit the richness in the network topologies.

c. Host Mobility Support. Location/Identifier split is the key to provide seamless support for host mobility [16]. Several of the recent proposals such as VL2 [18] and SEATTLE implement the location/identity split for the hosts. However, such a split requires the mapping of identifier to the location during the communication. To achieve this, SEATTLE requires the switches to implicitly store the mappings and use them to locate the destination hosts during the data forwarding. This adds a significant overhead in terms of the packet forwarding at first-hop switches, which was already noted in [2]. In case of VEIL and VL2, hosts are required to explicitly perform the identifier to location mapping for the destination hosts. VL2 achieves this by modifying the host network stack to perform the mapping. On the other hand, VEIL leverages the existing layer-2 semantics to achieve this mapping, whereby, a host uses ARP to map the IP address of the destination host to the 48-bit long *vid* (or Ethernet address if they are connected to the same VEIL switch). In order to avoid the network wide flooding of ARP packets, switches in VEIL capture these ARP packets, and use a unicast based forwarding to perform the resolution in a DHT style.

5.2.3 VEIL-click: Design Overview

veil-click is a practical realization of VIRO using *Virtual Ethernet Id Layer* (in short, VEIL) prototyped using Click Modular Router framework. Figure 5.1 shows an example of a large-scale layer-2 network created using *veil-click* based switches. As seen in this figure, all the host-device that connect to this network can be assigned an IP address using a single IP address

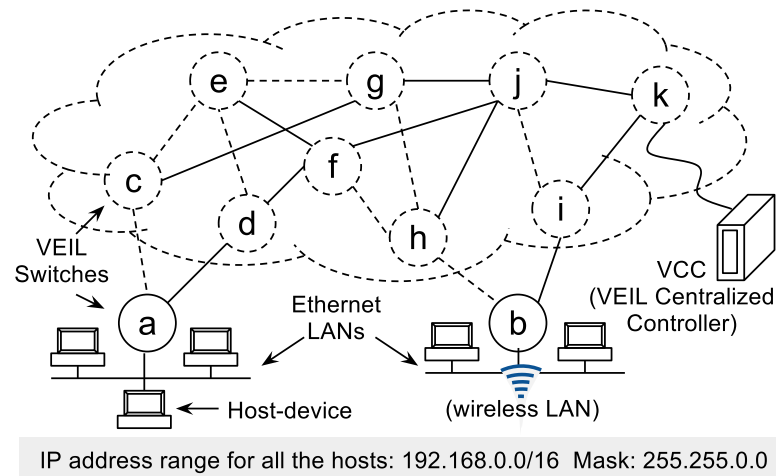


Figure 5.1: An example showing large-scale layer-2 network using VEIL-Click.

block. Therefore, it avoids the partitioning of the network into multiple subnets as performed in traditional layer-2/layer-3 networks. It enables the seamless mobility support for the *host-devices*, since they do not need to change their IP addresses, when they move within the network. As seen in Figure 5.1, the network consists of several *veil-switches* shown using circles, a VEIL centralized controller (in short *vcc*), and a large number of *host-devices* which can either directly connect to *veil-switches* or through Ethernet based Wired/Wireless LAN switches. The roles of each of these devices are as follows.

vcc. It bootstraps the network by performing the initial *vid*-assignment to *veil-switches*. In addition, it also assigns a *vid* to a new *veil-switch* that joins the network after the initial bootstrapping process. Although, VEIL uses a centralized *vid*-assignment for the *veil-switches*, it does not create a single point of failure. This is because once the *vid*-assignment to *veil-switches* is done, they do not need *vcc* for the packet forwarding and other related tasks. Moreover, it is possible to have additional *vcc* connected to the network for redundancy.

veil-switch. A *veil-switch* is the Click based prototype switch, which performs all the actions required to perform the routing and forwarding among the *veil-switches* using VIRO routing protocol. It also performs the *vid*-assignments for the *host-devices* connecting to it directly (or through Ethernet switches).

host-device. It represents any end-host device that connects to the network such as desktop/laptop computers, smartphones etc.

There are three key components of *veil-click*, namely, *vid*-assignment, routing and forwarding, and address resolution. *vid*-assignment to *veil-switches* is done by the *vcc*. To achieve this *veil-switches* send their neighbor information to the *vcc*, which computes the *vid* for the switch based upon its location in the topology. While, *host-devices* are assigned 48-bit long *vid* by the *veil-switch* they directly connect to. Therefore, no manual configuration is required to connect a *host-device* or a *veil-switch* to the network. The routing among the *veil-switches* is performed using the *vids* except at the edges where packet is forwarded to/from a *host-device*. Since we use a host-agnostic design, *veil-click* does not require any modifications to *host-devices* to communicate with each other using *veil-switches*, we intercept and use ARP packets sent by the hosts to resolve the IP addresses to *vids* instead of mapping them to the actual MAC addresses. The only exception to such handling of ARP packets is the case when two hosts are connected to the same *veil-switch* at the same interface, in which case, they directly use MAC addresses to communicate with each other and no intermediate *veil-switches* are involved in the communication. When a *host-device* moves in the network and connects to a different *host-switch*[1] , its IP address may remain the same, and its updated *vid* is pushed to other hosts that it is talking to as an ARP reply packet. Therefore, the mobility does not interrupt the network connection for the host.

5.3 VEIL-click: Key Components

veil-click consists of three essential components, namely, *vid*-management, routing and forwarding, and *host-device* namespace management. In this section we provide a brief description of each of these components.

5.3.1 vid Management

In *veil-click* the *vid*-assignment for the *veil-switches* is performed by the *vcc*. To achieve this, an 'in-band' communication channel to communicate with *vcc* is required. In the following we first describe the protocol used by *veil-switches* to construct and maintain the spanning tree used to communicate with *vcc*, and then the algorithm used by *vcc* to perform the *vid*-management.

[1] A *host-switch* for a *host-device* is the *veil-switch* that it is directly connected to.

vcc Communication Protocol

This protocol consists of two key operations: i. broadcast of the current "best" path to reach *vcc* to directly connected physical neighbors, and ii. subscription to one of the physical neighbors which advertises the "best" path. If a *veil-switch* has an outgoing interface that connects to *vcc*, it announces this information to all its other neighbors by advertising itself as an immediate upstream node to reach *vcc* and its distance to the *vcc* (as number of hops) is 1. Similarly, *vcc* advertises the distance 0 to *veil-switches* directly connected to it. Whenever, a *veil-switch* receives this advertisement it compares the advertised distance with the distance advertised by its current upstream node to reach the *vcc*. If the advertised distance is smaller then it installs the advertising node as its new upstream node to reach *vcc*, and sends a "subscription" packet to the node indicating that it is the downstream node for the node. When a node receives a "subscription" packet from one of its neighbor nodes, it installs that node as one of the downstream nodes.

A node uses the "upstream node" to forward the packet to the *vcc*, on the other hand downstream nodes are used to forward the packets coming from *vcc* to all the nodes in the network. Therefore, each node maintains a list of upstream nodes, which are the nodes that advertise the smallest cost to reach the *vcc*. If there are more than one such nodes then a node a can use any one of them to reach *vcc*. Similarly, it explicitly notifies each of its upstream nodes using the subscription request. The nodes that send a subscription request to a given node become the downstream nodes of it. We show this communication channel for the example network in Figure 5.1 using the continuous black lines.

vid-assignment

When a node learns its upstream node to reach *vcc*, it publishes its neighbor information to *vcc*. Upon receiving the neighbor information from a node, *vcc* stores it in its local database to construct the complete topology of the network, which is then used to perform the *vid*-assignment to *veil-switches* using the top-down graph partitioning based approach [67]. The key idea behind this *vid*-assignment process is to recursively partition the network graph into two sub-graphs using graph mincut. This mincut ensures the minimum possible number of edges between the two sub-graphs in the original graph. After each graph partition, nodes in new sub-graphs are appended a 0 or 1 suffix to their current *vids* respectively. This process

continues till we have only one node in each sub-graph. In our current implementation we use hMETIS [68] tool developed by Karypis *et al.* for efficiently partitioning the large graph structures. We also refer to this approach as the *top-down* since *vids* of each node grow from left to right by appending a bit to the *vids* at each step. In other words the *virtual binary tree* representing the *vid*-assignment is actually constructed from root to leaf nodes. Since, this approach tries to optimize the mincut toward the root of the virtual binary tree, it leaves enough empty slots at the leaves of the virtual binary tree, which is useful if network is constantly expanded by adding a very small number of nodes at a time to existing network.

We have also developed another alternate approach to perform the *vid*-assignment. In this approach, we start from each node as a single cluster, and recursively cluster the nodes to form a *super-node*[2] , such that there is at least one node in each cluster which are directly connected to each other. After each round, we prepend a 0 or 1 to clustering nodes. Since *vids* for nodes grow from right to left, we refer to this approach as the *bottom-up* clustering based *vid* assignment approach. We can perform several optimization to achieve a more balanced virtual-binary tree using bottom-up approach. One such optimization is to give preference to super-nodes which are smallest in size at each round of recursive clustering. This tries to ensure the similar size for each super-node during the process, which in turn results in the more vacant *vid* bits at the most significant part of the *vids*. This is in particular beneficial if a network is expected to grow over time by merging multiple networks together. A more conservative approach may use a hybrid of top-down and bottom-up approach to allow enough empty slots on both sides of the *vids*.

Our current default implementation of *vcc* uses the bottom-up clustering based approach to perform the initial *vid*-assignment. However, it can be easily modified to use a more efficient graph-partitioning approach in future. Using this approach *vcc* performs a *vid*-assignment for all the network interfaces of each *veil-switch* in the network topology. At the end of this process, 32-bit long *vid*-assignments are dispatched to corresponding *veil-switches* as a unicast process. When a new *veil-switch* joins the network after initial *vid*-assignment process, it first learns the upstream node to reach the *vcc* from its neighbors, and sends it neighbor information to *vcc*, which then assigns a *vid* to this node based upon its neighbor's *vid*. On the other hand when a *host-device* connects to a *veil-switch*, the *veil-switch* detects it by sniffing on the data packets sent by the device. Whenever it detects a new host IP address (ip) it assigns a unique 48-bit

[2] A super-node is essentially a group of nodes, which is considered as representative node for the group in the next round. Any node in a super-node is either physically directly connected to other nodes in the super-node or through other nodes in the same super-node.

long $hostvid$ to it by appending unique 16-bits to its 32-bit long switch-*vid* and pushes the mapping (ip, $hostvid$) to the *access-switch*[3] corresponding to IP address ip.

5.3.2 Routing & Forwarding

After the initial *vid*-assignment is performed by the *vcc*, each node periodically runs the bottom-up routing table construction process as described in [43]. Since, *vids* are topology-aware, each node summarizes the routing entry to reach other nodes using 32 unique *vid* prefixes, and therefore only need to store a maximum of 32 routing entries[4] . In order to forward a packet to a given destination, a *veil-switch* uses the routing entry corresponding to the longest matching *vid* prefix. In addition, the implementation of VIRO in *veil-click*, also extends the basic routing protocol to enable the multi-path routing and as well as fast-failure re-routing by having multiple routing entries at each *vid* prefix level. However, as mentioned in VIRO [43], it needs to be done carefully. Otherwise, it may cause the forwarding loops in the data plane if different *veil-switches* on the path choose conflicting forwarding entries. To avoid these loops, *veil-switches*, also include a forwarding identifier on the packet in the form of a shim-layer between the layer-2 and layer-3 headers. It ensures that all the nodes on the path choose a consistent forwarding entry for a given data packet.

Whenever, a *host-device* sends a packet to the network, *host-switch* detects it by looking at the source address in the Ethernet header and overwrites the source MAC address by the *vid* for the host, before forwarding it to other *veil-switches*. Similarly, if a switch receives a packet which has the destination MAC address as one of its *host-devices*' *vid*, it overwrites the destination address in the Ethernet header by the actual MAC address of the host before forwarding the packet to the host.

5.3.3 Host-device Namespace Management

veil-switches detect the *host-devices* directly connected to them by sniffing on the packets sent by them. Whenever, they discover a new *host-device*, they assign a unique *vid* to them, and push the (*vid*, ip) mapping to the *access-switch*. Also *host-switches* publish these mappings

[3] An access-switch for a host is the *veil-switch* whose *vid* is closest to the 32-bit long hash value of its IP address. Here, closeness is measured using the XOR distance.

[4] Number of routing entries in the routing table of a node in VIRO is same as the number of bits used to represent the *vids*. Since we use a 32-bit long *vid* strings in *veil-click* for the *veil-switches*, hence a maximum of 32 routing entries in the routing table.

periodically to *access-switches* corresponding to the mappings. When a *host-device* sends an ARP request message to resolve an IP address, the *host-switch* extracts the IP address (ip) in the request, and if it is not one of its *host-devices*, it forwards the query to the *access-switch* corresponding to ip as an encapsulated ARP request message. Upon receiving the encapsulated ARP request *access-switch* looks up the requested IP address in the mappings stored by it and replies back with the *vid* if found, else it discards the request.

5.4 VEIL Features

veil-click uses a unique modular design, which can be easily extended to incorporate several valuable features for large-scale layer-2 networks. These features not only simplify the design and management of large scale networks, but also provide many additional new features. In the following we describe some of these features.

5.4.1 No Manual Configuration

veil-click enables a completely manual configuration free setup for *veil-switches* and *host-devices*. Whenever a new *veil-switch* is connected to the existing network topology, it automatically learns the path to reach *vcc* by communicating with its neighbors, *vcc* then assigns a *vid* to the switch. Similarly, when a *host-device* connects to the network, it can communicate with any other *host-device* in the network using conventional TCP/IP based protocols, and therefore does not require any modification. In addition, *veil-click* allows a *host-device* to have a static and persistent IP address irrespective of which *veil-switch* it connects to, therefore no configuration is needed. On the other hand, it hugely simplifies the design of DHCP servers, by providing the flexibility of having a single large IP address pool for all the devices that connect to the network, and does not require the manual division of IP address pool to different Ethernet LANs in the network.

5.4.2 Policy Control using Custom Namespace Resolutions

Unlike traditional Ethernet networks, where ARP request messages are broadcasted in the network, *veil-click* allows broadcast-free IP address resolutions using a DHT style look up and store service. It not only helps in significantly reducing the overhead of ARP, but also allows

flexibility in restricting the "unwanted network" traffic for the hosts based on some pre-defined policies. This can be achieved by denying ARP resolutions based on the preset policies, which can take into account the identities of both source and destination hosts to make the resolution. For instance, if network policy prohibits the communication between two hosts, identified using their IP addresses, then ARP request can be dropped right away by the corresponding *access-switch*. Such an approach has additional advantage of pruning the traffic at the source itself by not providing the correct *vid* (as a MAC address), and therefore the host would assume the destination is nonexistent and would not even send the data packet to the network.

5.4.3 Robust Host Mobility Support

veil-click allows hosts to keep the same IP address when they move from one *veil-switch* to another. In addition, it provides a smooth hand-off during the transition, such that it does not interrupt the network connection between the two hosts, if one of them changes their *host-switch*. It is achieved through "push" based notifications used by *veil-click*. When a *host-device* changes its *host-switch* by connecting to a different *veil-switch* it gets a new *vid* based upon the new switch's *vid*. This new *vid* is pushed using an unsolicited ARP reply packet to all the other hosts the host was talking to.

5.4.4 Multi-path Routing & Fast Failure-Rerouting

Topologies for data-center networks and large-scale enterprise networks in general have rich path diversity. These topologies are designed to allow multiple paths to connect any pair of nodes for load-balancing and robustness. We leverage on VIRO's natural and seamless support for multi-path routing and fast failure re-routing to enable these features for VEIL based Ethernet networks.

In case of our current prototype implementation, we utilize multiple routing entries in the routing table at each bucket level to enable an m-way multi-path routing by simply learning and installing m-1 additional routing entries in addition to the default routing entry to reach each bucket level. Next we perform, a valiant load balancing by randomly distributing the traffic on all the m different gateways at each level. Unlike ECMP [66], additional routing entries in case of *veil-click* are not limited to only shortest path routes, and therefore provide more flexibility in multi-path routing. In addition, in case of link/node failures, *veil-click* allows the

local re-routing of traffic by using additional routing entries at each bucket level.

5.5 VEIL-CLICK: The Prototype

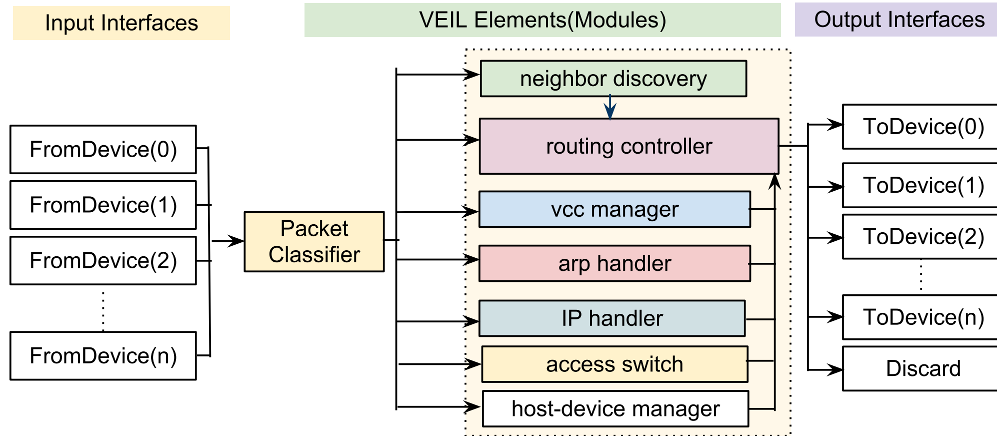


Figure 5.2: An overview of the design of Click based prototype.

In this section we describe our initial prototype using Click Modular Router [42], which implements the complete version of *veil-switches* and *vcc* described in this chapter. We also present initial evaluation results of the current prototype using real *host-devices*.

5.5.1 Basic Design Modules

veil-click is based on Click modular framework, where different functionalities are broken in to individual modules, which can be developed independently, and plugged-in together to compose a full fledged *veil-switch*. In case of *veil-click*, we build several elements (or modules) for various operations such as ARP handling, Routing Table management etc. as Click elements. These individual elements are connected together to form various components of *veil-click*, which are shown in Figure 5.2. In this figure rectangular boxes represent the Click elements (or a group of elements for simplified representation), while solid black lines represent the interconnections between them.

In Figure 5.2, FromDevice reads the incoming packets from a physical interface, and ToDevice element writes the packets to interfaces. The PacketClassifier element classifies the packet into different categories and passes them to the relevant element(s). There are five key groups

of elements in *veil-click*, namely: a) *Link Discovery Element* which performs the neighbor discovery via the periodic exchange of HELO packets. b) vcc *Manager* Elements in this group are responsible to exchange information with the *vcc*, and get the *vid*-assignment from the *vcc* using *vcc* communication protocol. c) *Routing Controller* These elements perform the necessary routing table construction tasks, and perform the packet forwarding tasks. d) *Data packet handlers* These elements handle the various types of data packets sent by the hosts, such as ARP and IP packets. e) *Host-device Manager* Elements in this group are responsible for various host related tasks such as, assigning the *vids* to the *host-devices* directly connected to the switch, publish and store the mapping to *access-switches* etc.

A detailed description of the code, readme instructions, and basic design details are available at our online code repository [69]. Below is a sample `click` script to realize an instance of *veil-click* switch.

A sample click script to realize a *veil-click* switch instance.

```
require(veil);

//elements to store state information.
interfaces::VEILInterfaceTable(
                    000000000000,08:00:27:7b:bb:b3,
                    000000000000,08:00:27:60:cb:86,
                    000000000000,08:00:27:8e:51:cd,
                    000000000000,08:00:27:6e:7d:bd,
                    UseStatic false,
                    PRINTDEBUG false
                    );
hosts::VEILHostTable(PRINTDEBUG true);
neighbors::VEILNeighborTable(PRINTDEBUG false);
mapping::VEILMappingTable(PRINTDEBUG true);
rendezvouspoints::VEILRendezvousTable(PRINTDEBUG false);
routes::VEILRouteTable(INTERFACETABLE interfaces, PRINTDEBUG false);
topo::VEILNetworkTopoVIDAssignment(VCCMAC 08:00:27:6e:7d:bd,
```

```
PRINTDEBUG false);

// output devices
out0::ToDevice(eth2);
out1::ToDevice(eth3);
out2::ToDevice(eth4);
out3::ToDevice(eth5);
q0 :: Queue -> out0;
q1 :: Queue -> out1;
q2 :: Queue -> out2;
q3 :: Queue -> out3;

// input devices
in0::FromDevice(eth2);
in1::FromDevice(eth3);
in2::FromDevice(eth4);
in3::FromDevice(eth5);

// hello generator
hellogen::VEILGenerateHelloNew (interfaces, PRINTDEBUG false);
hellogen[0]->q0;
hellogen[1]->q1;
hellogen[2]->q2;
hellogen[3]->q3;

c::Classifier(12/9876 26/0000,      // 0. VEIL_HELLO
              12/9876 26/0403,      // 1. VEIL_RDV_PUBLISH
              12/9876 26/0404,      // 2. VEIL_RDV_QUERY
              12/9876 26/0402,      // 3. VEIL_RDV_REPLY
              12/9876 26/0602,      // 4. VEIL_ENCAP_ARP
              12/9876 26/0201,      // 5. VEIL_MAP_PUBLISH
              12/9876 26/0202,      // 6. VEIL_MAP_UPDATE
```

```
                12/9876 26/0203,      // 7. NO_VID_TO_ACCESS_SWITCH
                12/9876 26/0601,      // 8. VEIL_ENCAP_IP
                12/9876 26/0801,      // 9. VEIL_ENCAP_MULTIPATH_IP
                12/9878,              // 10. ETHERTYPE_VEIL_IP
                12/0806,              // 11. ETHERTYPE_ARP
                12/0800,              // 12. ETHERTYPE_IP
                12/9876 26/0100%FF00, // 13. VCC Packets.
             -);                      // 14. Everything Else (DISCARD)
//set annotation for the incoming packets
in0 -> VEILSetPortAnnotation(0) -> c;
in1 -> VEILSetPortAnnotation(1) -> c;
in2 -> VEILSetPortAnnotation(2) -> c;
in3 -> VEILSetPortAnnotation(3) -> c;

// router to route packets using the routing table,
// and neighbor information
router::VEILRoutePacket(hosts, routes, interfaces, neighbors,
PRINTDEBUG false);

//need Queue to convert from push to pull
router[0] -> q0;
router[1] -> q1;
router[2] -> q2;
router[3] -> q3;
router[4] -> c;
router[5] -> Discard;

//Elements to perform the VCC related tasks
datasAd::TimedSource(DATA \<ffffffff ffff 0800 276e 7dbd 9876 0800
276e 7dbd 000000000000 0101 0200 0800 276e7dbd 0000>, INTERVAL 5) -> c;
vccstate::VEILSpanningTreeState(08:00:27:6e:7d:bd 08:00:27:6e:7d:bd 0,
PRINTDEBUG false);
```

```
vccgenerator::VEILGenerateVCCSTAdSub(interfaces, neighbors, vccstate,
PRINTDEBUG false);
vccprocessor::VEILProcessVCCSTAdSub(INTERFACETABLE interfaces,
NEIGHBORTABLE neighbors, SPANNINGTREESTATE vccstate, NETWORKTOPO topo,
PRINTDEBUG false);
vidgenerator::VEILGenerateVIDAssignmentPackets(INTERFACETABLE interfaces,
NEIGHBORTABLE neighbors, SPANNINGTREESTATE vccstate, NETWORKTOPO topo,
PRINTDEBUG false);
vccgenerator[0] -> q0;
vccgenerator[1] -> q1;
vccgenerator[2] -> q2;
vccgenerator[3] -> q3;
vccgenerator[4] -> c;

vidgenerator[0] -> q0;
vidgenerator[1] -> q1;
vidgenerator[2] -> q2;
vidgenerator[3] -> q3;
vidgenerator[4] -> c;

vccprocessor[0] -> q0;
vccprocessor[1] -> q1;
vccprocessor[2] -> q2;
vccprocessor[3] -> q3;

c[13] -> vccprocessor;
c[0] -> VEILProcessHello(neighbors, interfaces,PRINTDEBUG false);

prdv::VEILProcessRDV(routes, rendezvouspoints, interfaces,
PRINTDEBUG false) -> router;

c[1] -> prdv;
```

```
c[2] -> prdv;
c[3] -> prdv;

parp::VEILProcessARP(hosts, mapping, interfaces,
PRINTDEBUG true) ->  router;

c[11]  -> ARPPrint ->  parp;
c[4] -> Print (VEIL_ARP, MAXLENGTH 100) ->  parp;

paci::VEILProcessAccessInfo(hosts,mapping, interfaces,
PRINTDEBUG true) -> router;
c[5] -> paci;
c[6] -> paci;
c[7] -> paci;

pip::VEILProcessIP(hosts, mapping, interfaces,FORWARDING_TYPE 2,
PRINTDEBUG true)-> router;

c[8]  -> pip;
c[9]  -> pip;
c[10]  -> pip;
c[12] -> pip;

VEILBuildRouteTable(neighbors, routes, interfaces,
PRINTDEBUG false) -> router;

VEILPublishAccessInfo(hosts, PRINTDEBUG true) -> router;

c[14] -> Discard;
```

Figure 5.3: An initial prototype of *veil-switch*.

5.5.2 Initial Evaluation

At the time of writing of this chapter, we have finished the basic implementation of *veil-click*, which includes all the functionalities mentioned in the thesis. Fig. 5.3 shows our initial prototype switch. We have tested the basic working of the current prototype using a small testbed in our lab. Fig. 5.4 shows the basic set up for the testbed, which consists of 5 server machines with 5 Ethernet ports on each as *veil-switches*, 3 wireless access points which are directly connected to different *veil-switches* through wired Ethernet cables, and several laptop computers running Ubuntu/MAC OS X/Windows operating systems as test *host-devices*. We do not modify anything on the *host-devices* to connect them to the network, and no additional software is installed to allow them to communicate with the network. The complete test network is configured to use a single subnet prefix, and the *host-devices* are assigned static IP addresses.

Mobility Support: Using this testbed we evaluated the support for host-mobility by connecting the *host-devices* to different wireless access points, while they are communicating with each other. For these experiments, we set up a TCP connection between two *host-devices*, where source *host-device* generate the traffic at constant bit rate (1600 kbps), and move the destination *host-device* between two different wireless access points. In Figure 5.5 we show how the mobility of the destination *host-device* affects its traffic receiving rate. In this figure, red dashed lines represent the transitions from one access point to another, and solid blue lines represent the average good-put every second at the destination *host-device*. In Figure 5.5(a) we show how rate changes when destination *host-device* for multiple such transitions. As seen in this

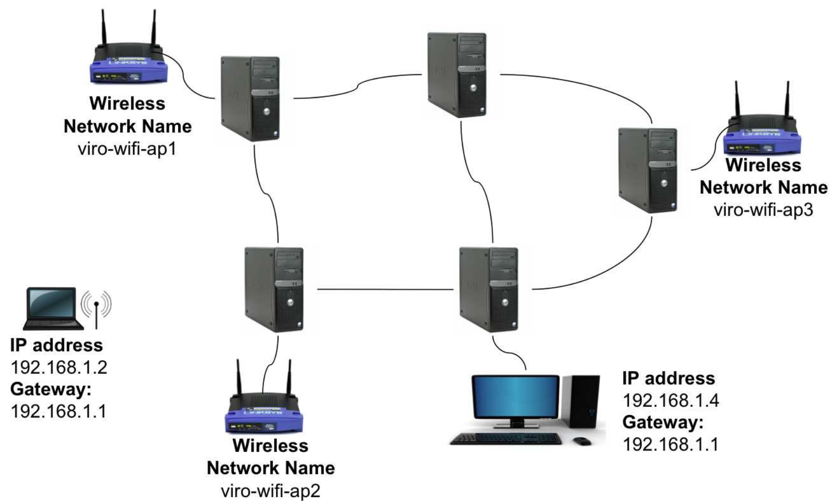


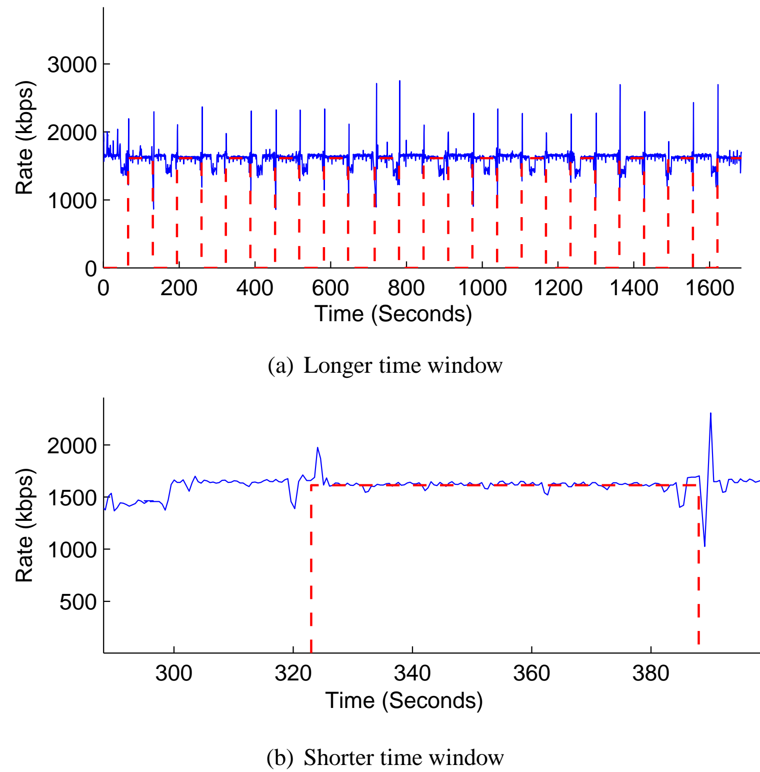
Figure 5.4: Intial testbed created using prototype *veil-switches*.

figure, the mobility causes minimal disruption for the TCP connection. We zoomed in on these individual transitions, and show one such transition in Figure 5.5(b). Our results show that TCP connection stabilizes with in 2-5 seconds during all such network transitions.

Multipath: Next, we set up experiments to evaluate the efficacy of multi-path routing for our *veil-click* prototype switch. For this experiment we use the same testbed as shown in Fig. 5.4. For our experiment we attached the `traffic source` which was connected directly to `viro-wifi-ap3` using a wired link. We connected the `traffic sink` to `viro-wifi-ap1`. We also capped the limit on maximum traffic on any of the link to 1200kbps, except for the link connecting wireless access routers to the corresponding *veil-switches*. In these experiments both traffic source and sink set up a TCP connection to transfer an infinitely long file, and we monitor the average TCP throughput at sink. We consider two types of setup for the experiments.

- *Single Path* Routing: In this case, we use the basic VIRO routing with no multi-path feature.
- *Multi-Path* Routing: We enable multi-path routing feature in VIRO. To achieve this, each node tries to distribute the incoming traffic to all the possible gateways for the given destination in equal proportions.

Fig. 5.6 compares the TCP throughput for both these setups. In this plot, x-axis represents



(a) Longer time window

(b) Shorter time window

Figure 5.5: Host-device mobility and the traffic bit rate.

time and y-axis represents the average receiving rate at sink for every 5 seconds. As seen in this plot, using single path limits the throughput to an average value around 1100kbps, but multi-path increases the net throughput by more than 30% to an average value of 1450kbps. Although, it is still significantly smaller than the maximum possible throughput of 2400kbps. The reason for this is the unequal costs for the different paths used in multi-path scenario, which creates noise for the round trip delay measurements for TCP, hence it limits the actual throughput for our experiments. In our experiments, there were two possible paths from traffic source to sink, and the round trip delay for these paths were 1.6ms and 2.8ms in case of zero queueing delay (i.e. when there is no congestion in the network, therefore the buffers at each *veil-switch* are empty).

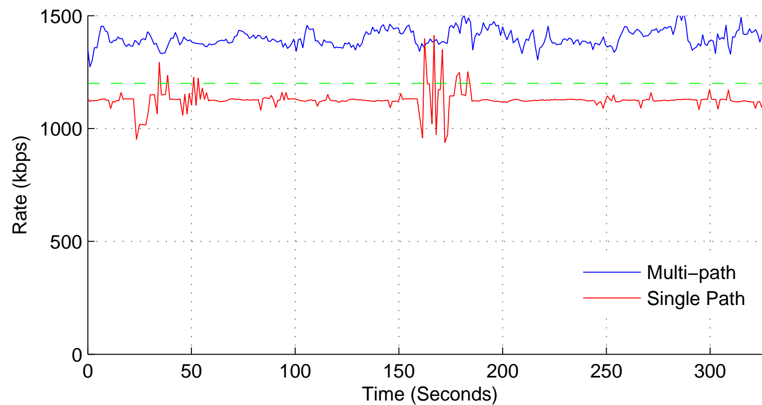


Figure 5.6: TCP throughput comparison.

5.6 Conclusion

In this chapter we presented a network architecture to create large-scale plug-&-play networks. It is designed with two broad sets of goals: i) to support – *with minimal manual configuration* – *large, dynamic* networks which connect tens or hundreds of thousands of diverse devices with *rich physical topologies* and reduce the management overhead; 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, 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.

Toward these goals, we demonstrated our initial prototype *veil-click*, which is built using Click Modular Router framework. *veil-click* aims to significantly simplify the current management overhead for large-scale enterprise networks by automating most of the network configurations for both *host-devices* and as well as the routing nodes in the network. In addition, it provides built-in mechanisms for multi-path routing, fast failure re-routing, and seamless mobility support. Finally, the source-code for *veil-click* is publicly available, and can be downloaded from our Google Code project repository [69].

Chapter 6

Conclusion and Discussion

In this thesis, we presented the several challenges faced by existing networking architectures. We also presented the solution in the form of VIRO and VEIL network architectures. In the following we present our concluding remarks and future work.

6.1 Conclusion

In this thesis we first presented VIRO — a novel routing architecture for large-scale networks. The key idea in our design is to introduce a *topology-aware*, structured virtual id (vid) space onto which both physical identifiers as well as higher layer addresses/names are mapped. VIRO completely eliminates *network-wide flooding* in both the data and control planes, and thus is highly scalable and robust. 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. Our evaluation of VIRO using many synthetic and real topologies shows the immense scalability and robustness of VIRO, while keeping the overheads very low. In addition, we also establish the correctness of basic VIRO routing algorithm using theoretical proofs, which are presented in Appendix A.

We also introduced VEIL — a realization of VIRO for large scale Ethernet (layer-2) networks. VEIL provides a scalable, robust and efficient Ethernet network architectures suitable for creating plug-&-play based advanced Ethernet networks. The presented design architecture does not require any changes to existing *host-devices*, and can be deployed in an incremental

manner using existing Ethernet switches. The key idea behind this design is to assign a 48-bit long *vid* to *host-devices* and as well as to *veil-switches*, so that it the source/destination Ethernet address field can be reused in the Ethernet headers, without requiring any modifications to existing Ethernet headers.

We have successfully implemented the VEIL routing architecture using the Click modular router framework. In this implementation different functionalities of VEIL are implemented as independent modules, which are written in C++. In order to realize an instance of *veil-switch*, we added multiple network interface cards to Linux based server machines and run *veil-click* on top of them. We also deployed a small testbed using these *veil-switches*, which are currently in use. Our experiments demonstrate that current implementation can support seamless mobility for the *host-devices* with minimal interruption in terms of the packet losses during the transition. Furthermore, current prototype also provides support for many advanced features, such as, multi-path routing and support for DHCP based device configuration.

6.2 Future Work

There are several possible extensions for the VIRO routing architecture presented in this thesis. For example, one may extend the basic VIRO routing protocol by introducing multiple rendezvous nodes at each level. This will provide additional robustness to handle the failures of rendezvous nodes. Similarly, one could use a variety of mechanisms to achieve fast-failure re-routing and multi-path routing using VIRO, evaluation of each of these mechanisms with different traffic patterns is part of current on-going work.

Similar to VLANs, VIRO can easily support multiple virtual topologies or virtualized networks on top of the same physical network substrate to enhance security, robustness, performance or network isolation. For example, we can construct multiple vid spaces and build routing tables for each *vid* space. Comparing existing layer-2 VLANs or layer-3 virtual topology routing, VIRO has the ability to potentially support far larger number of virtual topologies or virtualized networks, since VIRO routing tables are small and their computation incurs relatively little overheads.

Finally, we would like to remark that VIRO is designed as a new routing and forwarding paradigm for a *single network domain* to better support the availability, robustness, mobility, manageability and security requirements of emerging and future networks and applications. It does not explicitly address the *inter-domain routing* problems plaguing today's Internet. Nonetheless, we believe that adoption of VIRO for intra-domain routing and data forwarding will help alleviate the inter-domain routing issues. Also, the extension of VIRO routing architecture to address the inter-domain routing problems requires further research and is part of our future work.

References

[1] Ietf trill working group. http://www.ietf.org/html.charters/trill-charter.html accessed on: Apr 21 2011.

[2] Changhoon Kim, Matthew Caesar, and Jennifer Rexford. Seattle: A scalable ethernet architecture for large enterprises. *ACM Trans. Comput. Syst.*, 29:1:1–1:35, February 2011.

[3] Malcolm Scott and Jon Crowcroft. Moose: Addressing the scalability of ethernet. In *EuroSys 2008: Poster Session*, 2008. http://www.dcs.gla.ac.uk/Conferences/EuroSys2008/posters/11_Malcolm_Scott.pdf accessed on: Apr 21 2011.

[4] C. Kim and J. Rexford. Revisiting Ethernet: Plug-and-play made scalable and efficient. In *15th IEEE Workshop on Local & Metropolitan Area Networks, 2007. LANMAN 2007*, pages 163–169, 2007.

[5] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38:69–74, March 2008.

[6] A. Myers, E. Ng, and H. Zhang. Rethinking the service model: Scaling Ethernet to a million nodes. In *Proceedings of ACM SIGCOMM HotNets*, 2004.

[7] R. Perlman. Rbridges: transparent routing. In *INFOCOM 2004. Twenty-third Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 2, pages 1211–1218. IEEE, 2004.

[8] S. Ray, R. Guerin, and R. Sofia. A distributed hash table based address resolution scheme for large-scale Ethernet networks. In *IEEE International Conference on Communications, 2007. ICC'07.*, pages 6446–6453. IEEE, 2007.

[9] T.L. Rodeheffer, C.A. Thekkath, and D.C. Anderson. SmartBridge: A scalable bridge architecture. *ACM SIGCOMM Computer Communication Review*, 30(4):205–216, 2000.

[10] S. Sharma, K. Gopalan, S. Nanda, and T. Chiueh. Viking: A multi-spanning-tree Ethernet architecture for metropolitan area and cluster networks. In *INFOCOM 2004. Twenty-third Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 4, pages 2283–2294. IEEE, 2004.

[11] Cisco fabricpath. `http://www.cisco.com/en/US/prod/collateral/switches/ps9441/ps9402/white_paper_c11-605488.pdf` accessed on: Apr 21 2011.

[12] An ethernet address resolution protocol. `http://tools.ietf.org/html/rfc826`.

[13] J. Moy et al. OSPF Version 2, 1994.

[14] D. Oran. RFC1142: OSI IS-IS Intra-domain Routing Protocol. *RFC Editor United States*, 1990. `http://tools.ietf.org/html/rfc1142`.

[15] J. Saltzer. RFC1498: On the Naming and Binding of Network Destinations. *RFC Editor United States*, 1993.

[16] David Meyer. LISP, The Internet Protocol Journal, Volume 11.

[17] R. Moskowitz and P. Nikander. RFC 4423: Host identity protocol (HIP) architecture. *Internet Request for Comments*, 2006.

[18] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: a scalable and flexible data center network. *Commun. ACM*, 54:95–104, March 2011.

[19] Matthew Caesar, Tyson Condie, Jayanthkumar Kannan, Karthik Lakshminarayanan, and Ion Stoica. Rofl: routing on flat labels. In *Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '06, pages 363–374, New York, NY, USA, 2006. ACM.

[20] Bryan Ford. Unmanaged internet protocol: taming the edge network management crisis. *SIGCOMM Comput. Commun. Rev.*, 34:93–98, January 2004.

[21] C.E. Perkins, S.R. Alpert, and B. Woolf. *Mobile IP: Design Principles and Practices*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1997.

[22] Ethernet Spanning Tree Protocol (STP). `http://en.wikipedia.org/wiki/Spanning_Tree_Protocol` accessed on: Apr 21 2011.

[23] M. Goyal, KK Ramakrishnan, and W. Feng. Achieving faster failure detection in OSPF networks. In *IEEE International Conference on Communications, 2003. ICC'03.*, volume 1, pages 296–300. IEEE, 2003.

[24] J. Rexford, J. Wang, Z. Xiao, and Y. Zhang. BGP routing stability of popular destinations. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet Measurment*, pages 197–202. ACM, 2002.

[25] A. Feldmann, O. Maennel, Z.M. Mao, A. Berger, and B. Maggs. Locating Internet routing instabilities. In *ACM SIGCOMM Computer Communication Review*, volume 34, pages 205–218. ACM, 2004.

[26] B. Fortz, J. Rexford, and M. Thorup. Traffic engineering with traditional IP routing protocols. *IEEE Communications Magazine*, 40(10):118–124, 2002.

[27] B. Fortz and M. Thorup. Internet traffic engineering by optimizing OSPF weights. In *Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE INFOCOM'00*, volume 2, pages 519–528. IEEE, 2000.

[28] A. Sridharan and R. Guerin. Making IGP routing robust to link failures. *Lecture Notes in Computer Science*, 3462:634–646, 2005.

[29] P. Francois and O. Bonaventure. An evaluation of IP-based fast reroute techniques. In *Proceedings of the 2005 ACM conference on Emerging network experiment and technology*, pages 244–245. ACM, 2005.

[30] A. Kvalbein, A.F. Hansen, T. Cicic, S. Gjessing, and O. Lysne. Fast IP network recovery using multiple routing configurations. In *25th IEEE International Conference on Computer Communications. INFOCOM'06*, pages 1–11. IEEE, 2006.

[31] S. Lee, Y. Yu, S. Nelakuditi, Z.L. Zhang, and C.N. Chuah. Proactive vs reactive approaches to failure resilient routing. In *Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies. INFOCOM'04*, volume 1. IEEE, 2004.

[32] S. Nelakuditi, S. Lee, Y. Yu, and Z.L. Zhang. Failure insensitive routing for ensuring service availability. *Quality of ServiceIWQoS 2003*, pages 153–154, 2003.

[33] C. Alaettinoglu, V. Jacobson, and H. Yu. Towards milli-second IGP convergence. *IETF Draft*, 2000.

[34] C. Boutremans, G. Iannaccone, and C. Diot. Impact of link failures on VoIP performance. In *Proceedings of the 12th international workshop on Network and operating systems support for digital audio and video*, pages 63–71. ACM, 2002.

[35] G. Iannaccone, C. Chuah, R. Mortier, S. Bhattacharyya, and C. Diot. Analysis of link failures in an IP backbone. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, pages 237–242. ACM, 2002.

[36] A. Markopoulou, G. Iannaccone, S. Bhattacharyya, C.N. Chuah, and C. Diot. Characterization of failures in an IP backbone. In *Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies. INFOCOM 2004.*, volume 4, pages 2307–2317. IEEE, 2004.

[37] M. Gjoka, V. Ram, and X. Yang. Evaluation of IP fast reroute proposals. In *2nd International Conference on Communication Systems Software and Middleware. COMSWARE 2007.*, pages 1–8. IEEE, 2007.

[38] Albert Greenberg, Gisli Hjalmtysson, David A. Maltz, Andy Myers, Jennifer Rexford, Geoffrey Xie, Hong Yan, Jibin Zhan, and Hui Zhang. A clean slate 4d approach to network control and management. *SIGCOMM Comput. Commun. Rev.*, 35:41–54, October 2005.

[39] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: taking control of the enterprise. In *Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '07, pages 1–12, New York, NY, USA, 2007. ACM.

[40] Martin Casado, Tal Garfinkel, Aditya Akella, Michael J. Freedman, Dan Boneh, Nick McKeown, and Scott Shenker. Sane: a protection architecture for enterprise networks. In *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15*, Berkeley, CA, USA, 2006. USENIX Association.

[41] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. Nox: towards an operating system for networks. *SIGCOMM Comput. Commun. Rev.*, 38:105–110, July 2008.

[42] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M.F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 2000.

[43] Sourabh Jain, Yingying Chen, Saurabh Jain, and Zhi-Li Zhang. Viro: A plug & play, scalable, robust and namespace independent virtual id routing for future networks. In *30th Annual Joint Conference of the IEEE Computer and Communications Societies. INFOCOM 2011*. IEEE, 2011.

[44] S. Jain, Y. Chen, and Z.L. Zhang. VEIL: A Plug-&-Play Virtual (Ethernet) Id Layer for Below IP Networking. In *Below IP Networking workshop in conjunction with IEEE GLOBECOM*, pages 1–6. IEEE, 2009.

[45] Sourabh Jain and Zhi-Li Zhang. Simplifying Manageability, Scalability and Host Mobility in Large-Scale Enterprise Networks using VEIL-click. In *Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE 2011), co-located with USENIX NSDI 2011*. USENIX, 2011.

[46] Dhcp rfc - dynamic host configuration protocol rfc's (ietf). `http://www.bind9.net/rfc-dhcp`.

[47] Z. Kerravala. Configuration management delivers business resiliency. *The Yankee Group*, 2002.

[48] J. McQuillan, I. Richer, E. Rosen, B. Beranek, and N. Inc. The new routing algorithm for the ARPANET. *Communications, IEEE Transactions on [legacy, pre-1988]*, 28(5):711–719, 1980.

[49] Dijkstra's Single Source Shortest Path Algorithm. `http://en.wikipedia.org/wiki/Dijkstra's_algorithm` accessed on: Apr 21 2011.

[50] H. Balakrishnan, K. Lakshminarayanan, S. Ratnasamy, S. Shenker, I. Stoica, and M. Walfish. A layered naming architecture for the internet. In *ACM SIGCOMM Computer Communication Review*, volume 34, pages 343–352. ACM, 2004.

[51] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *Proceedings of IPTPS02*, 2002.

[52] Matthew Caesar, Miguel Castro, Edmund B. Nightingale, Greg O'Shea, and Antony Rowstron. Virtual ring routing: network routing inspired by dhts. In *Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '06, pages 351–362, New York, NY, USA, 2006. ACM.

[53] Lakshmi Ramachandran, Manika Kapoor, Abhinanda Sarkar, and Alok Aggarwal. Clustering algorithms for wireless ad hoc networks. In *DIALM '00: Proceedings of the 4th international workshop on Discrete algorithms and methods for mobile computing and communications*. ACM, 2000.

[54] Y. Yu, G.H. Lu, and Z.L. Zhang. Enhancing location service scalability with HIGH-GRADE. In *IEEE International Conference on Mobile Ad-hoc and Sensor Systems*. IEEE, 2004.

[55] Guor-Huar Lu, Sourabh Jain, Shanzhen Chen, and Zhi-Li Zhang. Virtual id routing: a scalable routing framework with support for mobility and routing efficiency. In *MobiArch Workshop*, 2008.

[56] N. Spring, R. Mahajan, and D. Wetherall. Measuring ISP topologies with Rocketfuel. In *ACM SIGCOMM Computer Communication Review*, volume 32, pages 133–145. ACM, 2002.

[57] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, SIGCOMM '08, pages 63–74, New York, NY, USA, 2008. ACM.

[58] A. Medina, A. Lakhina, I. Matta, and J. Byers. BRITE: An approach to universal topology generation. In *mascots*, page 0346. Published by the IEEE Computer Society, 2001.

[59] S. Sharma, K. Gopalan, S. Nanda, and T. Chiueh. Viking: A multi-spanning-tree Ethernet architecture for metropolitan area and cluster networks. In *INFOCOM 2004. Twenty-third Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 4, 2004.

[60] Thomas L. Rodeheffer, Chandramohan A. Thekkath, and Darrell C. Anderson. Smartbridge: a scalable bridge architecture. In *Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '00, pages 205–216, New York, NY, USA, 2000. ACM.

[61] Radhika Niranjan Mysore, Andreas Pamboris, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, Vikram Subramanya, and Amin Vahdat. Portland: a scalable fault-tolerant layer 2 data center network fabric. *SIGCOMM Comput. Commun. Rev.*, 39(4):39–50, 2009.

[62] Srihari Nelakuditi, Sanghwan Lee, Yinzhe Yu, Zhi-Li Zhang, and Chen-Nee Chuah. Fast local rerouting for handling transient link failures. *IEEE/ACM Trans. Netw.*, 2007.

[63] Karthik Lakshminarayanan, Matthew Caesar, Murali Rangan, Tom Anderson, Scott Shenker, and Ion Stoica. Achieving convergence-free routing using failure-carrying packets. In *SIGCOMM '07*.

[64] Murtaza Motiwala, Megan Elmore, Nick Feamster, and Santosh Vempala. Path splicing. In *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, SIGCOMM '08, pages 27–38, New York, NY, USA, 2008. ACM.

[65] P. Psenak, S. Mirtorabi, A. Roy, L. Nguen, and P. Pillay-Esnault. MT-OSPF: Multi topology (MT) routing in OSPF. *IETF, RFC4915*, 2007.

[66] Multipath issues in unicast and multicast next-hop selection. `http://tools.ietf.org/html/rfc2991`.

[67] Sourabh Jain, Yingying Chen, Saurabh Jain, and Zhi-Li Zhang. 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.pdf.

[68] G. Karypis and V. Kumar. Hmetis: A hypergraph partitioning package. *University of Minnesota*, 1998.

[69] The source code repository for veil-click. `http://code.google.com/p/veil-viro-umn`.

Appendix A

VIRO: Properties and Proofs

In this appendix we discuss the several theoretical properties of VIRO routing framework. Using these properties we derive the proof for the correctness of the basic VIRO routing table construction algorithm. We describe how VIRO ensures loop free forwarding, and how the limited information available to nodes in the form of gateways and nexthop nodes is sufficient to guarantee this.

A.1 Basic Properties

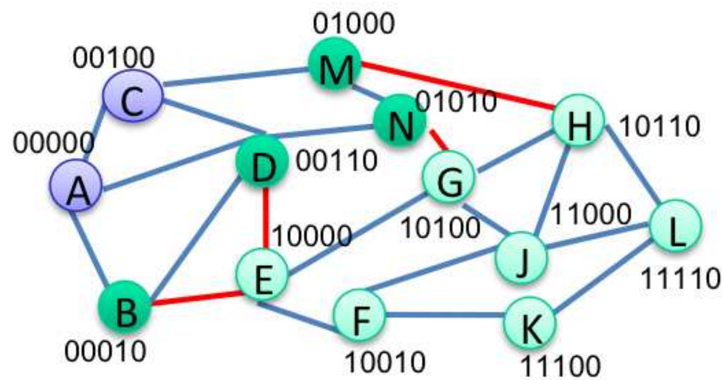


Figure A.1: Multiple gateways to reach $B_5(A)$ for nodes in $S_4(A)$

As mentioned in Chapter 3.4, a *consistent* rule is required to choose a gateway to reach a

given level k bucket for node x to avoid loop free routing. For example consider the topology shown in Fig. A.1. In this figure, any node $x \in S_4(A)$ has four gateways to reach $B_5(A)$. These gateway nodes are $\{B, D, M, N\}$. Let us consider the current routing tables (at the end of round-4) are shown in Table A.1. In the next round (round-5) nodes A and C queries

Routing Table for Node A			
Bucket	Prefix	NextHop	Gateway
1	00001	-	-
2	0001*	B	A
3	001**	C,D	A
4	01***	C	C

Routing Table for Node C			
Bucket	Prefix	NextHop	Gateway
1	00101	-	-
2	0011*	D	C
3	000**	A	C
4	01***	M	C

Table A.1: Routing tables at the end of round-4

their rendezvous node at level-5 for a gateway to reach their level-5 bucket. Now consider that there is no consistent policy for selecting a gateway and any one of the four possible gateways could be chosen by rendezvous node for nodes A and C. Say, A selects M as its gateway to reach its level-5 bucket and C chooses B for the same. After this selection, A's nexthop to reach $B_5(A)$ will be the nexthop to reach $M \in B_4(A)$ i.e. C and similarly C's nexthop to reach $B_5(C)$ will be $B \in B_3(C)$ i.e. node A. (See Table A.2 for the routing tables for nodes A and C after round-5) As seen in Table A.2, the arbitrary selection of gateways leads to a loop $A \rightarrow C \rightarrow A \ldots \rightarrow C \ldots$ between nodes A and C to reach their level-5 bucket. It is possible to avoid such loops by choosing a gateway using a consistent policy at every level. One such consistent policy is to use the logically closest gateway. Next we illustrate the consistent gateway selection policy and a formal proof for *loop-free* routing under this policy.

In the following we provide the proof for the correctness of the routing algorithm shown in Algorithm 2. We show how our proposed routing algorithm is “loop-free” and how it ensures connectivity between any two nodes if they are physically connected. First we explain the strategy to choose “Nexthops” and “Gateways” consistently at different nodes in the network.

Routing Table for Node A			
Bucket	Prefix	NextHop	Gateway
1	00001	-	-
2	0001*	B	A
3	001**	C,D	A
4	01***	C	C
5	1****	C	M

Routing Table for Node C			
Bucket	Prefix	NextHop	Gateway
1	00101	-	-
2	0011*	D	C
3	000**	A	C
4	01***	M	C
5	1****	A	B

Table A.2: Routing tables at the end of round-5

A.1.1 Gateway Selection Strategy

• Step 1 **Get the list of all the gateways**: Each node x say in a given “connected” sub-tree $S_{k-1}(x)$ when trying to reach it’s bucket $B_k(x)$ to form a larger connected sub-tree, learns all the possible gateways by communicating with the rendezvous point $rdv_k(x)$.

• Step 2 **Sort gateways into classes**: x now divides these ‘gateways’ to reach $B_k(x)$ in ‘equivalence classes’ based on the ‘logical distance’. i.e. all the ‘gateway nodes’ which are at the same ‘logical distance’ with respect to x belong to one ‘equivalence class’. Let’s denote a set containing gateways to reach bucket $B_k(x)$, which are at a logical distance of r from x by $G_r^x(B_k(x))$ to reach $B_k(x)$, where $0 \leq r < k$.

• Step 3 **Pick the logically closest**: Next, x picks the smallest r such that $G_r^x(B_k(x)) \neq \emptyset$. Let’s denote this value of r by r_1

• Step 4 **Derive the nexthop**: If the $r_1 > 0$ then $nh_k^x = nh_{r_1}^x$.

If $r_1 = 0$ (i.e. x is the gateway) then one of its physical neighbor $u \in N_k^x$ is the nexthop. i.e. $nh_k^x = u \in N_k^x$

• Every node s in the topology repeats the steps 1-4 for all the values of k starting from 1 to l

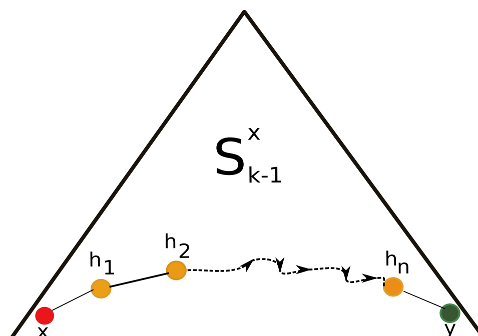


Figure A.2: Illustration of the property 1

A.1.2 Properties

We now illustrate certain properties of the routing protocol, which will be used in the proof later.

Property 1: If $x, y \in S_k(x)$, then physical path from x to y goes through the nodes from $S_k(x)$ only.

i.e. $x \rightsquigarrow y = x \rightarrow h_1 \rightarrow h_2 \ldots \rightarrow h_n \rightarrow y$ where $h_i \in S_k(x)\ for\ 1 \leq i \leq n$. This is because, in our algorithm the connectivity information for $B_k(x)$ is derived using the nodes from S^x_{k-1} only, where $1 \leq k \leq l$.

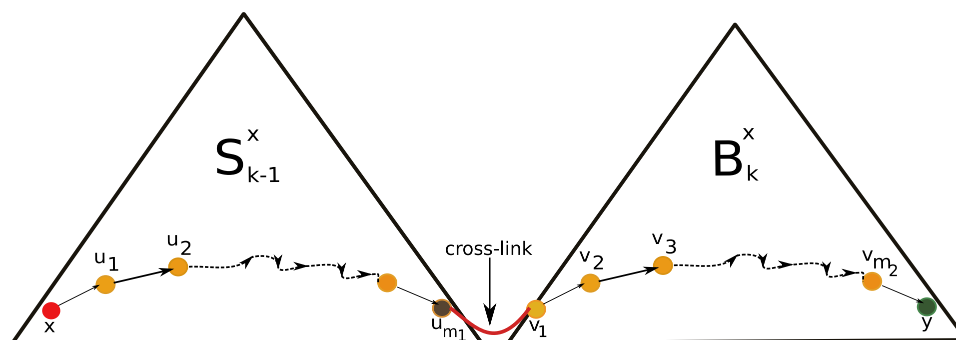


Figure A.3: Illustration of the property 2

Property 2: If $x \in S_{k-1}(x)$ and $y \in B_k(x)$, and the path $x \rightsquigarrow y$ consists of the nodes $u_i \in S_{k-1}(x)$, for $0 \leq i \leq m_1$ and $v_j \in B_k(x)$, for $0 \leq j \leq m_2$. Then there is only one "Cross link" (i.e. a link from $u_i \rightarrow v_j$ or $v_j \rightarrow u_i$) between $S_{k-1}(x)$ and $B_k(x)$.

Proof: Let's say there are two possible "Cross links" on the physical path $x \rightsquigarrow y = x \rightarrow$

$\ldots u_{m_1} \rightarrow v_1 \ldots v_p \rightarrow u_q \ldots \rightarrow y$, here $0 \leq q \leq m_1,\ 0 \leq p \leq m_2$.
Since the destination y lies in $B_k(x)$ and the source lies in $S_{k-1}(x)$, therefore there is at least one cross-link from $S_{k-1}(x)$ to B_k^x. Let's say this cross link is $u_{m_1} \rightarrow v_1$. Now consider the second cross-link $v_p \rightarrow u_q$.
We have $\delta(v_p, y) = k_1 \leq k-1$
$\Rightarrow nh_{k_1}^{v_p} = u_q$
$\Rightarrow u_q \in S_{k_1}(v_p)$ (from the property1)
$\Rightarrow u_q \in B_k(x)$,
Also we have $B_k(x) \bigcap S_{k-1}(x) = \emptyset$
$\Rightarrow u_q \notin S_{k-1}(x)$,
It contradicts with what we already have, i.e $u_q \in S_{k-1}(x)$. Therefore it is proved by contradiction that there is only one cross-link between $S_{k-1}(x)$ and $B_k(x)$.
Property 3: For a given source $x \in S_{k-1}(x)$ and any destination $y \in B_k(x)$, consider the physical path is $x \rightsquigarrow y:\ x \rightarrow u_1^y \rightarrow u_2^y \ldots \rightarrow u_{m_1^y}^y \rightarrow v_1^y \ldots \rightarrow v_{m_2^y}^y \rightarrow y$. Then path segment $x \rightarrow u_1^y \rightarrow u_2^y \ldots \rightarrow u_{m_1}^y$ is same for all $y \in B_k(x)$ for a given x.
Proof: Let's say for a given source $x \in S_{k-1}(x)$ and destinations $y_1, y_2 \in B_k(x)$ paths are:
$x \rightsquigarrow y_1:\ x \rightarrow u_1^{y_1} \rightarrow u_2^{y_1} \ldots \rightarrow u_{m_1^{y_1}}^{y_1} \rightarrow v_1^{y_1} \ldots \rightarrow v_{m_2^{y_1}}^{y_1} \rightarrow y_1$
$x \rightsquigarrow y_2:\ x \rightarrow u_1^{y_2} \rightarrow u_2^{y_2} \ldots \rightarrow u_{m_1^{y_2}}^{y_2} \rightarrow v_1^{y_2} \ldots \rightarrow v_{m_2^{y_2}}^{y_2} \rightarrow y_2$
Here $u_i \in S_{k-1}(x)$ and $v_j \in B_k(x)$
Since $\delta(x, y_1) = \delta(x, y_2) = k$, and therefore nexthop at x to reach y_1 and y_2 is nh_k^x. And, $u_1^{y_1} = u_1^{y_2}$. Similarly we have $\delta(u, y_1) = \delta(u, y_2) = k$, therefore nexthop at any $u \in S_{k-1}(x)$ is same for both y_1 and y_2. Hence the path segment which lies in $S_{k-1}(x)$ is same for all the destinations in $B_k(x)$
i.e.$x \rightarrow u_1^{y_1} \rightarrow u_2^{y_1} \ldots \rightarrow u_{m_1^{y_1}}^{y_1} = x \rightarrow u_1^{y_2} \rightarrow u_2^{y_2} \ldots \rightarrow u_{m_1^{y_2}}^{y_2}$

A.2 Routing Algorithm Correctness Proof

Here we provide a proof to show that there are no loops if above strategy is used to build the routing tables. *For this proof, we assume that there are no "Bridges".* The following proof is based on the principal of mathematical induction. Also, we continue using the notations from the description of the strategy. Please refer to Fig. A.4 for the following proof.
Base case $(k = 1)$**:** In this case any node x in the network needs to learn the route to $B_1(x)$. We

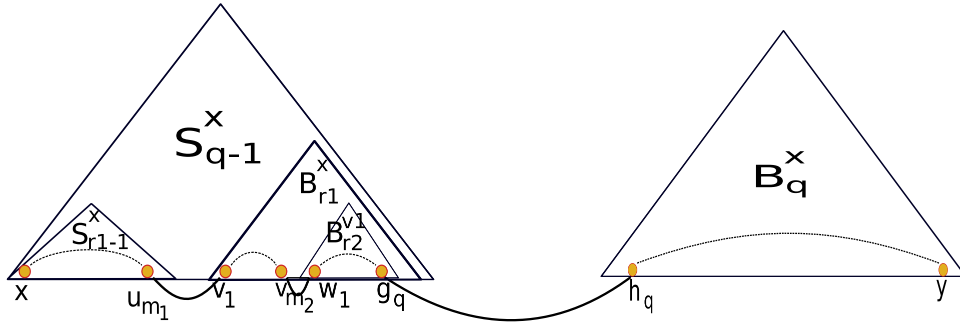


Figure A.4: Illustration of the no routing loops proof

know that $S_0(x)$ consists of x only. Therefore, from the assumption that there are no 'bridges' involved, x must be connected to the node in $B_1(x)$ directly, if there exist a node in this set. Also, when we perform the steps 1-4 we find that $nexthop$ is the destination node and gateway is the node x itself.

Let's assume that its true for $k = q - 1$ **here** $1 < q < l$ **:** It means that there are no loops for any node x to reach any other node $y \in S_{q-1}(x)$.

when $k = q$**, for** $1 < q \leq l$**:** In this step any node x in the network tries to reach its neighbors in the set $B_q(x)$ using the nodes in $S_{q-1}(x)$. First x contacts its rendezvous point $rdv_q(x)$ to learn all the possible 'gateways'. Then using the step 2 and 3 it finds the smallest r_1 such that $G^x_{r_1}(B_q(x)) \neq \emptyset$. It means there is no node in $S_{r_1-1}(x)$, which can act as a gateway to reach $B_q(x)$. Using step 4, for any node $u \in S_{r_1-1}(x)$ the nexthops to reach $B_q(x)$ is $nh^u_q = nh^u_{r_1}$. Now, consider $y \in B_q(x)$ and $z \in B_{r_1}(x)$, and paths:

$x \rightsquigarrow y : x \to u^y_1 \to \ldots \to u^y_{m^y_1} \to v_1 \ldots \to v_{m_2} \ldots \to y$ (here $u^y_i \in S_{r_1-1}(x)$ for $1 \leq i \leq m^y_1$ and $v_j \in B_{r_1}(x)$ for $1 \leq j \leq m_2$)

and $x \rightsquigarrow z : x \to u^z_1 \to \ldots \to u^z_{m^z_1} \ldots \to z$ (here $u^z_i \in S^x_{r_1-1}$ for $1 \leq i \leq m^z_1$)

Since for any $u \in S_{r_1-1}(x)$ the $nh^u_k = nh^u_{r_1}$.

Therefore $x \to u^y_1 \to \ldots \to u^y_{m^y_1}$ is same as $x \to u^z_1 \to \ldots \to u_{m^z_1}$ and $m^z_1 = m^y_1$. Also $S_{r_1-1}(x) \subseteq S_{q-1}(x)$, and S_{q-1} is loop free, therefore the path segment $x \to u^y_1 \to \ldots \to u^y_{m^y_1}$ is also loop free. Therefore the portion of the path that belongs to $S_{r_1-1}(x)$ is loop free. Now we need to show that remaining portion of the path $x \rightsquigarrow y$ i.e. $v_1 \rightsquigarrow y$ is loop free.

Let's say, with respect to node v_1 it's logically closest gateway to $B^{v_1}_k$ is at r_2 logical distance. Then $G^{v_1}_{r_2}(B_k(v_1)) \subseteq G^x_{r_1}(B_q(x)) \subseteq B_{r_1}(x)$ and $v_1 \in B_{r_1}(x)$. Therefore, $r_2 \leq r_1 - 1$.

Let's say $v_1 \rightsquigarrow dy = v_1 \to v_2 \ldots \to v_{m_1} \to w_1 \ldots w_{m_2} \to y$, where $v_j \in S_{r_2-1}(v_1)$ for

$1 \leq j \leq m_1$ and $w_i \in B_{r_2}(v_1)$, $1 \leq i \leq m_2$. Now consider any node $w \in B_{r_2}(v_1)$ then we can show that $v_1 \rightarrow v_2 \ldots \rightarrow v_{m_1}$ will be also present in $v_1 \rightsquigarrow w$ which is loop free. (Because $S_{r2}(v_1) \subseteq S_{q-1}(x)$, $nh^k_{v_i} = nh^w_{v_i}$ for $1 \leq i \leq m_1$, and $S_{q-1}(x)$ is loop free.) Hence portion of the path $v_1 \rightsquigarrow y$, which is $v_1 \rightsquigarrow w_1$ is also loop free. We also know that $v_1 \rightsquigarrow w_1$ consists of the nodes from $B^{v_1}_{r_1}$, the path $x \rightsquigarrow v_1$ consists of the nodes from $S_{r_1-1}(x) \cup \{v_1\}$, and $S_{r_1-1}(x) \cup B_{r_1}(v_1) = \emptyset$. For this reason the path $x \rightsquigarrow v_1 \rightsquigarrow w_1$ is also loop free. Furthermore, $G^{w_1}_{r_3}(B^{w_1}_q) \subseteq G^{v_1}_{r_2}(B^{v_1}_q)$ and $r_3 \leq r_2 - 1$. Now we have, $r_3 < r_2 < r_1$. We can extend it to have $0 < r_n < r_{n-1} \ldots < r_2 < r_1$, i.e. finally at some node say g^x_q (which is a gateway to reach $B_q(x)$) the "logical distance" to closest gateway will reduce to zero, which has one of it's physical neighbors say h^x_q in $B_q(x)$. Therefore the path from $x \rightsquigarrow g^x_q$ is loop free. Also, $h^x_q \rightsquigarrow y$ is loop free from the induction (because $B_q(x) = S^y_{q-1}$ is loop free.). From property 3 the cross-link from $S_{q-1}(x)$ to $B_q(x)$ on the path from $x \rightsquigarrow y$ is traversed only once. Therefore $x \rightsquigarrow g_q \rightarrow h_q \rightsquigarrow y$ is also loop free. Hence, there are no loops in on a path from $x \in S_{q-1}(x)$ to $y \in B_q(x)$ for any node x, and $S_{q-1}(x)$ and $B_q(x)$ are already loop free, so $S_q(x) = S_{q-1}(x) \cup B_q(x)$ is loop free. Therefore, by the principal of mathematical induction the proposed routing strategy is loop free.

In cases when Property 2 does not hold, we need bridges to connect the disconnected sub-tress. In such cases, we represent these disconnected sub-trees using different prefixes in the modified routing algorithm, which can be considered as two separate sub-tress. Therefore, the same proof can be extended for the loop-free routing in case of disconnected sub-tress.