Scalable and Efficient Self-configuring Networks

Changhoon Kim

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
By the Department of
Computer Science

Advisor: Professor Jennifer Rexford

September 2009

© Copyright by Changhoon Kim, 2009. All rights reserved.

Abstract

Managing today's data networks is highly expensive, difficult, and error-prone. At the center of this enormous difficulty lies configuration: a Sisyphean task of updating operational settings of numerous network devices and protocols. Much has been done to mask this configuration complexity intrinsic to conventional networks, but little effort has been made to redesign the networks themselves to make them easier to configure.

As part of a broad effort to *rearchitect networks with ease of configuration in mind*, this dissertation focuses on enabling *self-configuration* in *edge networks* – corporate or university-campus, data-center, or virtual private networks – which are rapidly growing and yet significantly under-explored. To ensure wide deployment, however, self-configuring networks must be scalable and efficient at the same time. To this end, we first identify three technical principles: *flat addressing* (enabling self-configuration), *traffic indirection* (enhancing scalability), and *usage-driven optimization* (improving efficiency). Then, to demonstrate the benefits of these principles, we design, implement, and deploy practical network architectures built upon the principles.

Our first architecture, SEATTLE, combines Ethernet's self-configuration capability with IP's scalability and efficiency. Its key contribution is a novel host-information resolution system that leverages the strong consistency of a network-layer routing protocol. The resulting architecture is suitable for enterprises and campuses to build a large-scale plug-and-play network. Extensive simulation and emulation tests, conducted by replaying real-world traffic traces on various real network topologies built with prototype SEATTLE switches, confirm that SEATTLE efficiently handles network failures and host mobility, while reducing control-plane overhead and state requirements by roughly two orders of magnitude compared with Ethernet bridging.

Our second solution, VL2, enables a plug-and-play network for a large cloud-computing

data center. The core objective of a data-center network is to maintain the utilization of the data-center servers at a uniformly high level, and doing so requires agility: the ability to assign any available server to any service. VL2 ensures agility by establishing reachability without addressing and routing configuration, and by furnishing huge server-to-server capacity. Meanwhile, VL2 offers all these benefits with only commodity IP and Ethernet functions without requiring any expensive high-performance component. We built a prototype VL2 network using commodity Ethernet switches interconnecting hundreds of servers. Tests with various real and synthetic traffic patterns confirm the VL2 design can achieve 93% of the optimal utilization in the worst case. Our prototype network will soon be expanded for a cloud-computing cluster composed of more than a thousand servers offering real-world service to customers.

Then we turn our focus to VPNs, networks that interconnect geographically distributed corporate sites through public carrier networks. VPNs today are built with an efficient self-configuring architecture, which allows customer sites to autonomously choose their own address blocks and communicate with one another through the shortest (i.e., most efficient) paths. This architecture, however, blindly replicates customers' routing information at every router in the VPN provider network and thus rapidly depletes routers' memory to cope with more customers, significantly impairing scalability. Our solution, based on traffic indirection, lowers routers' memory footprint by choosing a small number of hub routers in each VPN that maintain full routing information, and by allowing non-hub routers to keep a single default route to a hub. This solution can be implemented via a slight modification to the routers' configuration without requiring router hardware or software upgrade. Extensive evaluations using real traffic matrices, routing configurations, and VPN topologies demonstrate that Relaying reduces routing tables by up to 90%, while hiding the increase of latency and workload due to indirection.

Acknowledgments

I admit that working as a graduate student is not always the most fun thing to do. With my adviser, Jennifer Rexford, however, it can be. I have absolutely no doubt that she should be a role model of any adviser in our discipline. Jen has guided me with her incredible mind, amazing expertise on almost everything on networking, insatiable passion on research, love for teaching and mentoring, fantastic communication skills, fathomless patience, and – most importantly – incessant efforts to build a highly supportive environment for students. With all these, plus occasional feast with good wine, how can it not be fun to work with her?

Matthew Caesar has been another awesome colleague and mentor. His sharp intellect saved me from puddles many times, his diligence and passion showed me how to keep myself focused, and his wonderful taste in research problems served as a good model.

I've been extremely lucky to work with and learn from Albert Greenberg. Impromptu discussions with him were the sources of enlightenment. His invaluable advice and feedback throughout the most important period in my graduate career also helped me find right problems to work on, as well as exciting opportunities for further research.

I would like to thank my mentors and collaborators, especially Dave Maltz, Parveen Patel, and Sudipta Sengupta, during the extended internship at Microsoft Research. Dave has offered me numerous motivating questions, showed me how to efficiently and effectively lead a research group, and shared with me the fun of manipulating gadgets. Parveen's opinions and ideas, derived from his seasoned experience and knowledge, was one of the most valuable lessons. Sudipta has given me several awakening moments with his sharp observations and solutions. I also appreciate all the good discussions with Srikanth Kandula, James Hamilton, Parantap Lahiri, Navendu Jain, and John Dunagan.

I also met fantastic mentors at AT&T Labs: Alex Gerber, Shubho Sen, Dan Pei,

and Carsten Lund. Alex has continuously furnished me with challenging real problems, highly critical operational perspectives, along with extremely valuable data to play with. Shubho and Dan had lots of exciting discussions with me and offered me invigorating feedback on my research, as well as on my career. Carsten has helped me better understand the theoretic aspects of the Relaying work.

I'd love to thank Michael Freedman, Vivek Pai, and Margaret Martonosi for serving on my thesis committee and giving me great feedback.

At Princeton, especially in our research group, I've been always surrounded with truly bright people. From and with them, I received exceptionally good feedback and suggestions, learned various useful concepts and techniques, and shared lots of fun. Yi Wang has showed me how much people skills are important, and how to look around various non-research issues, including business and management. Haakon Ringberg and Sharon Goldberg have always helped me prepare for important talks with their sharp questions and comments. I also thank other members of the systems and networking group – Jiayue He, Elliott Karpilovsky, Rui Zhang-Shen, Wenjie Jiang, Eric Keller, Martin Suchara, Minlan Yu, Yaping Zhu, Wonho Kim, Sunghwan Ihm, Anirudh Badam, Wyatt Lloyd, Sid Sen, and Kyoungsoo Park – for fun conversations, pointed discussions, and for putting up with my pestering questions. I also thank Mark McCann for being my mountain-biking buddy in this boring – sorry, but that's true – central Jersey area. I've been also lucky to be with Yung Yi, my good old friend since university, who spared his good chunk of time on my research work and career options.

I'm grateful to the National Science Foundation, AT&T Labs, and Microsoft Research for their financial support. Specifically, this dissertation work was supported by NSF grant CNS-0519885.

I will miss Small World Coffee, which provided me with a good dosage of caffeine

and such a great place to work and chat.

I thank to my good old friend, Juli Scherer, for proofreading this dissertation.

My mother, late father, and brother have always been sources of comfort, refreshment, and encouragement. Without their support, I really couldn't have done it. Thank you, mom, dad, and Jonghun.

Despite too many evenings and holidays without their husband and dad, my wife and daughter have supported me with everything they could. Kyungle and Juni, I dedicate this dissertation to you.

Contents

Abstract iii

1 Introduction **1**

1.1 Re-designing Networks with Ease of Configuration in Mind 3

1.1.1 Towards self-configuring networks 3

1.1.2 Making self-configuring networks practical 4

1.2 Principles for Scalable and Efficient Self-configuring Networks 6

1.3 Architectures for Enterprises, Data Centers, and Virtual Private Networks 8

1.3.1 SEATTLE: A scalable Ethernet architecture for large enterprises . 10

1.3.2 VL2: A scalable and flexible data-center network 12

1.3.3 Relaying: A scalable routing architecture for VPNs 13

1.4 How to Read This Dissertation 15

2 SEATTLE: A Scalable Ethernet Architecture for Large Enterprises **18**

2.1 Motivation and Overview 19

2.1.1 Ethernet's simplicity + IP's scalability and efficiency
$\Rightarrow$ SEATTLE 21

2.1.2 Related work 23

2.2 Today's Enterprise and Access Networks 24

2.2.1 Ethernet bridging 25
2.2.2 Hybrid IP/Ethernet architecture 27
2.2.3 Virtual LANs 29
2.3 Network-Layer One-hop DHT 30
2.3.1 Scalable key-value management with a one-hop DHT 31
2.3.2 Responding to topology changes 34
2.3.3 Supporting hierarchy with a multi-level, one-hop DHT 36
2.4 Scaling Ethernet with a One-hop DHT 37
2.4.1 Host location resolution 37
2.4.2 Host address resolution 39
2.4.3 Handling host dynamics 40
2.5 Providing Ethernet-like Semantics 43
2.5.1 Bootstrapping hosts 44
2.5.2 Scalable and flexible VLANs 45
2.5.3 Methodology 47
2.6 Simulations 49
2.6.1 Control-plane scalability 49
2.6.2 Sensitivity to network dynamics 53
2.7 Implementation 55
2.7.1 Prototype design 55
2.7.2 Experimental results 57
2.8 Summary 61

3 VL2: Scalable and Flexible Data-Center Networks 63

3.1 Motivation and Overview 65

3.1.1 Principles and contributions of VL2 67
3.2 Background 69
3.3 Measurements and Implications 72
3.3.1 Data center traffic analysis 72
3.3.2 Flow distribution analysis 73
3.3.3 Traffic matrix analysis 74
3.3.4 Failure characteristics 77
3.4 Virtual Layer Two Networking 78
3.4.1 Scalable oversubscription-free topology 80
3.4.2 VL2 routing 82
3.4.3 Maintaining host information using VL2 directory system 88
3.5 Evaluation 91
3.5.1 VL2 Uniform high capacity 92
3.5.2 Performance isolation 94
3.5.3 Convergence after link failures 99
3.5.4 Directory-system performance 100
3.6 Discussion 104
3.7 Related Work 108
3.8 Summary 109

4 Relaying: A Scalable Routing Architecture for Virtual Private Networks 111
4.1 Motivation and Overview 113
4.1.1 Relaying: Don't keep it if you don't need it 115
4.2 Background 118
4.2.1 How MPLS VPN works 118

4.2.2 Desirable properties of a solution 120
4.3 Understanding VPNs 123
4.3.1 Data sources 123
4.3.2 Properties enabling memory saving 124
4.4 Overview of Relaying 126
4.4.1 Relaying through hubs 126
4.4.2 Hub selection vs. hub assignment 127
4.5 Baseline Performance of Relaying 129
4.5.1 Selecting heavy sources or sinks as hubs 130
4.5.2 Performance of the hub selection 131
4.6 Latency-constrained Relaying 133
4.6.1 LCR problem formulation 134
4.6.2 Algorithm to solve LCR 137
4.6.3 Solutions with usage-based matrices 138
4.6.4 Solutions with full-mesh matrices 142
4.6.5 Robustness 143
4.7 Latency-constrained, Volume-sensitive Relaying 144
4.7.1 LCVSR problem formulation 145
4.7.2 Algorithm to solve LCVSR 146
4.7.3 Solutions with usage-based matrices 147
4.7.4 Solutions with full-mesh matrices 149
4.7.5 Robustness 150
4.8 Implementation and Deployment 151
4.8.1 Implementing Relaying with BGP 152
4.8.2 Managing Relaying-enabled VPNs 154

4.9 Related Work 156
4.10 Summary 157

5 Conclusion 160

5.1 Summary of Contributions 160
5.1.1 Scalable and efficient self-configuring networks can be made practical 161
5.1.2 Principles and applications 163
5.2 Future Work and Open Issues 165
5.3 Concluding Remarks 168

List of Figures

2.1 Keys are consistently hashed onto resolver switches (s_i). 33

2.2 Hierarchical SEATTLE hashes keys onto *regions*. 35

2.3 Packet forwarding and lookup in SEATTLE 38

2.4 (a) Effect of cache timeout in *AP-large* with $50K$ hosts, (b) Table size increase in *DC*, and (c) Control overhead in *AP-large*. Error bars in these figures show confidence intervals for each data point. A sufficient number of simulation runs reduced these intervals. 50

2.5 (a) Stretch across different cache sizes in *AP-large* with $10K$ hosts, (b) Path stability, and (c) Effect of switch failures in *DC*. 54

2.6 Effect of host mobility in *Campus*. 55

2.7 Implementation architecture. 57

2.8 Effect of network dynamics – (a) table size, (b) control overhead, and (c) failover performance. 59

3.1 The conventional network architecture for data centers 69

3.2 Mice are numerous; 99% of flows are smaller than 100 MB. However, more than 90% of bytes are in flows larger than 100 MB. 74

3.3 Number of concurrent connections has two modes: (1) 10 flows per node more than 50% of the time and (2) 80 flows per node for at least 5% of the time. 75
3.4 Lack of short-term predictability: The cluster to which a traffic matrix belongs, i.e., the type of traffic mix in the TM, changes quickly and randomly. 76
3.5 Example Clos network between Aggregation and Intermediate switches provides a broad and richly connected backbone well-suited for VLB. Connectivity to the Internet is provided by Core Routers (CR). 81
3.6 VLB in an example VL2 network. Sender S sends packets to destination D via a randomly-chosen intermediate switch using IP-in-IP encapsulation. AAs are from $20/8$ and LAs are from $10/8$. H(`ft`) denotes a hash of the five tuple. 85
3.7 VL2 Directory System Architecture 89
3.8 VL2 testbed comprising 80 servers and 10 switches. 92
3.9 Aggregate goodput during a 2.7TB shuffle among 75 servers. 93
3.10 Aggregate goodput of two services with servers intermingled on the TORs. Service one's goodput is unaffected as service two ramps traffic up and down. 96
3.11 Aggregate goodput of service one as service two creates bursts containing successively more short TCP connections. 97
3.12 Fairness measures how evenly flows are split to intermediate switches from aggregation switches. Average utilization is for links between Aggregation and Intermediate switches. 98

3.13 Aggregate goodput as all links to switches Intermediate1 and Intermediate2 are unplugged in succession and then reconnected in succession. Approximate times of link manipulation marked with vertical lines. Network re-converges in < 1s after each failure and demonstrates graceful degradation. 99
3.14 The directory system provides high throughput and fast response time for lookups and updates 101
3.15 Three layouts for VL2: (a) Conventional DC floor layout, (b) Container-based layout with intermediate switches part of DC infrastructure, and (c) Fully "containerized" layout. (External connectivity, servers racks, and complete wiring not shown.) 107

4.1 (a) MPLS VPN service with three PEs; two customer VPNs (X, Y) exist, (b) Direct reachability, (c) Reachability under Relaying. 119
4.2 (a) CDFs of the proportion of active prefixes in a VRF, (b) CDFs of the distance to the i-th percentile closest VRF 125
4.3 (a) Gain and cost (de facto asgn.), (b) Sum of the products of volume and additional distance, (c) CDF of additional distances ($\alpha = 0.1$) 128
4.4 A sample serve-use relationship 135
4.5 LCR performance with usage-based C (a) CDF of additional distances, (b) CDF of additional distances (zoomed in), (c) Gain and cost 136
4.6 (a) LCR performance with C^{full}, (b) Robustness results (costs), (c) Robustness results (CDF of additional distances with C) 139
4.7 A sample serve-use relationship with penalties 145

4.8 LCVSR performance (a) Gain and cost (with C), (b) CDF of additional distances (with C) (c) Gain and cost (with C^{full}) 148
4.9 Robustness results during test weeks 153
4.10 BGP configuration for Relaying 155

List of Tables

1.1 Summary of network architectures this dissertation proposes 10

2.1 Per-packet processing time in micro-sec. 58

4.1 Proportions (in percentage) of hubs that remain as hubs across two windows (averaged across all windows and θ) 155

5.1 Specific aspects of self-configuration, scalability, and efficiency in the proposed architectures 161

5.2 Key principles and the varying applications of the principles 164

Chapter 1

Introduction

Networked services, such as e-mail, Web, and on-line businesses, have become inseparable from our everyday lives. Ensuring the performance and reliability of these critical services requires managing the communication networks upon which the services are running. This comprehensive activity, called *network management*, encompasses various operational tasks, including provisioning, running, monitoring, controlling, and troubleshooting networking devices and networked software systems. While the high-level goals and approaches of these management tasks differ from one another, carrying out these tasks in today's networks always involves *configuration* – a process of determining specific operational settings of networking devices and protocols, and applying them to appropriate points in a network.

Although it does not look particularly daunting at a glance, *configuration is indeed the Sisyphean task of network management*. Configuring a network introduces enormous costs [1, 2], correctly configuring a network is notoriously difficult [3, 4, 5, 6], and yet, configuration settings must be frequently and repeatedly updated just to maintain the status quo, rather than build up value-added features for a network [7].

The reasons are many. Most importantly, configuration is a highly device-centric ad-hoc task, retrofitted to diverse underlying devices and protocols. This makes it extremely difficult to come up with an elegant unifying solution for configuration and thus forces administrators to rely on domain-specific knowledge about various protocols, types of devices, and proprietary configuration-support solutions [8, 9]. The process of configuration can also be highly inefficient and labor-intensive because some configuration tasks (e.g., bootstrapping routers) are inherently hard to automate or perform from a remote location. Worse yet, all the distributed settings over a large number of heterogeneous devices have to be frequently updated for various reasons, such as maintenance, failure recovery, natural growth of the organization, and human-resource re-assignment. This forces administrators to keep repeating the onerous configuration tasks. Finally, despite the distributed and highly-dynamic nature of the task, all configuration settings have to be semantically consistent in order to achieve a network-level goal. This requirement steeply increases the configuration complexity and makes the network highly susceptible to disruptions caused by configuration errors [3].

Unfortunately, the prospects are bleak. Network sizes are increasing very fast today, inflating the configuration workload rapidly. Networks in large corporate or university campuses easily contain tens of thousands of end hosts, large data-center networks are built with hundreds of thousands of servers interconnected via tens of thousands of routers and switches, and metro-area provider networks are targeting for more than a million subscribers. Worse yet, networks themselves are becoming highly dynamic, due to technical advances such as host mobility, virtualization, and cloud computing [10]. Since this trend allows networks' sizes and topologies to be frequently adjusted for varying input workloads, the overall configuration overhead also increases substantially. Meanwhile, networks are also being increasingly deployed in non-information-technology industries

and developing regions, where network management is considered only a supporting duty, and thus technical and financial support for network management tends to be significantly limited. All these observations lead us to conclude that *ease of configuration is paramount*.

1.1 Re-designing Networks with Ease of Configuration in Mind

This dissertation focuses on addressing these huge difficulties of network configuration. Although we are not the first to look at this broad problem, our goal, approach, and solutions are substantially different from previous work.

1.1.1 Towards self-configuring networks

The past decade has seen the proliferation of network-management solutions that aim to facilitate configuration. Those conventional solutions range over approaches that try to automate common configuration patterns [11, 12, 13], offer centralized views that help administrators realize network-wide objectives [14, 15], or understand existing configuration settings and diagnose syntactic or semantic errors present in them [6, 16]. While succeeding in offering some benefits, those *management-layer* approaches (i.e., approaches running upon existing un-modified networks) are less likely to yield solutions with momentous improvement because they only mask the configuration complexity intrinsic to the underlying networks. In fact, there has been relatively little effort on *making the underlying networks easy to configure in the first place*, such as eliminating configuration altogether.

This dissertation tackles the latter issue by *re-designing the underlying network architectures themselves while focusing on ease of configuration*. Specifically, to cope with the rapidly growing networks with low configuration overhead and cost, and to deal with frequent network and host churns without configuration errors, we put forward *self-configuration* – enabling networks to maintain basic reachability and performance in a configuration-free (also known as plug-and-play) fashion – as the first-order technical goal.

1.1.2 Making self-configuring networks practical

Attempting to realize a self-configuring network architecture useful for a wide variety of environments is a highly ambitious task, especially if attempted in a single dissertation. Hence, as part of on-going research, this dissertation focuses on two key issues towards this broad goal.

Focus on edge networks

First, we focus on *edge networks* – networks that typically have little to no interdependence with other networks because they do not offer networking services for other networks. Corporate-campus networks, university-campus networks, data-center networks, or virtual private networks are all good examples. There are several reasons for specifically looking at this kind of network. Most of all, network management rarely receives sufficient attention in edge networks and, therefore, has to be done correctly with only limited technical and financial support. At the same time, due to little interdependence with other networks, edge networks can easily (and are even willing to) adopt new network architectures without consideration of interoperability with other networks. Additionally, edge networks are growing in size very quickly and yet are significantly under-

explored by the research community.

Emphasis on workable solutions

Second, we emphasize *workable solutions*. The huge workloads and difficulties of network configuration are afflicting administrators in operational networks today. As such, real-world deployment (or deployability at the least) must be considered the key measure of success. Therefore, even though we propose new network "architectures", we always consider prototyping and testing with real-world traffic as integral parts of design and evaluation. By the same token, we sometimes deliberately move away from architecturally pure or clean-slate approaches, and we instead embrace substituting backwards-compatible techniques, if doing so ensures more practicality.

Meanwhile, this emphasis on workable solutions also leads us to a more specific goal of this dissertation. That is because self-configuration alone is not sufficiently practical for real-world deployment. For example, Ethernet bridging supports plug-and-play networking by allowing hosts to communicate with each other using their own unique and permanent identifiers – MAC (Media Access Control) addresses – regardless of their locations in a network, and by letting a network self-learn hosts' address and location information. This mechanism, though effective in eliminating host-address configuration (i.e., assigning location-dependent identifiers to hosts) and routing configuration (i.e., informing routers/switches of hosts' identifiers and locations), does not permit the network to grow beyond a small-scale deployment. This limitation arises from the fact that the self-learning capability relies on frequent network-wide dissemination of individual hosts' information, known as broadcasting and flooding. Moreover, since broadcasting over a physically-rich topology can lead to forwarding loops, an Ethernet network must forward traffic only through a single spanning tree even when several alternative paths are available. This approach significantly lowers the scalability and efficiency (in terms

of path lengths, link utilization, etc.) of an Ethernet network as its size grows.

On the other hand, the Internet Protocol (IP) ensures scalability by hierarchically aggregating host-address and routing information, and it achieves efficiency using the shortest paths between routers. This approach makes it impossible to attain self-configuration, however, because the hierarchy in a network must be specifically structured and also frequently updated (for maintenance, supporting host mobility, etc.) by administrators.

In summary, networks today need self-configuration without sacrificing scalability and efficiency. As such, this dissertation specifically attempts to *design, build, and deploy architectural solutions that enable scalable and efficient self-configuring edge networks*.

1.2 Principles for Scalable and Efficient Self-configuring Networks

This dissertation makes two sets of contributions: *principles* and *applications*. The first set of contributions includes generic technical principles useful for designing scalable and efficient self-configuring networks. Additionally, the second set includes specific edge-network architectures that are built upon those generic principles. We introduce the principles first in this section and defer the discussion on applications to the next section.

While working on different edge networks with varying purposes, requirements, capabilities, and constraints, we were able to draw three key principles below that are highly useful for various networks.

- **Self-configuration via flat addressing** : We allow hosts to autonomously choose their own permanent location-independent identifiers, and the network to self-learn and replicate hosts' identifiers and locations, as opposed to having administra-

tors specifically assign hierarchical addresses (i.e., location-dependent temporary identifiers) at hosts and routing information (i.e., hierarchically-aggregated host-address blocks) at network devices. This enables self-configuration and thus allows administrators to forgo address and routing configuration to ensure reachability. This approach, however, can significantly increase the overhead for storing and disseminating host information because such information is not aggregatable. The following two principles offer solutions for this problem.

- **Scalability via traffic indirection** : Network devices deliver traffic indirectly through a small number of intermediate devices – rather than through shortest paths – that are chosen either randomly or systematically. This mechanism seems counter-productive at first glance, as using indirect paths increases, rather than decreases, the total amount of traffic carried through a network and the end-to-end latency as well. While these costs do exist, in edge networks, the benefits of traffic indirection often significantly exceed the costs for two main reasons. First, on the cost side, the indirection penalty is often negligible (as detour paths are only slightly longer in edge networks) and easily avoidable (as one can deliberately choose only highly-popular traffic sources and sinks as intermediaries, or forward only a small fraction of traffic through the intermediaries by employing caching – our third technical principle explained below). Second, in terms of benefits, traffic indirection enables a network to grow very large without requiring a concomitantly large amount of resources and thus enhances scalability. Specifically, traffic indirection exempts non-intermediate devices from the overhead of storing and disseminating the information for all hosts because those devices need to maintain just the information about other network devices (or just the intermediate devices), but not

about all the hosts. Traffic indirection is also useful for spreading traffic over a large number of paths and thus enables a network to cope with huge and skewed or drastically-varying traffic patterns without requiring excessive link capacities that are mostly squandered.

- **Efficiency via usage-driven optimization** : Network devices populate routing and host information only when and where it is needed by actual traffic and cache only frequently-used information, rather than blindly disseminating the information across the entire network. This approach improves efficiency by allowing network devices to maintain only popular host and routing information and yet make it possible to offer direct communication paths for popular communication patterns. Since in most edge networks only a small fraction of hosts are highly popular, caching only those popular hosts' information often substantially improves a network's efficiency in terms of the usage of expensive network resources (e.g., fast-access memory to store host information).

1.3 Architectures for Enterprises, Data Centers, and Virtual Private Networks

By building upon the principles introduced in the previous section, this dissertation proposes three architectures which are customized for different types of edge networks – enterprise networks, data-center networks, and virtual private networks (VPNs). An enterprise network, as we define it in this dissertation, is built and administered by a single organization and offers networking service only to the hosts in that organization. Often this type of network covers a limited geographical area, such as a corporate campus, a

university campus, a few residential/commercial districts, or a government organization. Nonetheless, it is technically possible to construct an enterprise network that spans multiple geographically distributed campuses (sites) by interconnecting them via VPNs – networks that offer dedicated and protected wide-area connectivity among the distributed sites through large public carrier networks (i.e., the Internet). A data-center network is a specialized enterprise network that interconnects a large number (typically tens to hundreds of thousands) of servers located in a data center. Compared to ordinary enterprise networks, data-center networks are more regular – in terms of topology and components – and usually aim to ensure a higher degree of performance and reliability.

The reasons we explore specifically these three types of networks are many and various. First, by offering solutions for different types of networks, we intend to show the generality of our key principles upon which the solutions are built. Second, while there are certainly other types of networks that can adopt these principles, we believe that these three types of networks represent the most important kinds of edge networks existing today. Third, the problem of ensuring scalability, efficiency, and self-configuration for non-edge networks, most importantly for service-provider backbones or the Internet, has been actively investigated by various previous and ongoing research work [17, 18, 19, 20, 21].

We briefly introduce each of these network architectures in this section and defer thorough explanation of the architectures to Chapters 2 to 4. For a quick overview, we summarize key issues and design features of each architecture in Table 1.1.

Table 1.1: Summary of network architectures this dissertation proposes

	SEATTLE (Chapter 2)	**VL2** (Chapter 3)	**Relaying** (Chapter 4)
Target	Enterprise networks	Data-center networks	VPN-provider networks
Motivation	Self-configuration is paramount in enterprises, because support for network management is often highly limited in those networks	Self-configuration improves scalability and agility (i.e. capability of assigning any resources to any services) of a cloud-computing data center	VPN providers are in dire need of memory capacity to cope with fast-growing numbers and sizes of customer VPNs
Problems with existing architecture	Self-configuring networks do not scale; scalable and efficient networks bear huge configuration overhead	Poor agility and limited server-to-server capacity lower servers' and links' utilization in a data center	Replicating every customer site's information at every customer-facing router impairs scalability
Goal	Combine IP's scalability (low overhead to maintain hosts' state) and efficiency (shortest-path forwarding) with Ethernet's self-configuration	Offer the image of a huge layer-2 switch furnishing full non-blocking capacity among servers and support for agility	Reduce routers' memory footprint required for storing customers' information without harming end-to-end performance
Key constraint	Huge heterogeneity of end-host environments	Limited programmability at switches and routers	Need for immediate deployment and transparency to customers
Approach	Clean slate on network	Clean slate on hosts	Backwards-compatible
Deployment	Several independent prototypes are built by different research groups	Are expected to be deployed for a large public cloud-computing infrastructure	Passed pre-deployment tests and are expected to be deployed for large customer VPNs

1.3.1 SEATTLE: A scalable Ethernet architecture for large enterprises

For most enterprises (especially those in non-IT sectors), network management in essence is a supporting duty, rather than a core, value-generating duty. Administrators of those networks, therefore, suffer most from a huge configuration workload because techni-

cal and financial support for network management is often highly limited. Our first architecture, SEATTLE, precisely addresses this problem. SEATTLE allows enterprises to build a large-scale plug-and-play network that ensures reachability entirely by itself, without requiring any addressing and routing configuration by administrators. Meanwhile SEATTLE employs shortest-path forwarding and thus ensures traffic-forwarding efficiency equivalent to that of existing IP networks.

To ensure these features, SEATTLE proposes *i*) a highly-scalable host-information resolution system leveraging the consistency offered by a network-layer routing protocol, *ii*) a traffic-driven host-information resolution and caching mechanism taking advantage of strong traffic locality in enterprise networks, and *iii*) a scalable and prompt cache-update protocol ensuring eventual consistency of host information in a highly dynamic network. Despite these novel mechanisms, SEATTLE still remains compatible with existing applications and protocols – Address Resolution Protocol (ARP), Dynamic Host Configuration Protocol (DHCP), link-local broadcast, Virtual LAN (VLAN), etc. – running at end hosts.

SEATTLE is available as several independent prototypes implemented by different research groups (for example, the system by De Carli, et. al. [22]), including our own prototype built with open-source routing platforms [23]. When instantiated as a Linux kernel-thread module, this prototype SEATTLE switch can handle the high-speed links (gigabits per second) commonly used in corporate and university campuses today. To attain this level of practicality, we deliberately adjusted our design to the unique requirements, technical constraints, and available options in enterprise networks. First, to ensure fast prototyping and easier deployment, SEATTLE re-uses many existing network-layer functions (e.g., link-state routing, encapsulation/decapsulation) simply by re-factoring and re-arranging them in a novel fashion. Second, we deliberately choose to modify only

network devices, offering exactly the same simple semantics as Ethernet. This is because enterprise networks often have significant host-level heterogeneity; each department in a university has numerous custom applications that work only on certain versions of host operating systems, and corporations often use diverse kinds of devices and operating systems for various historical, business, and management purposes. By taking a purely network-based approach, SEATTLE substantially lowers the overall design and implementation complexity of the system and enables rapid and inexpensive development and deployment as well.

1.3.2 VL2: A scalable and flexible data-center network

Many features of SEATTLE are also useful for other types of networks, most importantly for large cloud-computing service providers whose ultimate goal is administering their data centers in a configuration-free fashion. We address this crucial issue by proposing a highly scalable network architecture for cloud-computing data centers, called VL2. In addition to eliminating configuration for addressing and routing, this architecture also allows administrators to forgo sophisticated and frequent configuration for traffic engineering – a task of controlling the amount of traffic flowing through each link to avoid congestion – by delivering traffic through a large number of randomly-chosen indirect paths. Additionally, this approach can substantially reduce the cost of building a data-center network because spreading traffic over multiple paths enables a network to cope with highly skewed or drastically-varying traffic workloads with only a huge aggregate capacity (i.e., with a large number of inexpensive commodity components), rather than with huge individual links (i.e., expensive high-performance components). Together, the plug-and-play capability and the low-cost high-bandwidth server-to-server capacity enable administrators to assign any available servers to any services sharing the cloud-

computing infrastructure, and thus can substantially increase the statistical multiplexing gain (i.e., utilization) of a data center.

Despite these novel features, VL2 requires only network-layer technologies available today and thus is immediately deployable. In fact, we have implemented a VL2 prototype network serving an 80-server cluster using only low-cost commodity Ethernet switches. Our pre-deployment tests in this test setting demonstrate that the prototype network can offer a uniformly high capacity among servers (93% of optimal), and the way the prototype network achieves this performance is directly applicable to up to hundreds of thousands of servers. VL2 is also expected to be rolled out for real-world deployment for a large cloud-service provider.

To attain this level of practicality, VL2 leverages add-on functions in end hosts' operating systems without requiring any novel functions in a network, server hardware, or application software. Our observations on common data-center designs and operational settings justify this design decision. Unlike general enterprise networks, host operating systems in data centers are already highly customized for data-center-specific needs (e.g., virtualization), and adding a few more novel functionalities for VL2 in the host operating systems is relatively straight-forward and convenient. Additionally, end-hosts' environments are highly homogeneous in a data center and offer uniformly rich programmability at a relatively low cost.

1.3.3 Relaying: A scalable routing architecture for VPNs

Finally, we turn our focus to VPN, a fast-growing networking service used by corporate customers to interconnect geographically distributed sites. The conventional network architecture to support VPNs over a service-provider network allows provider routers to receive customer sites' addresses (i.e., route prefixes) from customer routers. Provider

routers then exchange this information with one another. This approach ensures self-configuration, as the customer sites can autonomously assign and announce their own address blocks, and the provider routers self-learn and blindly disseminate this varying information across the entire provider network without any specific changes to their configuration settings. The problem, however, is that this mechanism unnecessarily replicates customer routing information at a huge number of provider routers and thus significantly impairs the provider's scalability to cope with the rapidly growing sizes and number of customer VPNs; more specifically, routers consume too much high-speed memory to store the routing information about all sites.

We solve this problem via a novel routing architecture called Relaying. The Relaying architecture substantially lowers routers' memory footprint by allowing routers connected to highly-popular customer sites (namely hub routers) to maintain complete routing information about all customer sites, while letting routers connected to less-popular sites (namely spoke routers) maintain only a single routing-table entry pointing to a hub router. Thus any hub router can reach any customer site directly through the shortest paths, while spoke routers can reach other customer sites only through a hub router. Since the traffic distribution of customer sites is highly skewed (i.e., a small number of highly popular sites generate and receive most traffic in a VPN), choosing only those small number of routers connected to popular sites as hubs can significantly reduce the amount of routing information stored in expensive memory and yet ensure fairly good end-to-end performance for most traffic as well. To help administrators make informed decisions, we also design a number of optimization algorithms. Given traffic patterns among customer sites and the locations of customer-facing routers, these algorithms can choose the smallest number of hub routers, such that the increase of end-to-end latency due to indirection is bounded above by a certain parameter. We also develop a decision-support tool imple-

menting these algorithms.

The dire scalability problem that VPN providers experience today demands a solution that is immediately deployable; implementing Relaying should rely only on router mechanisms readily available today. Meanwhile, service-level agreements (SLAs) between a provider and customers require that a solution is transparent to customers; the impact of Relaying must be unnoticeable in terms of perceived end-to-end performance. Satisfying these requirements, our Relaying design can be implemented with only slight modification to conventional router-configuration settings for VPNs, without requiring any new router hardware or software. Relaying has passed pre-deployment laboratory tests (including experiments to examine line-rate packet-relaying performance at a hub router) by a large tier-1 carrier in the U.S. and is expected to be deployed for large corporate VPNs served by the carrier.

1.4 How to Read This Dissertation

In the remainder of this dissertation, we describe the detailed design, implementation, and evaluation results of the three network architectures briefly introduced in the previous section. We begin with SEATTLE in Chapter 2 as it provides readers with background knowledge on typical edge networks. It also describes how we use the key technical principles (introduced in Section 1.2) to enable self-configuration in a large yet familiar network, without unnecessarily delving into advanced networking issues covered in the subsequent chapters. While Chapter 2 focuses mainly on ensuring reachability and control-plane scalability in a plug-and-play fashion, we present VL2 in Chapter 3 and thus also deal with issues of avoiding congestion and enabling data-plane scalability in a plug-and-play fashion. Finally, we describe Relaying in Chapter 4. In addition to introducing

a highly-scalable routing architecture that preserves the self-configuring semantics of the VPN service, this work also studies the question of how to optimally deploy the new architecture in an operational network.

While solving the same high-level problem via the same set of technical principles, these architectures also reflect on various strategic approaches chosen for different types of networks under investigation. Understanding the broad context of these different approaches will help readers with different backgrounds and interests to efficiently navigate through the chapters of this dissertation.

The SEATTLE and the Relaying work aim to improve the scalability of *existing* self-configuring networks (i.e., Ethernet and VPNs respectively). On the other hand, the VL2 work intends to present a novel architecture for an *emerging* type of network: huge cloud-computing data-center networks.

This difference leads to the varying backwards-compatibility decisions made in this dissertation. SEATTLE ensures complete backwards-compatibility with existing hosts, as they are designed for “existing” networks. By the same token, Relaying works without requiring any modification to customer sites (address reassignment, software or hardware upgrade on customer routers, etc.). On the other hand, VL2 deliberately employs programmability available at end hosts – mostly because cloud-computing data-center networks are being built today – and thus has significant freedom to incorporate new end-host primitives. Note, however, that VL2 does not require existing applications (running on data centers) to be modified, simplifying VL2’s deployment.

While both SEATTLE and Relaying rely on new mechanisms (i.e., routing and forwarding) running inside the network, we decides to choose a clean-slate approach in SEATTLE (i.e., proposing a new type of network device different both from IP routers and Ethernet bridges), whereas in Relaying we choose a design that can be realized with

existing router functions only. These decisions are related to how urgent the challenges each type of network faces are. In enterprises, though, a scalable Ethernet is highly desirable, administrators can get by with a short-term patch solution – for example, interconnecting small Ethernet networks with IP routers – while waiting for SEATTLE or any other fundamental solutions to become available. Unfortunately, VPN providers are left with no interim solution but extremely inefficient over-provisioning. As such, they need a solution that works immediately on existing routers, without requiring any hardware or software upgrade. This also affects our research direction in the Relaying work, as VPN providers are as much interested in management-support algorithms and tools needed to deploy and run Relaying as the Relaying architecture itself.

Chapter 2

SEATTLE: A Scalable Ethernet Architecture for Large Enterprises

IP networks today require massive effort to configure and manage. Ethernet is vastly simpler to manage, but does not scale beyond small local area networks. This chapter describes an alternative network architecture called SEATTLE that achieves the best of both worlds: the scalability of IP combined with the simplicity of Ethernet. SEATTLE provides plug-and-play functionality via flat addressing, while ensuring scalability and efficiency through shortest-path routing and hash-based resolution of host information. In contrast to previous work on identity-based routing, SEATTLE ensures path predictability and stability and simplifies network management.

We begin by motivating this work in Section 2.1. Before plunging into the detailed designs, we summarize how conventional enterprise networks are built in Section 2.2. Then we describe our main contributions in Sections 2.3 where we introduce a highly-scalable and flexible host-information management system. Subsequently in Section 2.4, we show how we can use this system to build a scalable network architecture that ensures

the same semantics as Ethernet bridging. Next in Section 2.5, we enhance existing Ethernet mechanisms to make our design backwards-compatible with conventional Ethernet. We then evaluate our protocol using simulations in Section 2.6 and an implementation in Section 2.7. Our results show that SEATTLE scales to networks containing two orders of magnitude more hosts than a traditional Ethernet network. Additionally, it also demonstrates that SEATTLE efficiently handles network failures and host mobility while reducing control overhead and state requirements by roughly two orders of magnitude compared with Ethernet bridging.

2.1 Motivation and Overview

Ethernet stands as one of the most widely used networking technologies today. Due to its self-configuring capability, wide availability, and low cost, many enterprise and access provider networks utilize Ethernet as an elementary building block. Each host in an Ethernet is assigned a persistent MAC address, and Ethernet bridges automatically learn host addresses and locations. These "plug-and-play" semantics simplify many aspects of network configuration. Flat addressing simplifies the handling of topology changes and host mobility without requiring administrators to reassign addresses.

However, Ethernet is facing revolutionary challenges. Today's layer-2 networks are being built on an unprecedented scale and with highly demanding requirements in terms of efficiency and availability. Large data centers are being built, comprising hundreds of thousands of computers within a single facility [24], and maintained by hundreds of network operators. To reduce energy costs, these data centers employ virtual machine migration and adapt to varying workloads, placing additional requirements on agility (e.g., host mobility and fast topology changes). Additionally, large metro Ethernet de-

ployments contain over a million hosts and tens of thousands of bridges [25]. Ethernet is also being increasingly deployed in highly dynamic environments, such as backhaul for wireless campus networks, and transport for developing regions [26].

While an Ethernet-based solution becomes all the more important in these environments because it ensures service continuity and simplifies configuration, conventional Ethernet has some critical limitations. First, Ethernet bridging relies on network-wide *flooding* to locate end hosts. This results in large state requirements and control message overhead that grows with the size of the network. Second, Ethernet forces paths to comprise a *spanning tree*. Spanning trees perform well for small networks which often do not have many redundant paths anyway, but introduce substantial inefficiencies on larger networks that have more demanding requirements for low latency, high availability, and traffic engineering. Finally, popular bootstrapping protocols, such as Address Resolution Protocol (ARP) and Dynamic Host Configuration Protocol (DHCP), rely on *broadcasting*. This not only consumes excessive resources, but also introduces security vulnerabilities and privacy concerns.

Network administrators sidestep Ethernet's inefficiencies today by interconnecting small Ethernet LANs using routers running the Internet Protocol (IP). IP routing ensures efficient and flexible use of networking resources via shortest-path routing. It also has control overhead and forwarding-table sizes that are proportional to the number of subnets (i.e., prefixes), rather than the number of hosts. However, introducing IP routing breaks many of the desirable properties of Ethernet. For example, network administrators must now subdivide their address space to assign IP prefixes across the topology, and update these configurations when the network design changes. Subnetting leads to wasted address space, and laborious configuration tasks. Although DHCP automates host address configuration, maintaining consistency between DHCP servers and routers still

remains challenging. Moreover, since IP addresses are not persistent identifiers, ensuring service continuity across location changes (e.g., due to virtual machine migration or physical mobility) becomes more challenging. Additionally, access-control policies must be specified based on the host's current position, and updated when the host moves.

Alternatively, operators may use Virtual LANs (VLANs) to build IP subnets independently of host location. While the overhead of address configuration and IP routing may be reduced by provisioning VLANs over a large number of, if not all, bridges, doing so reduces benefits of broadcast scoping, and worsens data-plane efficiency due to larger spanning trees. Efficiently assigning VLANs over bridges and links must also consider hosts' communication and mobility patterns, and hence is hard to automate. Moreover, since hosts in different VLANs still require IP to communicate with one another, this architecture still inherits many of the challenges of IP mentioned above.

2.1.1 Ethernet's simplicity + IP's scalability and efficiency ⇒ SEATTLE

In this chapter, we address the following question: *Is it possible to build a protocol that maintains the same configuration-free properties as Ethernet bridging, yet scales to large networks?* To answer, we present a Scalable Ethernet Architecture for Large Enterprises (SEATTLE). Specifically, SEATTLE offers the following features:

A one-hop, network-layer DHT: SEATTLE forwards packets based on end-host MAC addresses. However, SEATTLE does *not* require each switch to maintain state for every host, *nor* does it require network-wide floods to disseminate host locations. Instead, SEATTLE uses the global switch-level view provided by a link-state routing protocol to form a one-hop DHT [27], which stores the *location* of each host. We use this network-

layer DHT to build a flexible directory service which also performs *address resolution* (e.g., storing the MAC address associated with an IP address), and more flexible service discovery (e.g., storing the least loaded DNS server or printer within the domain). In addition, to provide stronger fault isolation and to support delegation of administrative control, we present a hierarchical, multi-level one-hop DHT.

Traffic-driven location resolution and caching: To forward packets along shortest paths and to avoid excessive load on the directory service, switches cache responses to queries. In enterprise networks, hosts typically communicate with a small number of other hosts [28], making caching highly effective. Furthermore, SEATTLE also provides a way to piggyback location information on ARP replies, which eliminates the need for location resolution when forwarding data packets. This allows data packets to directly traverse the shortest path, making the network's forwarding behavior predictable and stable.

A scalable, prompt cache-update protocol: Unlike Ethernet which relies on timeouts or broadcasts to keep forwarding tables up-to-date, SEATTLE proposes an explicit and reliable cache update protocol based on unicast. This ensures that all packets are delivered based on up-to-date state while keeping control overhead low. In contrast to conventional DHTs, this update process is directly triggered by network-layer changes, providing fast reaction times. For example, by observing link-state advertisements, switches determine when a host's location is no longer reachable, and evict those invalid entries. Through these approaches, SEATTLE seamlessly supports host mobility and other dynamics.

Despite these features, our design remains compatible with existing applications and protocols running at end hosts. For example, SEATTLE allows hosts to generate broadcast ARP and DHCP messages, and internally converts them into unicast queries to a directory service. SEATTLE switches can also handle general (i.e., non-ARP and non-

DHCP) broadcast traffic through loop-free multicasting. To offer broadcast scoping and access control, SEATTLE also provides a more scalable and flexible mechanism that allows administrators to create VLANs without trunk configuration.

2.1.2 Related work

Our quest is to design, implement, and evaluate a practical replacement for Ethernet that scales to *large and dynamic* networks. Although there are many approaches to enhance Ethernet bridging, none of these are suitable for our purposes. RBridges [29, 30] leverage a link-state protocol to disseminate information about both bridge connectivity and host state. This eliminates the need to maintain a spanning tree and improves forwarding paths. CMU-Ethernet [31] also leverages link-state and replaces end-host broadcasting by propagating host information in link-state updates. Viking [32] uses multiple spanning trees for faster fault recovery, which can be dynamically adjusted to conform to changing load. SmartBridges [33] allows shortest-path forwarding by obtaining the network topology, and monitoring which end host is attached to each switch. However, its control-plane overheads and storage requirements are similar to Ethernet, as every end host's information is disseminated to every switch. Though SEATTLE was inspired by the problems addressed in these works, it takes a radically different approach that *eliminates* network-wide dissemination of per-host information. This results in substantially improved control-plane scalability and data-plane efficiency. While there has been work on using hashing to support flat addressing conducted in parallel with our work [34, 35], these works do not promptly handle host dynamics, require some packets to be detoured away from the shortest path or be forwarded along a spanning tree, and do not support hierarchical configurations to ensure fault/path isolation and the delegation of administrative control necessary for large networks.

The design we propose is also substantially different from recent work on identity-based routing (ROFL [21], UIP [36], and VRR [37]. Our solution is suitable for building a practical and easy-to-manage network for several reasons. First, these previous approaches determine paths based on a hash of the destination's identifier (or the identifier itself), incurring a stretch penalty (which is unbounded in the worst case). In contrast, SEATTLE does *not* perform identity-based routing. Instead, SEATTLE uses resolution to map a MAC address to a host's location, and then uses the location to deliver packets along the *shortest path* to the host. This reduces latency and makes it easier to control and predict network behavior. Predictability and controllability are extremely important in real networks, because they make essential management tasks (e.g., capacity planning, troubleshooting, traffic engineering) possible. Second, the path between two hosts in a SEATTLE network does not change as other hosts join and leave the network. This substantially reduces packet reordering and improves constancy of path performance. Finally, SEATTLE employs traffic-driven caching of host information, as opposed to the traffic-agnostic caching (e.g., finger caches in ROFL) used in previous works. By only caching information that is needed to forward packets, SEATTLE significantly reduces the amount of state required to deliver packets. However, our design also consists of several generic components, such as the multi-level one-hop DHT and service discovery mechanism, that could be adapted to the work in [21, 36, 37].

2.2 Today's Enterprise and Access Networks

To provide background for the remainder of the chapter, and to motivate SEATTLE, this section explains why Ethernet bridging does not scale. Then we describe hybrid IP/Ethernet networks and VLANs, two widely-used approaches which improve scalabil-

ity over conventional Ethernet, but introduce management complexity, eliminating the "plug-and-play" advantages of Ethernet.

2.2.1 Ethernet bridging

An Ethernet network is composed of *segments*, each comprising a single physical layer [1]. Ethernet *bridges* are used to interconnect multiple segments into a multi-hop network, namely a LAN, forming a single *broadcast domain*. Each host is assigned a unique 48-bit MAC (Media Access Control) address. A bridge learns how to reach hosts by inspecting the incoming frames, and associating the source MAC address with the incoming port. A bridge stores this information in a *forwarding table* that it uses to forward frames toward their destinations. If the destination MAC address is not present in the forwarding table, the bridge sends the frame on all outgoing ports, initiating a domain-wide flood. Bridges also flood frames that are destined to a broadcast MAC address. Since Ethernet frames do not carry a TTL (Time-To-Live) value, the existence of multiple paths in the topology can lead to *broadcast storms*, where frames are repeatedly replicated and forwarded along a loop. To avoid this, bridges in a broadcast domain coordinate to compute a *spanning tree* [38]. Administrators first select and configure a single *root bridge*; then, the bridges collectively compute a spanning tree based on distances to the root. Links not present in the tree are not used to carry traffic, causing longer paths and inefficient use of resources. Unfortunately, Ethernet-bridged networks cannot grow to a large scale due to following reasons.

Globally disseminating every host's location: Flooding and source-learning introduce two problems in a large broadcast domain. First, the forwarding table at a bridge can

[1] In modern switched Ethernet networks, a segment is just a point-to-point link connecting an end host and a bridge, or a pair of bridges.

grow very large because flat addressing increases the table size proportionally to the total number of hosts in the network. Second, the control overhead required to disseminate each host's information via flooding can be very large, wasting link bandwidth and processing resources. Since hosts (or their network interfaces) power up/down (manually, or dynamically to reduce power consumption), and change location relatively frequently, flooding is an expensive way to keep per-host information up-to-date. Moreover, malicious hosts can intentionally trigger repeated network-wide floods through, for example, MAC address scanning attacks [39].

Inflexible route selection: Forcing all traffic to traverse a single spanning tree makes forwarding more failure-prone and leads to suboptimal paths and uneven link loads. Load is especially high on links near the root bridge. Thus, choosing the right root bridge is extremely important, imposing an additional administrative burden. Moreover, using a single tree for all communicating pairs, rather than shortest paths, significantly reduces the aggregate throughput of a network.

Dependence on broadcasting for basic operations: DHCP and ARP are used to assign IP addresses and manage mappings between MAC and IP addresses, respectively. A host broadcasts a DHCP-discovery message whenever it believes its network attachment point has changed. Broadcast ARP requests are generated more frequently, whenever a host needs to know the MAC address associated with the IP address of another host in the same broadcast domain. Relying on broadcast for these operations degrades network performance. Moreover, every broadcast message must be processed by every end host; since handling of broadcast frames is often application or OS-specific, these frames are *not* handled by the network interface card, and instead must interrupt the CPU [40]. For portable devices on low-bandwidth wireless links, receiving ARP packets can consume a significant fraction of the available bandwidth, processing, and power resources. More-

over, the use of broadcasting for ARP and DHCP opens vulnerabilities for malicious hosts as they can easily launch ARP or DHCP floods [31].

2.2.2 Hybrid IP/Ethernet architecture

One way of dealing with Ethernet's limited scalability is to build enterprise and access provider networks out of multiple LANs interconnected by *IP routing*. In these *hybrid* networks, each LAN contains at most a few hundred hosts that collectively form an *IP subnet*. Communication across subnets is handled via certain fixed nodes called *default gateways*. Each IP subnet is allocated an *IP prefix*, and each host in the subnet is then assigned an IP address from the subnet's prefix. Assigning IP prefixes to subnets, and associating subnets with router interfaces is typically a manual process, as the assignment must follow the addressing hierarchy, yet must reduce wasted namespace, and must consider future use of addresses to minimize later reassignment. Unlike a MAC address, which functions as a host *identifier*, an IP address denotes the host's current *location* in the network.

The biggest problem of the hybrid architecture is its massive configuration overhead. Configuring hybrid networks today represents an enormous challenge. Some estimates put 70% of an enterprise network's operating cost as maintenance and configuration, as opposed to equipment costs or power usage [7]. In addition, involving human administrators in the loop increases reaction time to faults and increases potential for misconfiguration.

Configuration overhead due to hierarchical addressing: An IP router cannot function correctly until administrators specify subnets on router interfaces, and direct routing protocols to advertise the subnets. Similarly, an end host cannot access the network until it is configured with an IP address corresponding to the subnet where the host is currently

located. DHCP automates end-host configuration, but introduces substantial configuration overhead for managing the DHCP servers. In particular, maintaining consistency between routers' subnet configuration and DHCP servers' address allocation configuration, or coordination across distributed DHCP servers are not simple. Finally, network administrators must continually revise this configuration to handle network changes.

Complexity in implementing networking policies: Administrators today use a collection of access controls, QoS (Quality of Service) controls [41], and other policies to control the way packets flow through their networks. These policies are typically defined based on IP prefixes. However, since prefixes are assigned based on the topology, changes to the network design require these policies to be rewritten. More significantly, rewriting networking policies must happen immediately after network design changes to prevent reachability problems and to avoid vulnerabilities. Ideally, administrators should only need to update policy configurations when the *policy* itself, not the *network*, changes.

Limited mobility support: Supporting seamless host mobility is becoming increasingly important. In data centers, migratable virtual machines are being widely deployed to improve power efficiency by adapting to workload, and to minimize service disruption during maintenance operations. Large universities or enterprises often build campus-wide wireless networks, using a wired backhaul to support host mobility across access points. To ensure service continuity and minimize policy update overhead, it is highly desirable for a host to retain its IP address regardless of its location in these networks. Unfortunately, hybrid networks constrain host mobility only within a single, usually small, subnet. In a data center, this can interfere with the ability to handle load spikes seamlessly; in wireless backhaul networks, this can cause service disruptions. One way to deal with this is to increase the size of subnets by increasing broadcast domains, introducing the scaling problems mentioned in Section 2.2.1.

2.2.3 Virtual LANs

VLANs address some of the problems of Ethernet and IP networks. VLANs allow administrators to group multiple hosts sharing the same networking requirements into a single broadcast domain. Unlike a physical LAN, a VLAN can be defined *logically*, regardless of individual hosts' locations in a network. VLANs can also be overlapped by allowing bridges (not hosts) to be configured with multiple VLANs. By dividing a large bridged network into several appropriately-sized VLANs, administrators can reduce the broadcast overhead imposed on hosts in each VLAN, and also ensure isolation among different host groups. Compared with IP, VLANs simplify mobility, as hosts may retain their IP addresses while moving between bridges in the same VLAN. This also reduces policy reconfiguration overhead. Unfortunately, VLANs introduces several problems:

Trunk configuration overhead: Extending a VLAN across multiple bridges requires the VLAN to be trunked (provisioned) at each of the bridges participating in the VLAN. Deciding which bridges should be in a given VLAN must consider traffic and mobility patterns to ensure efficiency, and hence is often done manually.

Limited control-plane scalability: Although VLANs reduce the broadcast overhead imposed on a particular end host, bridges provisioned with multiple VLANs must maintain forwarding-table entries and process broadcast traffic for *every* active host in *every* VLAN visible to themselves. Unfortunately, to enhance resource utilization and host mobility, and to reduce trunk configuration overhead, VLANs are often provisioned larger than necessary, worsening this problem. A large forwarding table complicates bridge design, since forwarding tables in Ethernet bridges are typically implemented using Content-Addressable Memory (CAM), an expensive and power-intensive technology.

Insufficient data-plane efficiency: Larger enterprises and data centers often have richer

topologies, for greater reliability and performance. Unfortunately, a single spanning tree is used in each VLAN to forward packets, which prevents certain links from being used. Although configuring a disjoint spanning tree for each VLAN [32, 42] may improve load balance and increase aggregate throughput, effective use of per-VLAN trees requires periodically moving the roots and rebalancing the trees, which must be manually updated as traffic shifts. Moreover, inter-VLAN traffic must be routed via IP gateways, rather than shortest physical paths.

2.3 Network-Layer One-hop DHT

The goal of a conventional Ethernet is to route packets to a destination specified by a MAC address. To do this, Ethernet bridges collectively provide end hosts with a service that maps MAC addresses to physical locations. Each bridge implements this service by maintaining next-hop pointers associated with MAC addresses in its forwarding table, and relies on domain-wide flooding to keep these pointers up to date. Additionally, Ethernet also allows hosts to look up the MAC address associated with a given IP address by broadcasting *Address Resolution Protocol* (ARP) messages.

In order to provide the same interfaces to end hosts as conventional Ethernet, SEATTLE also needs a mechanism that maintains mappings between MAC/IP addresses and locations. To scale to large networks, SEATTLE operates a distributed directory service built using a *one-hop, network-level DHT*. We use a *one-hop* DHT to reduce lookup complexity and simplify certain aspects of network administration such as traffic engineering and troubleshooting. We use a *network-level* approach that stores mappings at switches, so as to ensure fast and efficient reaction to network failures and recoveries, and avoid the control overhead of a separate directory infrastructure. Moreover, our network-level

approach allows storage capability to increase naturally with network size, and exploits *caching* to forward data packets directly to the destination without needing to traverse any intermediate DHT hops [43, 44].

2.3.1 Scalable key-value management with a one-hop DHT

Our distributed directory has two main parts. First, running a link-state protocol ensures each switch can observe all other switches in the network, and allows any switch to route any other switch along shortest paths. Second, SEATTLE uses a *hash function* to map host information to a switch. This host information is maintained in the form of (*key, value*). Examples of these key-value pairs are (*MAC address, location*), and (*IP address, MAC address*).

Link-state protocol maintaining switch topology

SEATTLE enables shortest-path forwarding by running a link-state protocol. However, distributing *end-host* information in link-state advertisements, as advocated in previous proposals [31, 29, 33, 30], would lead to serious scaling problems in the large networks we consider. Instead, SEATTLE's link-state protocol maintains only the *switch*-level topology, which is much more compact and stable. SEATTLE switches use the link-state information to compute shortest paths for unicasting, and multicast trees for broadcasting.

To automate configuration of the link-state protocol, SEATTLE switches run a discovery protocol to determine which of their links are attached to hosts, and which are attached to other switches. Distinguishing between these different kinds of links is done by sending control messages that Ethernet hosts do not respond to. This process is similar to how Ethernet distinguishes switches from hosts when building its spanning tree. To identify themselves in the link-state protocol, SEATTLE switches determine their own

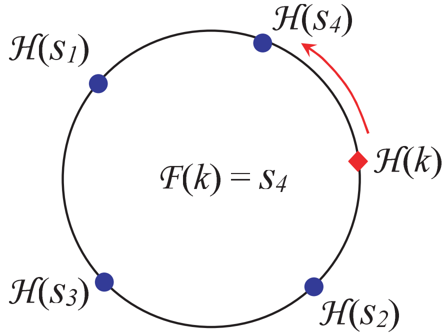


Figure 2.1: Keys are consistently hashed onto resolver switches (s_i).

unique *switch IDs* without administrator involvement. For example, each switch does this by choosing the MAC address of one of its interfaces as its switch ID.

Hashing key-value pairs onto switches

Instead of disseminating per-host information in link-state advertisements, SEATTLE switches learn this information in an on-demand fashion, via a simple hashing mechanism. This information is stored in the form of (key$=k$,value$=v$) pairs. A *publisher* switch s_a wishing to publish a (k, v) pair via the directory service uses a hash function $\mathcal{F}$ to map k to a switch identifier $\mathcal{F}(k) = r_k$, and instructs switch r_k to store the mapping (k, v). We refer to r_k as the *resolver* for k. A different switch s_b may then look up the value associated with k by using the same hash function to identify which switch is k's resolver. This works because each switch knows all the other switches' identifiers via link-state advertisements from the routing protocol, and hence $\mathcal{F}$ works identically across all switches. Switch s_b may then forward a lookup request to r_k to retrieve the value v. Switch s_b may optionally cache the result of its lookup, to reduce redundant resolutions. All control messages, including lookup and publish messages, are unicast with reliable delivery.

Reducing control overhead with consistent hashing: When the set of switches changes due to a network failure or recovery, some keys have to be re-hashed to different resolver switches. To minimize this re-hashing overhead, SEATTLE utilizes *Consistent Hashing* [45] for $\mathcal{F}$. This mechanism is illustrated in Figure 2.1. A consistent hashing function maps keys to *bins* such that the change of the bin set causes minimal churn in the mapping of keys to bins. In SEATTLE, each switch corresponds a bin, and a host's information corresponds to a key. Formally, given a set $S = \{s_1, s_2, ..., s_n\}$ of switch identifiers, and a key k,

$$\mathcal{F}(k) = argmin_{\forall s_i \in S}\{\mathcal{D}(\mathcal{H}(k), \mathcal{H}(s_i))\}$$

where $\mathcal{H}$ is a regular hash function, and $\mathcal{D}(x, y)$ is a simple metric function computing the counter-clockwise distance from x to y on the circular hash-space of $\mathcal{H}$. This means $\mathcal{F}$ maps a key to the switch with the closest identifier not exceeding that of the key on the hash space of $\mathcal{H}$. As an optimization, a key may be additionally mapped to the next m closest switches along the hash ring, to improve resilience to multiple failures. However, in our evaluation, we will assume this optimization is disabled by default.

Balancing load with virtual switches: The scheme described so far assumes that all switches are equally powerful, and hence low-end switches will need to service the same load as more powerful switches. To deal with this, we propose a new scheme based on running multiple *virtual switches* on each physical switch. A single switch locally creates one or more virtual switches. The switch may then increase or decrease its load by spawning/destroying these virtual switches. Unlike techniques used in traditional DHTs for load balancing [44], it is *not* necessary for our virtual switches to be advertised to other physical switches. To reduce size of link-state advertisements, instead of advertising

every virtual switch in the link-state protocol, switches only advertise the number of virtual switches they are currently running. Each switch then locally computes virtual switch IDs using the following technique. All switches use the same function $\mathcal{R}(s, i)$ that takes as input a switch identifier s and a number i, and outputs a new identifier unique to the inputs. A physical switch w only advertises in link-state advertisements its own physical switch identifier s_w and the number L of virtual switches it is currently running. Every switch can then determine the virtual identifiers of w by computing $\mathcal{R}(s_w, i)$ for $1 \leq i \leq L$. Note that it is possible to automate determining a desirable number of virtual switches per physical switch [46].

Enabling flexible service discovery: This design also enables more flexible service discovery mechanisms without the need to perform network-wide broadcasts. This is done by utilizing the hash function $\mathcal{F}$ to map a string defining the service to a switch. For example, a printer may hash the string *"PRINTER"* to a switch, at which it may store its location or address information. Other switches can then reach the printer using the hash of the string. Services may also encode additional attributes, such as load or network location, as simple extensions. Multiple servers can redundantly register themselves with a common string to implement anycasting. Services can be named using techniques shown in previous work [47].

2.3.2 Responding to topology changes

The switch-level topology may change if a new switch/link is added to the network, an existing switch/link fails, or a previously failed switch/link recovers. These failures may or may not *partition* the network into multiple disconnected components. Link failures are typically more common than switch failures, and partitions are very rare if the network has sufficient redundancy.

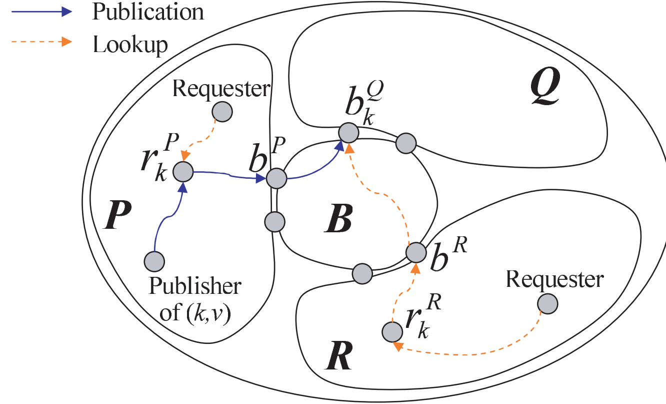


Figure 2.2: Hierarchical SEATTLE hashes keys onto *regions*.

In the case of a link failure/recovery that does not partition a network, the set of switches appearing in the link-state map does not change. Since the hash function $\mathcal{F}$ is defined with the set of switches in the network, the resolver a particular key maps to will not change. Hence all that needs to be done is to update the link-state map to ensure packets continue to traverse new shortest paths. In SEATTLE, this is simply handled by the link-state protocol.

However, if a switch fails or recovers, the set of switches in the link-state map changes. Hence there may be some keys k whose old resolver r_k^{old} differs from a new resolver r_k^{new}. To deal with this, the value (k, v) must be moved from r_k^{old} to r_k^{new}. This is handled by having the switch s_k that originally published k monitor the liveness of k's resolver through link-state advertisements. When s_k detects that r_k^{new} differs from r_k^{old}, it republishes (k, v) to r_k^{new}. The value (k, v) is eventually removed from r_k^{old} after a timeout. Additionally, when a value v denotes a location, such as a switch id s, and s goes down, each switch scans the list of locally-stored (k, v) pairs, and remove all entries whose value v equals s. Note this procedure correctly handles network partitions because the link-state protocol ensures that each switch will be able to see only switches present

in its partition.

2.3.3 Supporting hierarchy with a multi-level, one-hop DHT

The SEATTLE design presented so far scales to large, dynamic networks [48]. However, since this design runs a single, network-wide link-state routing protocol, it may be inappropriate for networks with highly dynamic infrastructure, such as networks in developing regions [26]. A single network-wide protocol may also be inappropriate if network operators wish to provide stronger fault isolation across geographic regions, or to divide up administrative control across smaller routing domains. Moreover, when a SEATTLE network is deployed over a wide area, the resolver could lie far both from the source and destination. Forwarding lookups over long distances increases latency and makes the lookup more prone to failure. To deal with this, SEATTLE may be configured hierarchically, by leveraging a *multi-level, one-hop DHT*. This mechanism is illustrated in Figure 2.2.

A hierarchical network is divided into several *regions*, and a *backbone* providing connectivity across regions. Each region is connected to the backbone via its own *border switch*, and the backbone is composed of the border switches of all regions. Information about regions is summarized and propagated in a manner similar to *areas* in OSPF. In particular, each switch in a region knows the identifier of the region's border switch, because the border switch advertises its role through the link-state protocol. In such an environment, SEATTLE ensures that only inter-region lookups are forwarded via the backbone while all regional lookups are handled within their own regions, and link-state advertisements are only propagated locally within regions. SEATTLE ensures this by defining a separate *regional* and *backbone* hash ring. When a (k, v) is inserted into a region P and is published to a regional resolver r_k^P (i.e., a resolver for k in region P), r_k^P additionally

forwards (k, v) to one of the region P's border switches b^P. Then b^P hashes k again onto the backbone ring and publishes (k, v) to another backbone switch b_k^Q, which is a backbone resolver for k and a border switch of region Q at the same time. Switch b_k^Q stores k's information. If a switch in region R wishes to lookup (k, v), it forwards the lookup first to its local resolver r_k^R, which in turn forwards it to b^R, and b^R forwards it to b_k^Q. As an optimization to reduce load on border switches, b_k^Q may hash k and store (k, v) at a switch within its own region Q, rather than storing (k, v) locally. Since switch failures are not propagated across regions, each publisher switch periodically sends probes to backbone resolvers that lie outside of its region. To improve availability, (k, v) may be stored at multiple backbone resolvers (as described in Section 2.3.1), and multiple simultaneous lookups may be sent in parallel.

2.4 Scaling Ethernet with a One-hop DHT

The previous section described the design of a distributed network-level directory service based on a one-hop DHT. In this section, we describe how the directory service is used to provide efficient packet delivery and scalable address resolution. We first briefly describe how to forward data packets to MAC addresses in Section 2.4.1. We then describe our remaining contributions: an optimization that eliminate the need to look up host location in the DHT by piggy-backing that information on ARP requests in Section 2.4.2, and a scalable dynamic cache-update protocol in Section 2.4.3.

2.4.1 Host location resolution

Hosts use the directory service described in Section 2.3 to publish and maintain mappings between their MAC addresses and their current locations. These mappings are used to

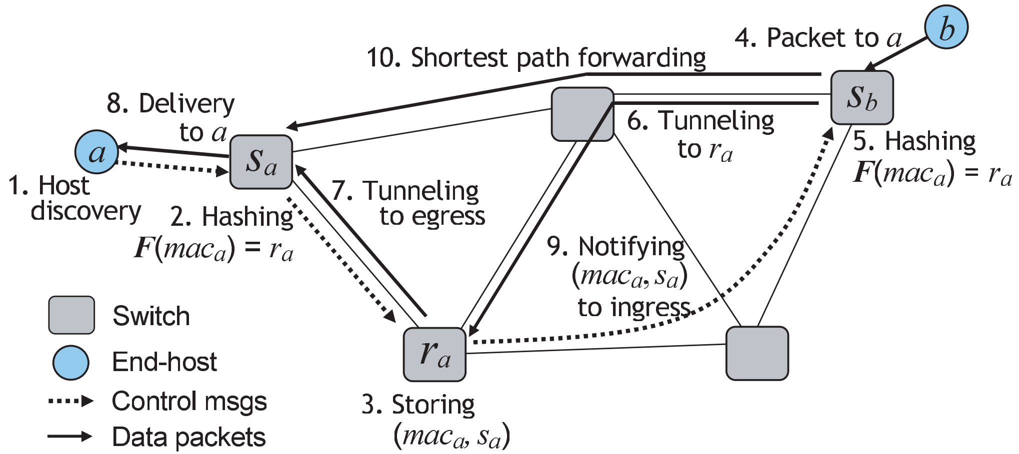


Figure 2.3: Packet forwarding and lookup in SEATTLE

forward data packets, using the procedure shown in Figure 2.3. When a host a with MAC address mac_a first arrives at its access switch s_a, the switch must publish a's MAC-to-location mapping in the directory service. Switch s_a does this by computing $\mathcal{F}(mac_a) = r_a$, and instructing r_a to store (mac_a, s_a). We refer to r_a as the *location resolver* for a. Then, if some host b connected to switch s_b wants to send a data packet to mac_a, b forwards the data packet to s_b, which in turn computes $\mathcal{F}(mac_a) = r_a$. Switch s_b then and forwards the packet to r_a. Since r_a may be several hops away, s_b encapsulates the packet with an outer header with r_a's address as the destination. Switch r_a then looks up a's location s_a, and forwards the packet on towards s_a. In order to limit the number of data packets traversing the resolver, r_a also notifies s_b that a's current location is s_a. Switch s_b then caches this information. While forwarding the first few packets of a flow via a resolver switch increases path lengths, in the next section we describe an optimization that allows data packets to traverse only shortest paths, by piggy-backing location information on ARP replies.

Note SEATTLE manages per-host information via reactive resolution, as opposed to the proactive dissemination scheme used in previous approaches [31, 29, 33]. The scal-

ing benefits of this reactive resolution increase in enterprise/data-center/access provider networks because most hosts communicate with a small number of popular hosts, such as mail/file/Web servers, printers, VoIP gateways, and Internet gateways [28]. To prevent forwarding tables from growing unnecessarily large, the access switches can apply various cache-management policies. For correctness, however, the cache-management scheme must not evict the host information of the hosts that are directly connected to the switch or are registered with the switch for resolution. Unlike Ethernet bridging, cache misses in SEATTLE do not lead to flooding, making the network resistant to cache poisoning attacks (e.g., forwarding table overflow attack) or a sudden shift in traffic. Moreover, those switches that are not directly connected to end hosts (i.e., aggregation or core switches) do not need to maintain any cached entries.

2.4.2 Host address resolution

In conventional Ethernet, a host with an IP packet first broadcasts an ARP request to look up the MAC address of the host owning the destination IP address contained in the request. To enhance scalability, SEATTLE avoids broadcast-based ARP operations. In addition, we extend ARP to return both the *location* and the MAC address of the end host to the requesting switch. This allows data packets following an ARP query to directly traverse shortest paths.

SEATTLE replaces the traditional broadcast-based ARP with an extension to the one-hop DHT directory service. In particular, switches use $\mathcal{F}$ with an IP address as the key. Specifically, when host a arrives at access switch s_a, the switch learns a's IP address ip_a (using techniques described in Section 2.5.1), and computes $\mathcal{F}(ip_a) = v_a$. The result of this computation is the identifier of another switch v_a. Finally, s_a informs v_a of (ip_a, mac_a). Switch v_a, the *address resolver* for host a, then uses the tuple to handle future

ARP requests for ip_a redirected by other remote switches. Note that host a's location resolver (i.e., $\mathcal{F}(mac_a)$) may differ from a's address resolver (i.e., $\mathcal{F}(ip_a)$).

Optimizing forwarding paths via ARP: For hosts that issue an ARP request, SEATTLE eliminates the need to perform forwarding via the location resolver as mentioned in Section 2.4.1. This is done by having the address resolver switch v_a also maintain the location of a (i.e., s_a) in addition to mac_a. Upon receiving an ARP request from some host b, the address resolver v_a returns both mac_a and s_a back to b's access switch s_b. Switch s_b then caches s_a for future packet delivery, and returns mac_a to host b. Any packets sent by b to a are then sent directly along the shortest path to a.

It is, however, possible that host b already has mac_a in its ARP cache and immediately sends data frames destined to mac_a without issuing an ARP request in advance. Even in such a case, as long as the s_b also maintains a's location associated with mac_a, s_b can forward those frames correctly. To ensure access switches cache the same entries as hosts, the timeout value that an access switch applies to the cached location information should be larger than the ARP cache timeout used by end hosts [2]. Note that, even if the cache and the host become out of sync (due to switch reboot, etc.), SEATTLE continues to operate correctly because switches can resolve a host's location by hashing the host's MAC address to the host's location resolver.

2.4.3 Handling host dynamics

Hosts can undergo three different kinds of changes in a SEATTLE network. First, a host may change location, for example if it has physically moved to a new location (e.g., wireless handoff), if its link has been plugged into a different access switch, or if it is a

[2]The default setting of the ARP cache timeout in most common operating systems ranges 10 to 20 minutes.

virtual machine and has migrated to a new hosting system that allows the VM to retain its MAC address. Second, a host may change its MAC address, for example if its NIC card is replaced, if it is a VM and has migrated to a new hosting system that requires the VM to use the host's MAC address, or if multiple physical machines collectively acting as a single server or router (to ensure high availability) experience a fail-over event [49]. Third, a host may change its IP address, for example if a DHCP lease expires, or if the host is manually reconfigured. In practice, multiple of these changes may occur simultaneously. When these changes occur, we need to keep the directory service up-to-date, to ensure correct packet delivery.

SEATTLE handles these changes by modifying the contents of the directory service via *insert*, *delete*, and *update* operations. An insert operation adds a new (k, v) pair to the DHT, a delete operation removes a (k, v) pair from the DHT, and the update operation updates the value v associated with a given key k. First, in the case of a location change, the host h moves from one access switch s_h^{old} to another s_h^{new}. In this case, s_h^{new} inserts a new MAC-to-location entry. Since h's MAC address already exists in the DHT, this action will update h's old location with its new location. Second, in the case of a MAC address change, h's access switch s_h inserts an IP-to-MAC entry containing h's new MAC address, causing h's old IP-to-MAC mapping to be updated. Since a MAC address is also used as a key of a MAC-to-location mapping, s_h deletes h's old MAC-to-location mapping and inserts a new mapping, respectively with the old and new MAC addresses as keys. Third, in the case of an IP address change, we need to ensure that future ARP requests for h's old IP address are no longer resolved to h's MAC address. To ensure this, s_h deletes h's old IP-to-MAC mapping and insert the new one. Finally, if multiple changes happen at once, the above steps occur simultaneously.

Ensuring seamless mobility: As an example, consider the case of a mobile host h mov-

ing between two access switches, s_h^{old} and s_h^{new}. To handle this, we need to update h's MAC-to-location mapping to point to its new location. As described in Section 2.4.1, s_h^{new} inserts (mac_h, s_h^{new}) into r_h upon arrival of h. Note that the location resolver r_h selected by $\mathcal{F}(mac_h)$ does *not* change when h's location changes. Meanwhile, s_h^{old} deletes (mac_h, s_h^{old}) when it detects h is unreachable (either via timeout or active polling). Additionally, to enable prompt removal of stale information, the location resolver r_h informs s_h^{old} that (mac_h, s_h^{old}) is obsoleted by (mac_h, s_h^{new}).

However, host locations cached at other access switches must be kept up-to-date as hosts move. SEATTLE takes advantage of the fact that, even after updating the information at r_h, s_h^{old} may receive packets destined to h because other access switches in the network might have the stale information in their forwarding tables. Hence, when s_h^{old} receives packets destined to h, it explicitly notifies ingress switches that sent the misdelivered packets of h's new location s_h^{new}. To minimize service disruption, s_h^{old} also forwards those misdelivered packets s_h^{new}.

Updating remote hosts' caches: In addition to updating contents of the directory service, some host changes require informing other *hosts* in the system about the change. For example, if a host h changes its MAC address, the new mapping (ip_h, mac_h^{new}) must be immediately known to other hosts who happened to store (ip_h, mac_h^{old}) in their local ARP caches. In conventional Ethernet, this is achieved by broadcasting a *gratuitous ARP request* originated by h [50]. A gratuitous ARP is an ARP request containing the MAC and IP address of the host sending it. This request is not a query for a reply, but is instead a notification to update other end hosts' ARP tables and to detect IP address conflicts on the subnet. Relying on broadcast to update other hosts clearly does not scale to large networks. SEATTLE avoids this problem by unicasting gratuitous ARP packets only to hosts with invalid mappings. This is done by having s_h maintain a *MAC revocation list*.

Upon detecting h's MAC address change, switch s_h inserts $(ip_h, mac_h^{old}, mac_h^{new})$ in its revocation list. From then on, whenever s_h receives a packet whose source or destination (IP, MAC) address pair equals (ip_h, mac_h^{old}), it sends a *unicast* gratuitous ARP request containing (ip_h, mac_h^{new}) to the source host which sent those packets. Note that, when both h's MAC address and location change at the same time, the revocation information is created at h's old access switch by h's address resolver $v_h = \mathcal{F}(ip_h)$.

To minimize service disruption, s_h also informs the source host's ingress switch of (mac_h^{new}, s_h) so that the packets destined to mac_h^{new} can then be directly delivered to s_h, avoiding an additional location lookup. Note this approach to updating remote ARP caches does not require s_h to look up each packet's IP and MAC address pair from the revocation list because s_h can skip the lookup in the common case (i.e., when its revocation list is empty). Entries from the revocation list are removed after a timeout set equal to the ARP cache timeout of end hosts.

2.5 Providing Ethernet-like Semantics

To be fully backwards-compatible with conventional Ethernet, SEATTLE must act like a conventional Ethernet from the perspective of end hosts. First, the way that hosts interact with the network to bootstrap themselves (e.g., acquire addresses, allow switches to discover their presence) must be the same as Ethernet. Second, switches have to support traffic that uses broadcast/multicast Ethernet addresses as destinations. In this section, we describe how to perform these actions without incurring the scalability challenges of traditional Ethernet. For example, we propose to eliminate broadcasting from the two most popular sources of broadcast traffic: ARP and DHCP. Since we described how SEATTLE switches handle ARP without broadcasting in Section 2.4.2, we discuss only DHCP in

this section.

2.5.1 Bootstrapping hosts

Host discovery by access switches: When an end host arrives at a SEATTLE network, its access switch needs to discover the host's MAC and IP addresses. To discover a new host's MAC address, SEATTLE switches use the same MAC learning mechanism as conventional Ethernet, except that MAC learning is enabled only on the ports connected to end hosts. To learn a new host's IP address or detect an existing host's IP address change, SEATTLE switches snoop on gratuitous ARP requests. Most operating systems generate a gratuitous ARP request when the host boots up, the host's network interface or links comes up, or an address assigned to the interface changes [50]. If a host does not generate a gratuitous ARP, the switch can still learn of the host's IP address via snooping on DHCP messages, or sending out an ARP request only on the port connected to the host. Similarly, when an end host fails or disconnects from the network, the access switch is responsible for detecting that the host has left, and deleting the host's information from the network.

Host configuration without broadcasting: For scalability, SEATTLE resolves DHCP messages without broadcasting. When an access switch receives a broadcast DHCP discovery message from an end host, the switch delivers the message directly to a DHCP server via unicast, instead of broadcasting it. SEATTLE implements this mechanism using the existing DHCP relay agent standard [51]. This standard is used when an end host needs to communicate with a DHCP server outside the host's broadcast domain. The standard proposes that a host's IP gateway forward a DHCP discovery to a DHCP server via IP routing. In SEATTLE, a host's access switch can perform the same function with

Ethernet encapsulation. Access switches can discover a DHCP server using a similar approach to the service discovery mechanism in Section 2.3.1. For example, the DHCP server hashes the string "DHCP_SERVER" to a switch, and then stores its location at that switch. Other switches then forward DHCP requests using the hash of the string.

2.5.2 Scalable and flexible VLANs

SEATTLE completely eliminates flooding of unicast packets. However, to offer the same semantics as Ethernet bridging, SEATTLE needs to support transmission of packets sent to a *broadcast address*. Supporting broadcasting is important because some applications (e.g., IP multicast, peer-to-peer file sharing programs, etc.) rely on subnet-wide broadcasting. However, in large networks to which our design is targeted, performing broadcasts in the same style as Ethernet may significantly overload switches and reduce data plane efficiency. Instead, SEATTLE provides a mechanism which is similar to, but more flexible than, VLANs.

In particular, SEATTLE introduces a notion of *group*. Similar to a VLAN, a group is defined as a set of hosts who share the same broadcast domain regardless of their location. Unlike Ethernet bridging, however, a broadcast domain in SEATTLE does not limit unicast layer-2 reachability between hosts because a SEATTLE switch can resolve any host's address or location without relying on broadcasting. Thus, groups provide several additional benefits over VLANs. First, groups do not need to be manually assigned to switches. A group is automatically extended to cover a switch as soon as a member of that group arrives at the switch[3]. Second, a group is not forced to correspond to a single IP subnet, and hence may span multiple subnets or a portion of a subnet, if desired.

[3]The way administrators associate a host with corresponding group is beyond the scope of this dissertation. For Ethernet, management systems that can automate this task (e.g., mapping an end host or flow to a VLAN) are already available [52], and SEATTLE can employ the same model.

Third, unicast reachability in layer-2 between two different groups may be allowed (or restricted) depending on the access-control policy — a rule set defining which groups can communicate with which — between the groups.

The flexibility of groups ensures several benefits that are hard to achieve with conventional Ethernet bridging and VLANs. When a group is aligned with a subnet, and unicast reachability between two different groups is not permitted by default, groups provide exactly the same functionality as VLANs. However, groups can include a large number of end hosts and can be extended to anywhere in the network without harming control-plane scalability and data-plane efficiency. Moreover, when groups are defined as subsets of an IP subnet, and inter-group reachability is prohibited, each group is equivalent to a private VLAN (PVLAN), which are popularly used in hotel/motel networks [53]. Unlike PVLANs, however, groups can be extended over multiple bridges. Finally, when unicast reachability between two groups is allowed, traffic between the groups takes the shortest path, without traversing default gateways.

Multicast-based group-wide broadcasting: Some applications may rely on subnet-wide broadcasting. To handle this, all broadcast packets within a group are delivered through a multicast tree sourced at a dedicated switch, namely a *broadcast root*, of the group. The mapping between a group and its broadcast root is determined by using $\mathcal{F}$ to hash the group's identifier to a switch. Construction of the multicast tree is done in a manner similar to IP multicast, inheriting its safety (i.e., loop freedom) and efficiency (i.e., to receive broadcast only when necessary). When a switch first detects an end host that is a member of group g, the switch issues a join message that is carried up to the nearest graft point on the tree toward g's broadcast root. When a host departs, its access switch prunes a branch if necessary. When an end host in g sends a broadcast packet, its access switch marks the packet with g and forwards it along g's multicast tree.

Separating unicast reachability from broadcast domains: In addition to handling broadcast traffic, groups in SEATTLE also provide a namespace upon which reachability policies for unicast traffic are defined. When a host arrives at an access switch, the host's group membership is determined by its access switch and published to the host's resolvers along with its location information. Access control policies are then applied by a resolver when a host attempts to look up a destination host's information.

In this section, we start by describing our simulation environment. Next, we describe SEATTLE's performance under workloads collected from several real operational networks. We then investigate SEATTLE's performance in dynamic environments by generating host mobility and topology changes.

2.5.3 Methodology

To evaluate the performance of SEATTLE, we would ideally like to have several pieces of information, including complete layer-two topologies from a number of representative enterprises and access providers, traces of all traffic sent on every link in their topologies, the set of hosts at each switch/router in the topology, and a trace of host movement patterns. Unfortunately, network administrators (understandably) were not able to share this detailed information with us due to privacy concerns, and also because they typically do not log events on such large scales. Hence, we leveraged real traces where possible, and supplemented them with synthetic traces. To generate the synthetic traces, we made realistic assumptions about workload characteristics, and varied these characteristics to measure the sensitivity of SEATTLE to our assumptions.

In our packet-level simulator, we replayed packet traces collected from the Lawrence Berkeley National Lab campus network by Pang et. al. [54]. There are four sets of traces, each collected over a period of 10 to 60 minutes, containing traffic to and from

roughly 9,000 end hosts distributed over 22 different subnets. The end hosts were running various operating systems and applications, including malware (some of which engaged in scanning). To evaluate sensitivity of SEATTLE to network size, we artificially injected additional hosts into the trace. We did this by creating a set of virtual hosts, which communicated with a set of random destinations, while preserving the distribution of destination-level popularity of the original traces. We also tried injecting MAC scanning attacks and artificially increasing the rate at which hosts send [39].

We measured SEATTLE's performance on four representative topologies. *Campus* is the campus network of a large (roughly 40,000 students) university in the United States, containing 517 routers and switches. *AP-small* (AS 3967) is a small access provider network consisting of 87 routers, and *AP-large* (AS 1239) is a larger network with 315 routers [55]. Because SEATTLE switches are intended to replace both IP routers and Ethernet bridges, the routers in these topologies are considered as SEATTLE switches in our evaluation. To investigate a wider range of environments, we also constructed a model topology called *DC*, which represents a typical data center network composed of four full-meshed core routers each of which is connected to a mesh of twenty one aggregation switches. This roughly characterizes a commonly-used topology in data centers [24].

Our topology traces were anonymized, and hence lack information about how many hosts are connected to each switch. To deal with this, we leveraged CAIDA Skitter traces [56] to roughly characterize this number for networks reachable from the Internet. However, since the CAIDA skitter traces form a sample representative of the wide-area, it is not clear whether they apply to the smaller-scale networks we model. Hence for *DC* and *Campus*, we assume that hosts are evenly distributed across leaf-level switches.

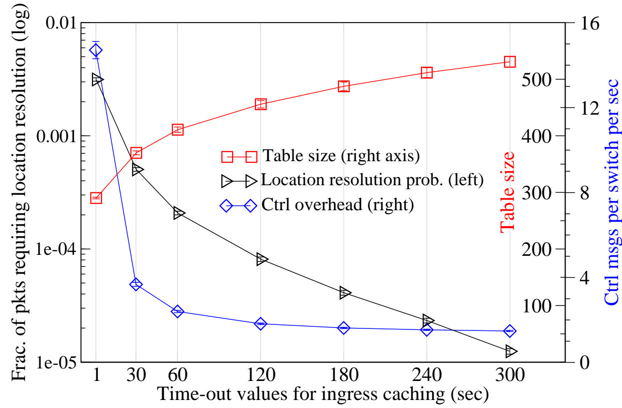
Given a fixed topology, the performance of SEATTLE and Ethernet bridging can vary depending on traffic patterns. To quantify this variation we repeated each simulation run

25 times, and plot the average of these runs with 99% confidence intervals. For each run we vary a random seed, causing the number of hosts per switch, and the mapping between hosts and switches to change. Additionally for the cases of Ethernet bridging, we varied spanning trees by randomly selecting one of the core switches as a root bridge. Our simulations assume that all switches are part of the same broadcast domain. However, since our traffic traces are captured in each of the 22 different subnets (i.e., broadcast domains), the traffic patterns among the hosts preserve the broadcast domain boundaries. Thus, our simulation network is equivalent to a VLAN-based network where a VLAN corresponds to an IP subnet, and all non-leaf Ethernet bridges are trunked with all VLANs to enhance mobility.

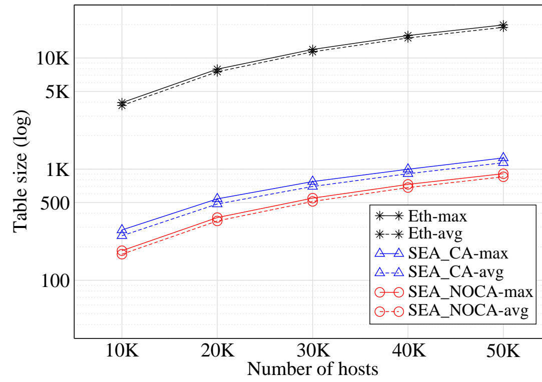
2.6 Simulations

2.6.1 Control-plane scalability

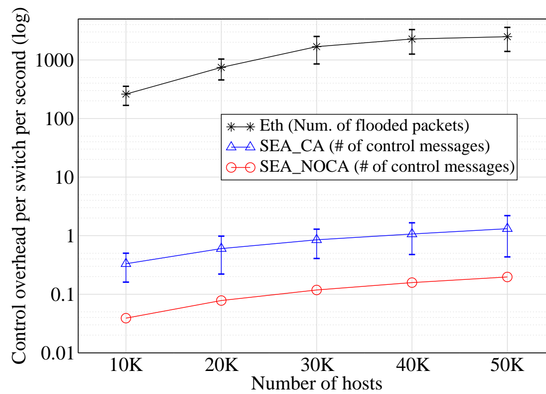
Sensitivity to cache eviction timeout: SEATTLE caches host information to route packets via shortest paths and to eliminate redundant resolutions. If a switch removes a host-information entry before a locally attached host does (from its ARP cache), the switch will need to perform a location lookup to forward data packets sent by the host. To eliminate the need to queue data packets at the ingress switch, those packets are forwarded through a location resolver, leading to a longer path. To evaluate this effect, we simulated a forwarding table management policy for switches that evicts unused entries after a timeout. Figure 2.4a shows performance of this strategy across different timeout values in the *AP-large* network. First, the fraction of packets that require data-driven location lookups (i.e., lookups not piggy-backed on ARPs) is very low and decreases quickly



(a)



(b)



(c)

Figure 2.4: (a) Effect of cache timeout in *AP-large* with $50K$ hosts, (b) Table size increase in *DC*, and (c) Control overhead in *AP-large*. Error bars in these figures show confidence intervals for each data point. A sufficient number of simulation runs reduced these intervals.

with larger timeout. Even for a very small timeout value of 60 seconds, over 99.98% of packets are forwarded without a separate lookup. We also confirmed that the number of data packets forwarded via location resolvers drops to zero when using timeout values larger than 600 seconds (i.e., roughly equal to the ARP cache timeout at end hosts). Also control overhead to maintain the directory decreases quickly, whereas the amount of state at each switch increases moderately with larger timeout. Hence, in a network with properly configured hosts and reasonably small (e.g., less than 2% of the total number of hosts in this topology) forwarding tables, SEATTLE always offers shortest paths.

Forwarding table size: Figure 2.4b shows the amount of state per switch in the *DC* topology. To quantify the cost of ingress caching, we show SEATTLE's table size with and without caching (*SEA_CA* and *SEA_NOCA* respectively). Ethernet requires more state than SEATTLE without caching, because Ethernet stores active hosts' information entries at almost every bridge. In a network with s switches and h hosts, each Ethernet bridge must be provisioned to store an entry for each destination, resulting in $O(sh)$ state requirements across the network. SEATTLE requires only $O(h)$ state since only the access and resolver switches need to store location information for each host. In this particular topology, SEATTLE reduces forwarding-table size by roughly a factor of 22. Although not shown here due to space constraints, we find that these gains increase to a factor of 64 in *AP-large* because there are a larger number of switches in that topology. While the use of caching drastically reduces the number of redundant location resolutions, we can see that it increases SEATTLE's forwarding-table size by roughly a factor of 1.5. However, even with this penalty, SEATTLE reduces table size compared with Ethernet by roughly a factor of 16. This value increases to a factor of 41 in *AP-large*.

Control overhead: Figure 2.4c shows the amount of control overhead generated by SEATTLE and Ethernet. We computed this value by dividing the total number of control

messages over all links in the topology by the number of switches, then dividing by the duration of the trace. SEATTLE significantly reduces control overhead as compared to Ethernet. This happens because Ethernet generates network-wide floods for a significant number of packets, while SEATTLE leverages unicast to disseminate host location. Here we again observe that use of caching degrades performance slightly. Specifically, the use of caching (*SEA_CA*) increases control overhead roughly from 0.1 to 1 packet per second as compared to *SEA_NOCA* in a network containing $30K$ hosts. However, *SEA_CA*'s overhead still remains a factor of roughly 1000 less than in Ethernet. In general, we found that the difference in control overhead increased roughly with the number of links in the network.

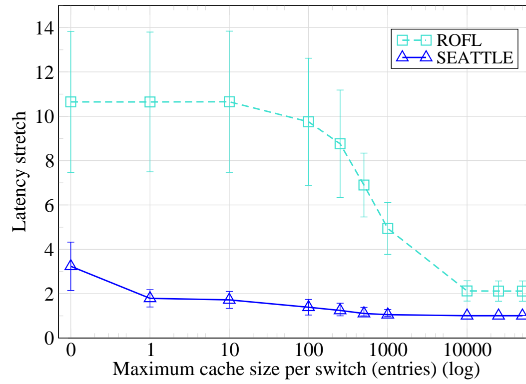
Comparison with id-based routing approaches: We implemented the ROFL, UIP, and VRR protocols in our simulator. To ensure a fair comparison, we used a link-state protocol to construct vset-paths [37] along shortest paths in UIP and VRR, and created a UIP/VRR node at a switch for each end host the switch is attached to. Performance of UIP and VRR was quite similar to performance of ROFL with an unbounded cache size. Figure 2.5a shows the average relative latency penalty, or *stretch*, of SEATTLE and ROFL [21] in the *AP-large* topology. We measured stretch by dividing the time the packet was in transit by the delay along the shortest path through the topology. Overall, SEATTLE incurs smaller stretch than ROFL. With a cache size of 1000, SEATTLE offers a stretch of roughly 1.07, as opposed to ROFL's 4.9. This happens because *i*) when a cache miss occurs, SEATTLE resolves location via a single-hop rather than a multi-hop lookup, and *ii*) SEATTLE's caching is driven by traffic patterns, and hosts in an enterprise network typically communicate with only a small number of popular hosts. Note that SEATTLE's stretch remains below 5 even when a cache size is 0. Hence, even with worst-case traffic patterns (e.g., every host communicates with all other hosts, switches

maintain very small caches), SEATTLE still ensures reasonably small stretch. Finally, we compare *path stability* with ROFL in Figure 2.5b. We vary the rate at which hosts leave and join the network, and measure path stability as the number of times a flow changes its path (the sequence of switches it traverses) in the presence of host churn. We find that ROFL has over three orders of magnitude more path changes than SEATTLE.

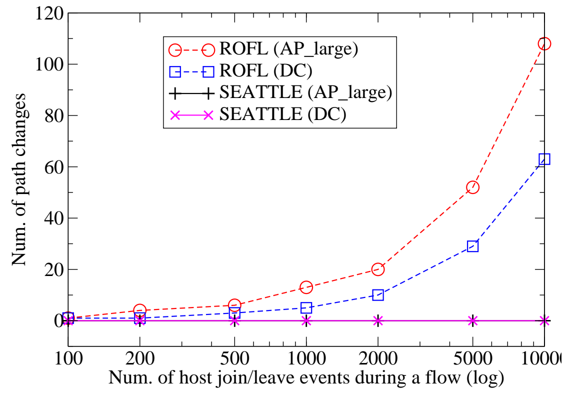
2.6.2 Sensitivity to network dynamics

Effect of network changes: Figure 2.5c shows performance during switch failures. Here, we cause switches to fail randomly, with failure inter-arrival times drawn from a Pareto distribution with $\alpha = 2.0$ and varying mean values. Switch recovery times are drawn from the same distribution, with a mean of 30 seconds. We found SEATTLE is able to deliver a larger fraction of packets than Ethernet. This happens because SEATTLE is able to use all links in the topology to forward packets, while Ethernet can only forward over a spanning tree. Additionally, after a switch failure, Ethernet must recompute this tree, which causes outages until the process completes. Although forwarding traffic through a location resolver in SEATTLE causes a flow's fate to be shared with a larger number of switches, we found that availability remained higher than that of Ethernet. Additionally, using caching improved availability further.

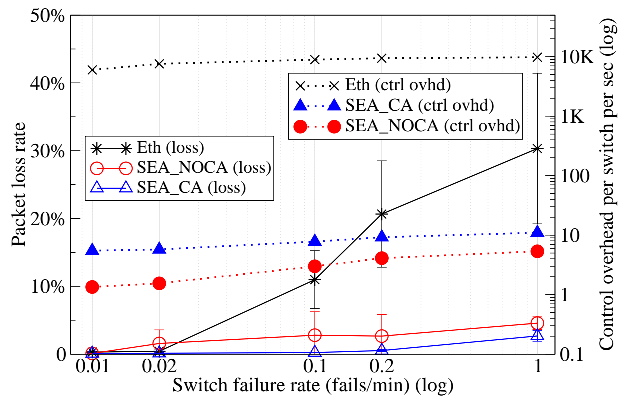
Effect of host mobility: To investigate the effect of physical or virtual host mobility on SEATTLE performance, we randomly move hosts between access switches. We drew mobility times from a Pareto distribution with $\alpha = 2.0$ and varying means. For high mobility rates, SEATTLE's loss rate is lower than Ethernet (Figure 2.6). This happens because when a host moves in Ethernet, it takes some time for switches to evict stale location information, and learn the host's new location. Although some host operating



(a)



(b)



(c)

Figure 2.5: (a) Stretch across different cache sizes in *AP-large* with $10K$ hosts, (b) Path stability, and (c) Effect of switch failures in *DC*.

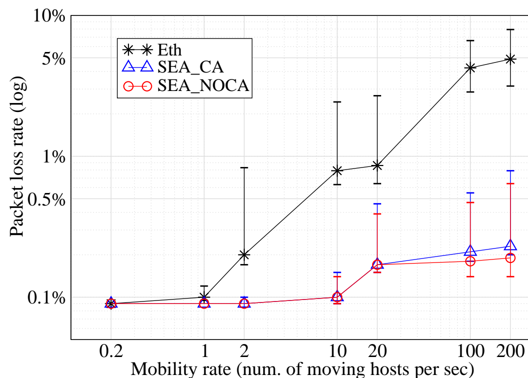


Figure 2.6: Effect of host mobility in *Campus*.

systems broadcast a gratuitous ARP when a host moves, this increases broadcast overhead. In contrast, SEATTLE provides both low loss and broadcast overhead by updating host state via unicasts.

2.7 Implementation

To verify SEATTLE's performance and practicality through a real deployment, we built a prototype SEATTLE switch using two open-source routing software platforms: user-level *Click* [57] and *XORP* [58]. We also implemented a second version of our prototype using kernel-level Click [23]. Section 2.7.1 describes the structure of our design, and Section 2.7.2 presents evaluation results.

2.7.1 Prototype design

Figure 2.7 shows the overall structure of our implementation. SEATTLE's control plane is divided into two functional modules: *i*) maintaining the switch-level topology, and *ii*) managing end-host information. We used XORP to realize the first functional module,

and used Click to implement the second. We also extended Click to implement SEATTLE's data-plane functions, including consistent hashing and packet encapsulation. Our control and data plane modifications to Click are implemented as the *SeattleSwitch* element shown in Figure 2.7.

SEATTLE control plane: First, we run a XORP OSPF process at each switch to maintain a complete switch-level network map. The XORP RIBD (Routing Information Base Daemon) constructs its routing table using this map. RIBD then installs the routing table into the forwarding plane process, which we implement with Click. Click uses this table, namely *NextHopTable*, to determine a next hop. The FEA (Forwarding Engine Abstraction) in XORP handles inter-process communication between XORP and Click. To maintain host information, a SeattleSwitch utilizes a *HostLocTable*, which is populated with three kinds of host information: (a) the outbound port for every local host; (b) the location for every remote host for which this switch is a resolver; and (c) the location for every remote host cached via previous lookups. For each insertion or deletion of a locally-attached host, the switch generates a corresponding registration or deregistration message. Additionally, by monitoring the changes of the NextHopTable, the switch can detect whether the topology has changed, and host re-registration is required accordingly. To maintain IP-to-MAC mappings to support ARP, a switch also maintains a separate table in the control plane. This table contains only the information of local hosts and remote hosts that are specifically hashed to the switch. When our prototype switch is first started up, a simple neighbor-discovery protocol is run to determine which interfaces are connected to other switches, and over each of these interfaces it initiates an OSPF session. The link weight associated with the OSPF adjacency is by default set to be the link latency. If desired, another metric may be used.

SEATTLE data plane: To forward packets, an ingress switch first learns an incoming

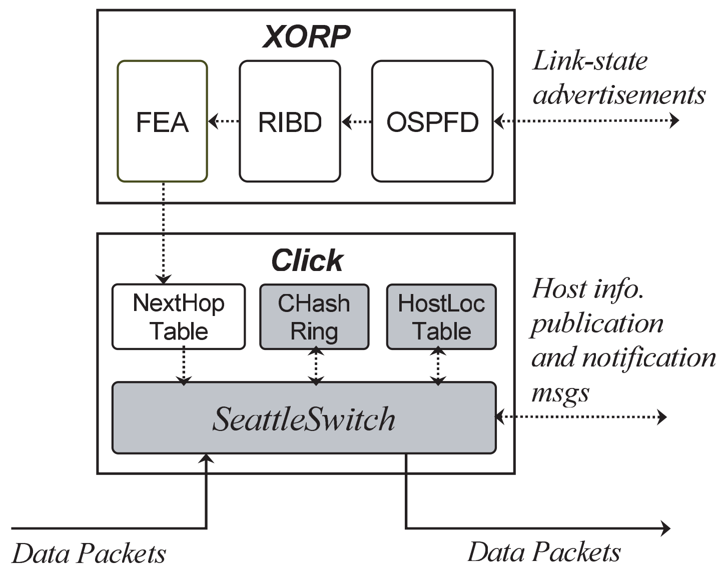


Figure 2.7: Implementation architecture.

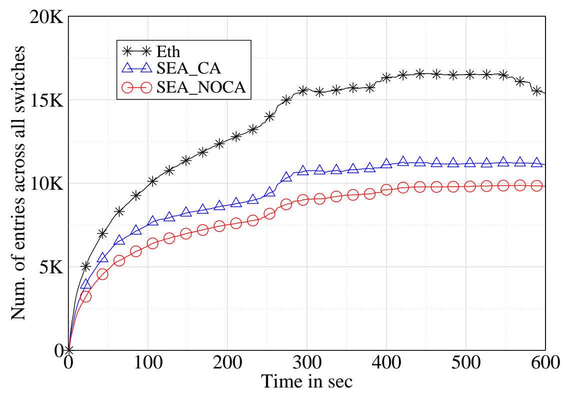
packet's source MAC address, and if necessary, adds the corresponding entry in HostLocTable. Then the switch looks up the destination MAC address in the HostLocTable and checks to see if *i*) the host is locally attached, *ii*) the host is remote, and its location is cached, or *iii*) the host is explicitly registered with the switch. In the case of *iii*) the switch needs to send a host location notification to the ingress. In all cases, the switch then forwards the packet either to the locally attached destination, or encapsulates the packet and forwards it to the next hop toward the destination. Intermediate switches can then simply forward the encapsulated packet by looking up the destination in their NextHopTables. In addition, if the incoming packet is an ARP request, the ingress switch executes the hash function $\mathcal{F}$ to look up the corresponding resolver's id, and re-writes the destination to that id, and delivers the packet to the resolver for resolution.

2.7.2 Experimental results

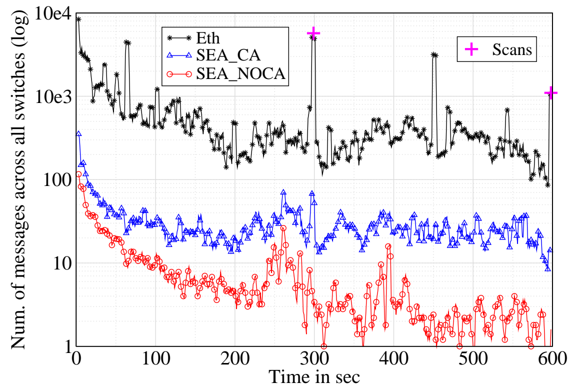
Next, we evaluate a deployment of our prototype implementation on Emulab. To ensure correctness, we cross-validated the simulator and implementation with various traces and topologies, and found that average stretch, control overhead, and table size from implementation results were within 3% of the values given by the simulator. We first present a set of microbenchmarks to evaluate per-packet processing overheads. Then, to evaluate dynamics of a SEATTLE network, we measure control overhead and switch state requirements, and evaluate switch fail-over performance.

Packet processing overhead: Table 2.1 shows per-packet processing time for both SEATTLE and Ethernet. We measure this as the time from when a packet enters the switch's inbound queue, to the time it is ready to be moved to an outbound queue. We break this time down into the major components. From the table, we can see that an ingress switch in SEATTLE requires more processing time than in Ethernet. This happens because the ingress switch has to encapsulate a packet and then look up the next-hop table with the outer header. However, SEATTLE requires less packet processing overhead than Ethernet at non-ingress hops, as intermediate and egress switches do not need to learn source MAC addresses, and consistent hashing (which takes around 2.2 us) is required only for ARP requests. Hence, SEATTLE requires less overall processing time on paths longer than 3.03 switch-level hops. In comparison, we found the average number of switch-level hops between hosts in a real university campus network (*Campus*) to be over 4 for the vast majority of host pairs. Using our kernel-level implementation of SEATTLE, we were able to fully saturate a 1 Gbps link.

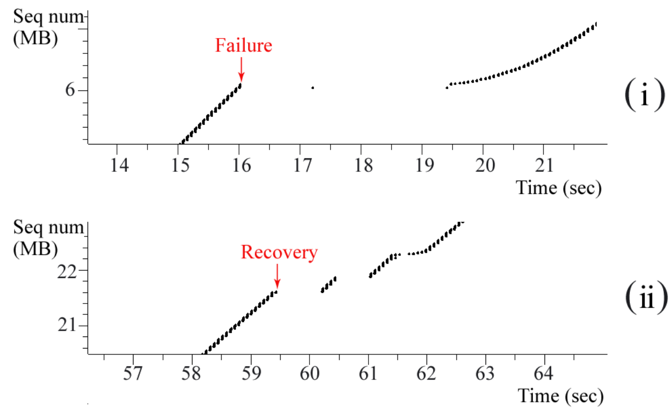
Effect of network dynamics: To evaluate the dynamics of SEATTLE and Ethernet, we instrumented the switch's internal data structures to periodically measure performance



(a)



(b)



(c)

Figure 2.8: Effect of network dynamics – (a) table size, (b) control overhead, and (c) failover performance.

Table 2.1: Per-packet processing time in micro-sec.

	learn src	*look-up host tbl*	*encap*	*look-up nexthop tbl*	*Total*
SEA-ingress	0.61	0.63	0.67	0.62	2.53
SEA-egress	-	0.63	-	-	0.63
SEA-others	-	-	-	0.67	0.67
ETH	0.63	0.64	-	-	1.27

information. Figures 2.8a and 2.8b show forwarding-table size and control overhead, respectively, measured over one-second intervals. We can see that SEATTLE has much lower control overhead when the systems are first started up. However, SEATTLE's performance advantages do not come from cold-start effects, as it retains lower control overhead even after the system converges. As a side note, the forwarding-table size in Ethernet is not drastically larger than that of SEATTLE in this experiment because we are running on a small four node topology. However, since the topology has ten links (including links to hosts), Ethernet's control overhead remains substantially higher. Additionally, we also investigate performance by injecting host scanning attacks [39] into the real traces we used for evaluation. Figure 2.8b includes the scanning incidences occurred at around 300 and 600 seconds, each of which involves a single host scanning 5000 random destinations that do not exist in the network. In Ethernet, every scanning packet sent to a destination generates a network-wide flood because the destination is not existing, resulting in sudden peaks on it's control overhead curve. In SEATTLE, each scanning packet generates one unicast lookup (i.e., the scanning data packet itself) to a resolver, which then discards the packet.

Fail-over performance: Figure 2.8c shows the effect of switch failure. To evaluate SEATTLE's ability to quickly republish host information, here we intentionally disable caching, induce failures of the resolver switch, and measure throughput of TCP when all

packets are forwarded through the resolver. We set the OSPF hello interval to 1 second, and dead interval to 3 seconds. After the resolver fails, there is some convergence delay before packets are sent via the new resolver. We found that SEATTLE restores connectivity quickly, typically on the order of several hundred milliseconds after the dead interval. This allows TCP to recover within several seconds, as shown in Figure 2.8c-i. We found performance during failures could be improved by having the access switch register hosts with the next switch along the ring in advance, avoiding an additional re-registration delay. When a switch is repaired, there is also a transient outage while routes move back over to the new resolver, as shown in Figure 2.8c-ii. In particular, we were able to improve convergence delay during recoveries by letting switches continue to forward packets through the old resolver for a grace period. In contrast, optimizing Ethernet to attain low (a few sec) convergence delay exposes the network to a high chance of broadcast storms, making it nearly impossible to realize in a large network.

2.8 Summary

Operators today face significant challenges in managing and configuring large networks. Many of these problems arise from the complexity of administering IP networks. Traditional Ethernet is not a viable alternative (except perhaps in small LANs) due to poor scaling and inefficient path selection. We believe that SEATTLE takes an important first step towards solving these problems, by providing scalable self-configuring routing. Our design provides effective protocols to discover neighbors and operates efficiently with its default parameter settings. Hence, in the simplest case, network administrators can ensure reachability without any configuration settings on network devices. However, SEATTLE also provides add-ons for administrators who wish to customize network operation.

Experiments with our initial prototype implementation show that SEATTLE provides efficient routing with low latency, quickly recovers after failures, and handles host mobility and network churn with low control overhead.

Moving forward, we are interested in investigating the deployability of SEATTLE in various other types of networks. We are also interested in ramifications on switch architectures, and how to design switch hardware to efficiently support SEATTLE. Finally, to ensure deployability, this chapter assumes Ethernet stacks at end hosts are not modified. It would be interesting to consider what performance optimizations are possible if end host software can be changed. We intend to answer some of these questions in the following chapters.

Chapter 3

VL2: Scalable and Flexible Data-Center Networks

We introduced in the previous chapter a scalable and efficient plug-and-play network architecture for conventional corporate-campus or university-campus networks. Moving forward, we are interested in what other types of networks can benefit by employing the SEATTLE architecture, or the technical principles used in SEATTLE. At the same time, we are also curious about what other configuration tasks, in addition to those for addressing and routing, can be handled in a plug-and-play fashion. Most notably, we are specifically interested in the configuration activity aiming to ensure networking performance in general, such as traffic engineering. While we deliberately chose a network-based implementation in the previous chapter, we also recognize that modifying end hosts might be recommended, or even unavoidable, in some other types of networks. Motivated by all these questions, in this chapter, we present VL2, an innovative yet practical network architecture that meets the needs of huge data centers.

To be cost effective, data centers must enable any server to be assigned to any ser-

vice. The VL2 network architecture meets the three objectives required for this agility: uniform high capacity between servers, performance isolation between services, and Ethernet layer-2 semantics. VL2 provides (1) flat addressing to allow service instances to be placed anywhere in the network, (2) Valiant Load Balancing (VLB) that uses randomization to spread traffic uniformly across network paths, (3) a new end-system-based address resolution service to achieve layer-2 Ethernet semantics while scaling to large server pools. To build a scalable and reliable network architecture, VL2 leverages proven network technologies that are already available at low cost in high-speed hardware implementations. As a result, VL2 networks can be deployed today, and we have built a working prototype. Our VL2 prototype shuffles 2.7 TB of data among 75 servers in 395 seconds - 93% of the optimal utilization.

We begin in Section 3.1 by giving an overview of cloud-computing data centers and motivating our research. Subsequently in Section 3.2, we give a brief introduction to conventional data-center networks and clarify our technical goals. Then in Section 3.3, we present detailed measurements of traffic and fault data from a large operational cloud service provider. Based on these data, we derive our design and implementation in Section 3.4. In Section 3.5, we evaluate the merits of the VL2 design using measurement, analysis, and experiments. Subsequently, in Section 3.6, we address various operational issues on VL2, from concerns on the effectiveness of VLB in a data center, to the estimated cost of deploying and operating a VL2 network. Finally we summarize related work in Section 3.7 and conclude this chapter in Section 3.8.

3.1 Motivation and Overview

Cloud services are driving the creation of huge data centers, holding tens to hundreds of thousands of servers, that concurrently support a large and dynamic number of distinct services (web apps, email, map-reduce clusters, etc.). The case for cloud service data centers depends on a scale-out design: reliability and performance achieved through large pools of inexpensive resources that can be rapidly reassigned between services as needed. With data centers being built that house over 100,000 servers, at an amortized cost approaching $12 million per month [59], the most desirable property for a data center is *agility* — the ability to assign any server to any service. Anything less inevitably results in stranded resources and wasted money.

Unfortunately, the data center network is not up to the task, falling short in several ways. First, existing architectures do not provide enough capacity between the servers they interconnect. Conventional architectures rely on tree-like network configurations built from high-cost hardware. Due to the cost of the equipment, the capacity between different branches of the tree is typically oversubscribed by factors of 1:5 or more, with paths through the highest levels of the tree oversubscribed by factors of 1:80 to 1:240. This limits communication between servers to the point it fragments the server pool — congestion and computation hot-spots are prevalent even when spare capacity is available elsewhere in the data center. Second, while data centers host multiple services, the network does little to prevent a traffic flood in one service from affecting the other services around it — when one service experiences a traffic flood, it is common for all those sharing the same network subtree to suffer collateral damage. Third, the routing design in conventional networks achieves scale by assigning servers topologically significant IP addresses and dividing servers up among VLANs (i.e., IP subnets). However, this creates

an enormous configuration burden when servers must be reassigned among services further fragmenting the resources of the data center, and the human involvement typically required in these reconfigurations limits the speed of the process.

Cloud-service application owners do not want to be forced to alter their services to work around the structure or limitations of the data center network, as they frequently are doing today. Rather, they want to work with a mental model that all the servers currently assigned to their service, and only those servers, are connected by a single non-blocking Ethernet switch — a *Virtual Layer 2*. Realizing this vision for the data center network concretely translates into building a network that meets the following objectives: First, the network should provide *uniform high capacity* between all servers, meaning the maximum rate of a flow should be limited only by the available capacity on the network interface cards of the sending and receiving servers and there is no need to consider network topology when adding servers to a service. Second, the data center needs *performance isolation* - the traffic of one service should be unaffected by the traffic handled by any other service, just as if each service was connected by a separate physical switch. Third, the network should provide *layer-2 semantics*, just as if the servers were on a LAN. Because LANs have flat addressing, where any IP address can be connected to any port of an Ethernet switch, data center management software can assign any server to any service and configure that server with whatever IP address the service expects. The network configuration of each server should be identical to what it would be if connected via a LAN, and features like link-local broadcast that many legacy applications depend on should work.

3.1.1 Principles and contributions of VL2

In this chapter we design, implement and evaluate VL2, a network architecture for data centers that meets these three objectives and thereby provides agility. The design is motivated by extensive measurements of the traffic in existing production data centers. In crafting VL2, we used four design principles that distinguish our work from other research efforts.

Randomizing to Cope with Volatility: Our measurements show data centers have tremendous volatility in their workload, their traffic, and their failure patterns. Our response is to create large pools of resources and then spread work over them randomly, trading off some performance on the best-cases to improve the worst-case to the average case. We choose a Clos topology for VL2 because of the extensive path diversity it possesses, and we route flows across it using the Valiant Load Balancing technique of indirecting through randomly chosen nodes to obtain the hotspot-free guarantees it offers. Through application of this principle we are able to achieve both the uniform capacity and performance isolation objectives.

Embracing End Systems: The software and operating systems on data centers servers are already extensively modified for use inside the data center, for example, to create hypervisors for virtualization or blob filesystems to store data across servers. Rather than limit ourselves from altering the software on servers, we embrace the opportunity to leverage the programmability they offer. We instead limit ourselves from making any changes to the hardware of the switches or servers, and we require that legacy applications work unmodified. By using software on the servers to work within the limitations of the low-cost switch ASICs currently available, we are able to create a design that can be built and deployed today. For example, we eliminate the scalability problems created by broadcast ARP packets by intercepting ARP requests on the servers and converting

them into lookup requests to a directory system, rather than attempting to control ARPs via software or hardware changes on the switches.

Separating Names from Locations: As many have recognized [17], separating names from locations creates a degree of freedom that can be used to implement new features. We leverage this principle to enable agility in the data center and to improve utilization by reducing fragmentation that the binding between addresses and locations had previously caused. Combining this principle with the previous one enables VL2 to meet the layer-2 semantics objective: allowing developers to assign IP addresses without regard for the network topology and without having to reconfigure their applications or the switches.

Building on proven networking technology: As a pragmatic issue, we have found that reusing network technologies that have robust, mature implementations in network switches both simplifies the design of VL2 as well as increases operator willingness to deploy it. For example, VL2's design reduces the load on the directory system by leveraging the link-state routing protocols already implemented on the switches to hide certain failures from servers. This principle supports all three objectives.

In the remainder of this chapter we will make the following contributions, in roughly this order.

- A first of its kind study of the traffic patterns in production data center. We find that there is tremendous volatility in the traffic, cycling among 50-60 different patterns during a day and spending less than 100 s in each pattern at the 50th percentile.

- We present the design of VL2 and the components that comprise it.

- Every component of VL2 has been built and deployed in an 80-server cluster. Using the cluster, we experimentally validate that VL2 has the properties set out as

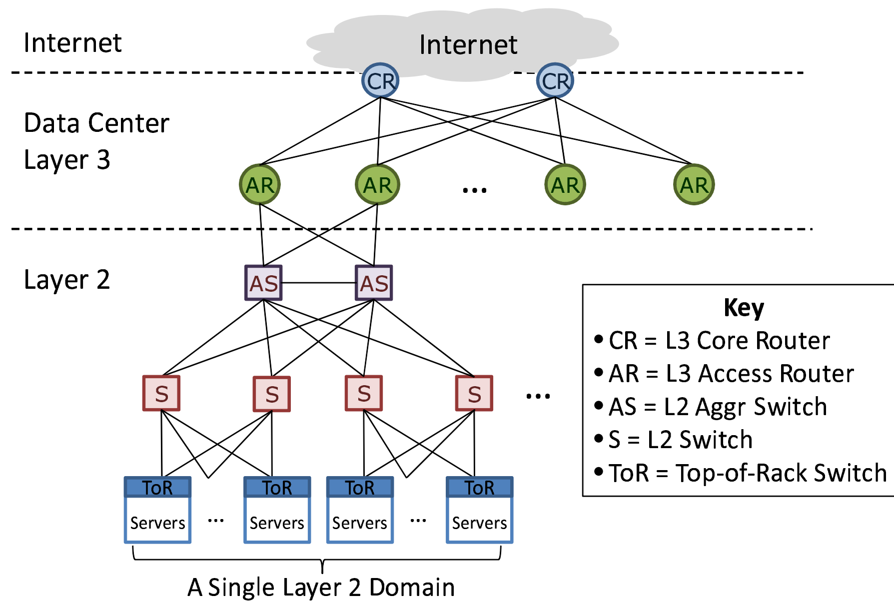


Figure 3.1: The conventional network architecture for data centers

objectives, such as uniform capacity and performance isolation. We also demonstrate the speed of the network, such as its ability to shuffle 2.7 TB of data among 75 servers in 395 s.

- We apply Valiant Load Balancing in a new context, the inter-switch fabric of a data center, and show that flow level traffic splitting achieves almost identical split ratios (within 1% of optimal fairness index) on realistic data center traffic and it smoothes utilization while eliminating persistent congestion.

- We justify the design trade-offs made in VL2, analyze the cost of the network, and describe how it can be cabled for both open floor plan data centers and containers.

3.2 Background

In this section, we first explain the dominant design pattern for data center architecture today. Then we discuss why this architecture is insufficient to serve large cloud-service data centers.

As shown in Figure 3.1, the network is a hierarchy reaching from a layer of servers in racks at the bottom to a layer of core routers at the top. There are typically 20 to 40 servers per rack, each singly connected to a Top of Rack (ToR) switch with 1 Gbps links. ToRs connect to two aggregation switches for redundancy, and these switches aggregate further eventually connecting to access routers.

At the top of the hierarchy, core routers carry traffic between access routers and manage traffic into and out of the data center. All links use Ethernet as a physical-layer protocol, with a mix of copper and fiber cabling. All the switches below each pair of access routers form a single layer-2 domain. The number of servers that can be connected to a single layer-2 domain is typically limited to a few hundred due to Ethernet scaling overheads (packet flooding and ARP broadcasts). To limit these overheads and to isolate different services or logical server groups (e.g., email, search, web front ends, web back ends), servers are partitioned into virtual LANs (VLANs) placed into distinct layer-2 domains.

Unfortunately this conventional design suffers from the following fundamental limitations:

Limited server-to-server capacity: As we go up the hierarchy we are confronted with steep technical and financial barriers in sustaining high bandwidth. Thus, as traffic moves up through the layers of switches and routers, the over-subscription ratio increases rapidly. For example, typically servers have 1:1 over-subscription to other servers in the

same rack; i.e., they can communicate at the full rate (e.g., 1 Gbps) of their interfaces. We found that up-links from ToRs are typically 1:5 to 1:20 oversubscribed (i.e., 1 to 4 Gbps of up-link for 20 servers), and paths through the highest layer of the tree can be 1:240 oversubscribed. This large over-subscription factor severely limits the entire data-center's performance.

Fragmentation of resources: As the cost and performance of communication depends on distance in the hierarchy, the conventional design encourages service planners to cluster servers proximately in the hierarchy. Moreover, spreading service outside a single layer-2 domain frequently requires the onerous task of reconfiguring IP addresses and VLAN trunks, since the IP addresses used by servers are topologically determined by the access routers above them. Collectively, this contributes to the squandering of computing resources across the data center. The consequences are egregious. Even if there is plentiful spare capacity throughout the data center, it is often effectively reserved by a single service (and not shared), so that this service can scale out to proximate servers quickly to respond rapidly to spikes in demand or to failures. In fact, the growing resource needs of one service have forced data center operations to evict other services in the same layer 2 domain, incurring significant cost and disruption.

Poor reliability and utilization: Above the ToR, the basic resilience model is 1:1. For example, if an aggregation switch or access router fails, there must be sufficient remaining idle capacity on the counterpart device to carry the load. This forces each device and link to be run only at most 50% of its maximum utilization. Inside a layer-2 domain, use of the Spanning Tree Protocol means that even when multiple paths between switches exist, only a single one is used. In the layer-3 portion, Equal Cost Multipath (ECMP) is typically used: when multiple paths of the same length are available to a destination, each router uses a hash function to spread flows evenly across the available

next hops. However, the conventional topology offers at most two paths.

3.3 Measurements and Implications

In order to design VL2, we first needed to understand the data center environment in which it would operate. Interviews with architects, developers, and operators led to the objectives described in Section 3.1, but selecting the technical mechanisms on which to build the network requires a quantitative understanding of the traffic matrix (who sends how much data to whom and when) and churn (how often does the state of the network change due to switch/link failures and recoveries, etc.). We analyzed these aspects by studying production data centers of a large cloud service provider, and we use the results to justify our choices in designing VL2 and in generating workloads to stress the VL2 testbed.

Our measurement studies found two key results with implications for the network design. First, the traffic patterns inside a data center are highly divergent (as even over 50 representative traffic matrices only loosely cover the actual traffic matrices seen) and change rapidly and unpredictably. Second, the hierarchical spanning tree topology is intrinsically unreliable — even with a huge effort and expense to increase the reliability of the network devices close to the top of the hierarchy, we still see failures on those devices resulting in significant downtimes.

3.3.1 Data center traffic analysis

Analysis of Netflow and SNMP data from the data centers reveals several macroscopic trends. First, the internal to external traffic volume ratio today is typically about 4:1 (except for CDN applications). Second, data center computation is focused where high

speed access to data on memory or disk is fast and cheap. Although data is distributed across multiple data centers, intense computation and communication on data does not straddle data centers due to the cost of long-haul links. Third, an increasing fraction of the computation in data centers involves back-end computations driving the demands for network bandwidth and storage.

To uncover the exact nature of traffic inside a data center, we instrumented a highly utilized 1,500 node cluster in a data center that supports data mining on petabytes of data. The servers are distributed roughly evenly across 75 top of rack (ToR) switches, which are connected in a hierarchical fashion, as shown in Figure 3.1. We collected socket-level event logs from all machines over a period of two months.

3.3.2 Flow distribution analysis

Distribution of flow size: Figure 3.2 illustrates the nature of flows within the monitored data center. The flow size statistics (marked as '+'s) show that the majority of flows are small (few KB); discussions with developers revealed most of these small flows to be hellos and meta-data requests to the distributed file system. To bring out what is going on with longer flows, we provide a statistic termed *total bytes* (marked as 'o's), by weighting each flow size by its number of bytes. Total bytes tells us, for a random byte, the distribution of the flow size it belongs to. Almost all the bytes in the data center are transported in flows whose lengths vary from about 100 MB to a few GB. The mode at around 100 MB springs from the fact that the distributed file system breaks long files into 100-MB-long chunks.

Similar to Internet flow characteristics [60], we find that there are myriad small flows (mice). On the other hand, as compared with Internet flows, the distribution is simpler and more uniform. The reason is that in data centers, internal flows arise in an engineered

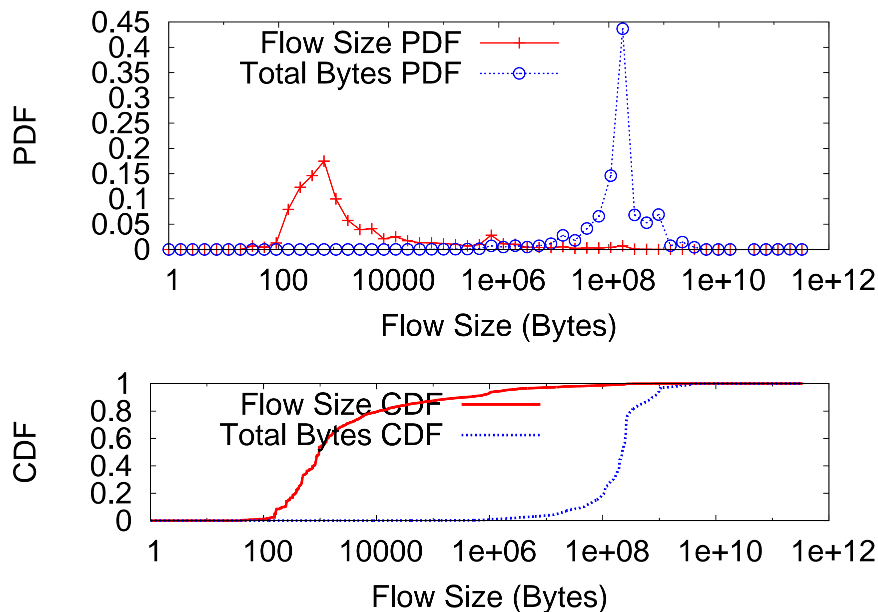


Figure 3.2: Mice are numerous; 99% of flows are smaller than 100 MB. However, more than 90% of bytes are in flows larger than 100 MB.

environment driven by careful design decisions (e.g., the 100-MB-long chunk size is driven by the need to amortize disk-seek times over read times) and by strong incentives to use storage and analytic tools with well understood resilience and performance.

Number of Concurrent Flows: Figure 3.3 shows the probability density function (as a fraction of time) for the number of concurrent flows going in and out of a machine, computed over all 1,500 monitored machines for a representative day's worth of flow data. There are two modes. More than 50% of the time, an average machine has about ten concurrent flows, but for at least 5% of the time an average machine has greater than 80 concurrent flows. We almost never see more than 100 concurrent flows.

We use these statistics on flow size distribution and number of concurrent flows to drive VL2 evaluation in Section 3.5.

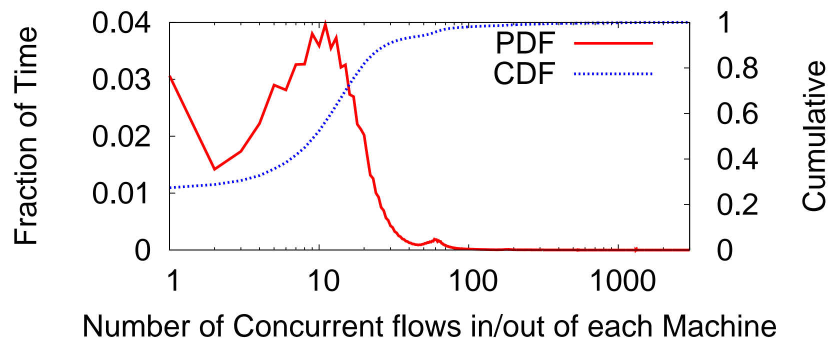


Figure 3.3: Number of concurrent connections has two modes: (1) 10 flows per node more than 50% of the time and (2) 80 flows per node for at least 5% of the time.

3.3.3 Traffic matrix analysis

Distinct traffic patterns: Next, we ask the question: *Is there some degree of regularity in the traffic that might be advantageously exploited through careful measurement and traffic engineering?* If traffic in the DC were to follow a few simple patterns, then a few snapshots of the traffic between all pairs of servers (termed the traffic matrix or TM) would represent these patterns. Further, optimizing on those few representative TMs would yield a routing design that would be capacity-efficient for most traffic.

A technique due to Zhang et al. [61] quantifies the variability in traffic matrices by the approximation error arising when clustering similar TMs. In short, the technique recursively collapses the traffic matrices that are *most similar to each other* into a cluster, where the distance (i.e., similarity) reflects how much traffic needs to be shuffled to make one TM look like the other. We then choose a representative TM for each cluster, such that any routing that can deal with the representative TM performs no worse on every TM in the cluster. Using a single representative TM per cluster yields a fitting error (quantified by the distances between representative TMs and the actual TMs they represent), which quickly decreases as the number of clusters increases but does not dip beyond a certain

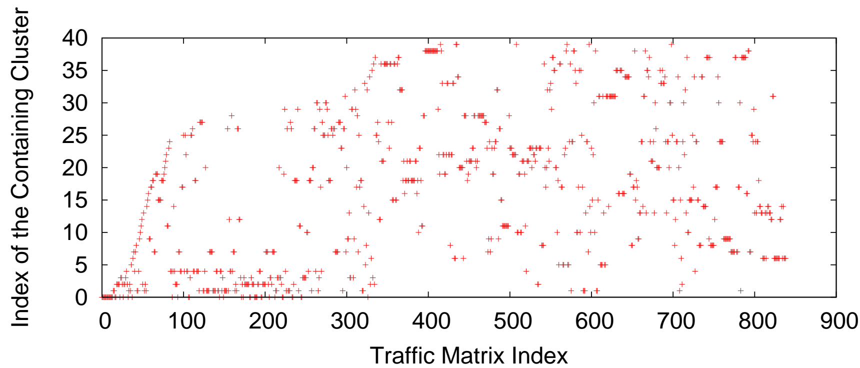


Figure 3.4: Lack of short-term predictability: The cluster to which a traffic matrix belongs, i.e., the type of traffic mix in the TM, changes quickly and randomly.

knee point. Finally, we find the fewest number of clusters that reduces the fitting error below the knee point. The resulting set of clusters and their representative TMs indicates the number of distinct types of traffic matrices present in the set. Surprisingly, we find the number of representative traffic matrices in our data center is quite large — even when approximating with $50-60$ clusters, the fitting error remains high (0.6) and decreases moderately even beyond that point. For comparison, in an ISP network with a comparable TM dimension (AT&T's PoP level topology), only 12 representative traffic matrices yield a good approximation (i.e., fitting error < 0.25) [62].

Instability of traffic patterns: Given the significant variability in traffic, one might wonder whether traffic is predictable in the near term: *Does traffic in the next minute look similar to the traffic now?* Traffic predictability enhances the ability of an operator to engineer network routing as traffic demand changes. To measure the ability to predict the traffic pattern in the network, Figure 3.4 plots the index (which denotes the types of the top-40 traffic matrices, see above) for each 100-sec-long traffic matrix over the day. The figure shows the traffic pattern changes nearly constantly, with no periodicity that could help predict the future. Computing the run lengths (how long the network follows the same matrix), we find the median run length is 1 (i.e., the network changes matrix every

100 s or faster): only 1% of the time does the network retain the same matrix for > 800 s.

The lack of predictability stems largely from fundamental mechanisms used to improve performance of data center applications: randomness. For example, the distributed file system spreads data chunks randomly across servers for load distribution and redundancy. Similarly, the servers assigned to each job are chosen more or less randomly from the pool of available servers.

3.3.4 Failure characteristics

To design VL2 to tolerate failures and churn found in data centers, we collected failure logs over an year from eight data centers in-production comprising hundreds of thousands of servers and hosting 100+ cloud services that serve millions of active users. We analyzed both hardware and software failures using SNMP polling/traps, syslogs, server alarms, and transaction monitoring frameworks for about 36M error events resulting in 300k alarm tickets.

How frequent are network element failures? We define a failure as an event that occurs when a system or component is unable to perform its required function and that lasts over 30 s. We find that as expected, most failures are small in size (e.g., 95% of network device failures involve < 20 devices) while large correlated failures are rare (e.g., 3700 servers fail within 10 minutes). Further, downtimes can be significant: 95% of failures are resolved in 10 min, 98% in < 1 hour, 99.6% in < 1 day, but 0.09% last $>$ 10 days.

What is the pattern of element failure? As discussed in Section 3.2, conventional data center networks apply 1+1 redundancy to improve reliability at higher layers of the spanning tree topology. However, these techniques are still insufficient — we find that in 0.3% of failures, all redundant components in a network device group became un-

available (e.g., the pair of switches that comprise each node in the conventional network (Figure 3.1) or both the uplinks from a switch). In one incident, the failure of a core switch (due to a faulty supervisor card) affected ten million users for about four hours. We found the main causes of these downtimes are network misconfigurations, firmware bugs, and faulty components (e.g., ports). With no obvious way to prevent all failures from the top of the hierarchy, VL2's approach is to broaden the topmost levels of the network so that the impact of failures is muted and performance degrades gracefully, moving from 1+1 redundancy to n+m redundancy.

3.4 Virtual Layer Two Networking

Before describing our design in detail, we briefly revisit our design principles and preview how they will be used in the VL2 design.

Randomizing to Cope with Volatility: The huge divergence and unpredictability of data-center traffic matrices suggest that optimization-based approaches will not be very effective at avoiding congestion. Instead, VL2 uses Valiant Load Balancing (VLB): destination-independent (e.g., random) traffic spreading across multiple intermediate nodes. The theory behind VLB offers provably hot-spot-free performance for *arbitrary traffic matrices*, subject only to ingress/egress capacity bounds [63] as in the hose traffic model [64]. In our context, the ingress/egress constraints correspond to server line-card speeds. Additionally, traffic spreading allows us to offer huge server-to-server capacities at a modest cost because doing so requires only a network with a huge *aggregate* capacity, which can be easily built with a large number of inexpensive devices. We introduce our network topology suited for traffic spreading in Section 3.4.1. The topology offers a huge bisection bandwidth through a large number of equal-cost paths

between servers. Then we present our routing mechanism to randomly spread traffic (more specifically, flows) in Section 3.4.2.

VLB, in theory, ensures a *non-interfering* packet switched network [65] (the counterpart of a non-blocking circuit switched network) as long as *i*) the offered traffic patterns conform to the hose model, and *ii*) traffic spreading ratios are uniform. While our mechanisms to realize VLB do not perfectly meet both these conditions, we show in Section 3.5.1 that our scheme's performance is close to the optimum.

We also study specifically how this loose enforcement of the conditions above affects our system's performance. To meet condition-*i*, we rely on TCP's end-to-end congestion control mechanism to enforce the hose model on offered traffic. Unfortunately, in cloud-computing data centers, non-TCP (e.g., UDP, or any sorts of non-TCP-compliant) traffic co-exists with TCP traffic. We conduct experiments in Section 3.5.2 to see how our design works under such situations. Satisfying condition-*ii* is even harder in practice for two reasons. First, to avoid out-of-order delivery, we spread flows – not packets. Unfortunately flows differ in size. Second, for state-less traffic spreading, we *randomly* – rather than uniformly – associate flows with paths. We conduct experiments in Section 3.5.2 to quantify how this factor manifests itself in practice.

Separating names from locators: To enable agility (such as hosting any service on any server, dynamically growing and shrinking a server pool, and migrating virtual machines), we use an addressing scheme that separates servers' names, termed application-specific addresses (AAs), from their locators, termed location-specific addresses (LAs). VL2 uses a directory system to maintain the mappings between names and locators in a scalable and reliable fashion. A shim layer running in the networking stack on every server, called the VL2 agent, invokes the directory system's resolution service. We evaluate the performance of the directory system in Section 3.5.4.

Embracing End Systems: In a data center, the rich and homogeneous programmability available at end systems provides a mechanism to rapidly realize any new functionality. For example, the VL2 agent enables fine-grained path control by adjusting the randomization used in VLB. In addition, to realize the separation of names and locators, the agent replaces Ethernet's ARP functionality with queries to the VL2 directory system. The directory system itself is also realized on servers, rather than switches, and thus offers flexibility, such as fine-grained, context-aware server access control, or dynamic service re-provisioning.

Building on proven networking technology: While embracing end-system functionality, VL2 also leverages the mature and robust IP routing and forwarding technologies already available in commodity switches. Those include the link-state routing protocol, equal-cost multi-path (ECMP) forwarding, IP anycasting, and IP multicasting. VL2 employs a link-state routing protocol to maintain the switch-level topology, but not to disseminate end hosts' information. This protects switches from needing to learn the huge, frequently-changing host information and thus substantially improves the network's control-plane scalability. Furthermore, through a routing design that utilizes ECMP forwarding along with anycast addresses shared by multiple switches, VL2 spreads traffic over multiple paths and hides network churns from the directory system and end hosts as well.

We next describe each aspect of the VL2 system and how they work together to implement a virtual layer-2 network. These aspects include the network topology, the addressing design, the routing design, and the directory system that manages name-locator mappings.

3.4.1 Scalable oversubscription-free topology

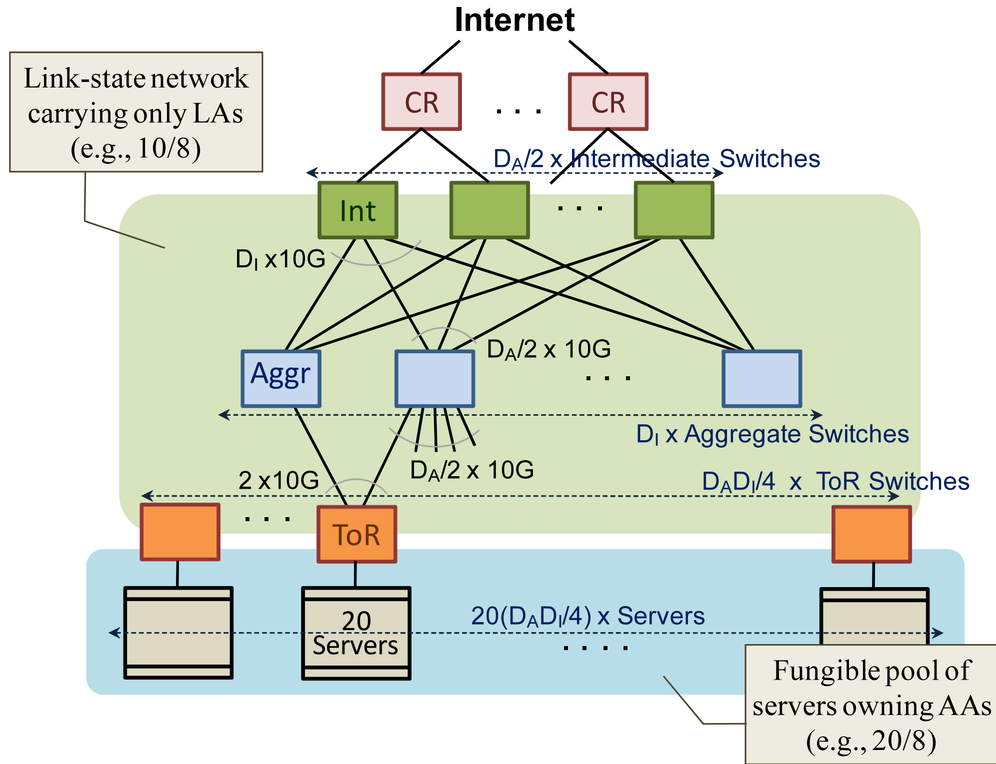


Figure 3.5: Example Clos network between Aggregation and Intermediate switches provides a broad and richly connected backbone well-suited for VLB. Connectivity to the Internet is provided by Core Routers (CR).

As described in Section 3.3, the way conventional data-center networks concentrate traffic into a few devices at the highest levels restricts the total bisection bandwidth and also significantly impacts the network when the devices fail. Instead, we choose a topology driven by our principle to use randomization for coping with traffic volatility. Rather than *scale up* individual network devices with more capacity and features, we *scale out* the devices — build a broad network offering huge *aggregate* capacity using a large number of simple, inexpensive devices, as shown in Figure 3.5. This is an example of a folded Clos network [65] where the links between the Intermediate switches and the Aggrega-

tion switches form a complete bipartite graph. As in the conventional topology, ToRs connect to two Aggregation switches, but the large number of paths between any two Aggregation switches means that if there are n Intermediate switches, the failure of any one of them reduces the bisection bandwidth by only $1/n$ — a desirable property we call *graceful degradation of bandwidth*, evaluated in Section 3.5.3. Further, it is easy and less expensive to build a Clos network for which there is no over-subscription (further discussion on cost is given in Section 3.6). For example, in Figure 3.5, we use D_A-port Aggregation and D_I-port Intermediate switches, and connect these switches such that the capacity between each layer is $D_I D_A/2$ times the link capacity.

The Clos topology is exceptionally well suited for VLB in that by indirectly forwarding traffic through an Intermediate switch at the top tier or "spine" of the network, the network can provide bandwidth guarantees for any traffic matrices subject to the hose model. Meanwhile, routing is extremely simple and resilient on this topology — take a random path up to a random intermediate switch and a random path down to a destination ToR switch.

3.4.2 VL2 routing

This section explains the motion of packets in a VL2 network, and how the topology, routing design, VL2 agent, and directory system combine to virtualize the underlying network fabric and create the illusion for the network layer, and anything above it, that the host is connected to a big, non-interfering data-center-wide layer-2 switch.

Address resolution and packet forwarding

To implement the principle of separating names from locators, VL2 uses two different *IP-address* families. Figure 3.5 illustrates this separation. The network infrastructure oper-

ates using location-specific addresses (LAs); all switches and interfaces are assigned LAs, and switches run an IP-based (i.e., layer-3) link-state routing protocol that disseminates only these LAs. This allows switches to obtain the complete knowledge about the switch-level topology, as well as forward any packets encapsulated with LAs along the shortest paths. On the other hand, applications operate using permanent application-specific addresses (AAs), which remain unaltered no matter how servers' locations change due to virtual-machine migration or re-provisioning. Each AA (server) is associated with an LA, the identifier of the ToR switch to which the servers is connected. The VL2 directory system stores the mapping of AAs to LAs, and this mapping is created when application servers are provisioned to a service and assigned an AA IP address.

The crux of offering the layer-2 semantics is having servers believe they share a single large IP subnet (i.e., the entire AA space) with other servers in the same service, while eliminating the ARP and DHCP scaling bottlenecks that plague large Ethernets.

Packet forwarding: Since AA addresses are not announced into the routing protocols of the network, for a server to receive a packet the packet's source must first encapsulate the packet (Figure 3.6), setting the destination of the outer header to the LA of the ToR under which the destination server (i.e., the destination AA) is located. Once the packet arrives at its destination ToR, the ToR switch decapsulates the packet and delivers it based on the destination AA in the inner header.

Address resolution: Servers in each service are configured to believe that they all belong to the same IP subnet, so when an application sends a packet to an AA for the first time, the networking stack on the host generates a broadcast ARP request for the destination AA. The VL2 agent running in the source host's networking stack intercepts the ARP request and converts it to a unicast query to the VL2 directory system. The directory system answers the query with the LA of the ToR to which packets should be

tunneled.

Inter-service access control by directory service: Servers cannot send packets to an AA if they cannot obtain the LA of the ToR to which they must tunnel packets for that AA. This means the directory service can enforce access-control policies on communication. When handling a lookup request, the directory system knows which server is making the request, the services to which both source and destination belong, and the isolation policy between those services. If the policy is "deny", the directory server simply refuses to provide the LA. An advantage of VL2 is that, when inter-service communication is allowed, packets flow directly from sending server to receiving server, without being detoured to an IP gateway as is required to connect two VLANs in the conventional architecture.

These addressing and forwarding mechanisms were chosen for two main reasons. First, they make it possible to utilize low-cost switches, which often have small routing tables (typically just 16K entries) that can hold only LA routes, without concern for the huge number of AAs. Second they allows the control plane to support agility with very little overhead; the design obviates frequent link-state advertisements to disseminate host-state changes and host/switch reconfiguration.

Random traffic spreading over multiple paths

To offer hot-spot-free performance for arbitrary traffic matrices without any esoteric traffic engineering or optimization, VL2 utilizes two related mechanisms: VLB and ECMP. The goals of both are similar — VLB distributes traffic across multiple intermediate nodes chosen independently of destinations (e.g., randomly), and ECMP across multiple equal-cost paths so as to offer larger capacity. When using these mechanisms, VL2 uses *flows*, rather than packets, as the basic unit of traffic spreading and thus avoids out-or-

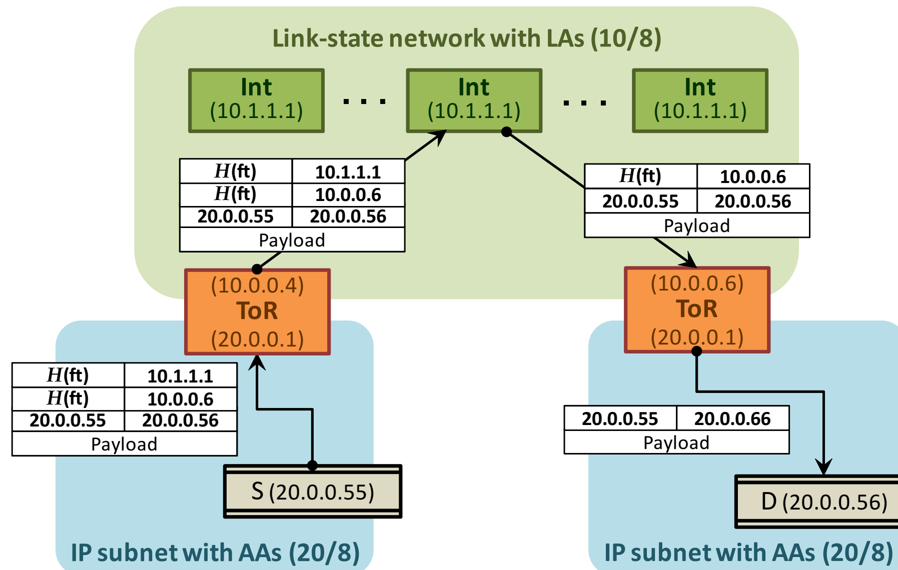


Figure 3.6: VLB in an example VL2 network. Sender S sends packets to destination D via a randomly-chosen intermediate switch using IP-in-IP encapsulation. AAs are from $20/8$ and LAs are from $10/8$. H(`ft`) denotes a hash of the five tuple.

order delivery. As explained below, VLB and ECMP are complementary in that each can be used to overcome limitations in the other.

Realizing the benefits of VLB requires forcing traffic to bounce off a randomly-chosen Intermediate switch. Figure 3.6 illustrates traffic forwarding in an example VL2 network. The VL2 agent on each sender implements this "bouncing" function by encapsulating each packet to an Intermediate switch, wrapped around the header that tunnels the packet to the destination's ToR. Hence the packet is first delivered to one of the Intermediate switches, decapsulated by the switch, delivered to the ToR's LA, decapsulated again, and finally sent to the destination server.

While encapsulating packets to a specific, but randomly chosen, Intermediate switch correctly realizes VLB, it would require updating a potentially huge number (e.g., 100K) of VL2 agents whenever an Intermediate switch's availability changes due to switch/link failures or recoveries. Instead, we assign the same LA address to all Intermediate

switches, and the directory systems returns this *anycast address* to agents as part of the lookup results. Since all Intermediate switches are exactly three hops away from a source host, now ECMP simply takes care of delivering packets encapsulated with the anycast address to any one of the active Intermediate switches. Upon switch or link failures, ECMP will react, eliminating the need to notify agents and ensuring scalability. ECMP mechanisms in modern switches choose next hops in a destination-independent fashion (e.g., based on the hash of five-tuple values), satisfying the VLB semantics.

In practice, however, the use of ECMP leads to two "technical" problems. First, switches today only support up to 16-way ECMP, with 256-way ECMP being released by some vendors this year. If there should be more paths available than ECMP can use, then VL2 defines several anycast addresses, each associated with only as many Intermediate switches as ECMP can accommodate. When an Intermediate switch fails, VL2 reassigns the anycast addresses from that switch to other Intermediate switches so that all anycast addresses remain live and servers can remain unaware of the network churn. Second, inexpensive commodity switches cannot correctly retrieve the five-tuple values when a packet is encapsulated with multiple IP headers. As a solution, the agent at the source computes a hash of the five-tuple values and writes that value into a header field the switch does use in making an ECMP-forwarding decision. VL2 uses the source IP address field, and the type-of-service (ToS) is another option.

A final issue for both ECMP and VLB is the chance that uneven flow sizes and random spreading decisions will cause transient congestion on some links. Our evaluation did not find this to be a problem on data center workloads (Section 3.5.2), but should it occur, the sender can change the path its flows take through the network by altering the value of the fields that ECMP uses to select a next-hop. Initial results show the VL2 agent can detect and deal with such situations with simple mechanisms, such as re-hashing the large flows

periodically or when TCP detects a severe congestion event (e.g., a full window loss or Explicit Congestion Notification).

Backwards-compatibility

To ensure complete layer-2 semantics, the routing and forwarding solutions must also be backwards compatible and transparent to the existing data-center applications. This section describes how a VL2 networks handle external traffic (from and to the Internet), as well as general layer-2 broadcast traffic.

Interaction with hosts in the Internet: 20% of the traffic handled in our cloud-computing data centers is to or from the Internet, so the network must be able to handle these large volumes. Since VL2 employs a layer-3 routing fabric to implement a virtual layer-2 network, the external traffic can directly flow across the high-speed silicon of the switches that make up VL2, without being forced through gateway servers to have their headers rewritten, as required by some designs (e.g., Monsoon [10]).

Servers that need to be directly reachable from the Internet (e.g., front-end web servers) are assigned two addresses: an LA in addition to the AA used for intra-data-center communication with back-end servers. This LA is drawn from a pool that is announced via BGP and is externally reachable. Traffic from the Internet can then directly reach the server, and traffic from the server to external destinations will be routed toward the core routers while being spread across the available links and core routers by ECMP.

Handling Broadcast: VL2 provides layer-2 semantics to applications for backwards compatibility, and that includes supporting broadcast and multicast. VL2's approach is to eliminate the most common sources of broadcast completely, such as ARP and DHCP. ARP is handled by the mechanism described above, and DHCP messages are intercepted at the ToR using conventional DHCP relay agents and unicast forwarded to

DHCP servers. To handle other general layer-2 broadcast traffic, every service is assigned an IP multicast address, and all broadcast traffic in that service is handled via IP multicast using the service-specific multicast address. The VL2 agent rate-limits broadcast traffic to prevent storms.

3.4.3 Maintaining host information using VL2 directory system

The VL2 directory system is a scalable, reliable and high performance store designed for data center workloads. It provides two key functionalities: (1) *lookups* and *updates* for AA-to-LA mappings, and (2) a reactive cache update mechanism that supports latency-sensitive operations, such as live virtual machine migration.

Characterizing requirements

We expect the lookup workload for the directory system to be frequent and bursty. As discussed in Section 3.3.1, servers can communicate with up to hundreds of other servers in a short time period with each flow generating a lookup for an AA-to-LA mapping. For updates, the workload is driven by failures and server startup events. As discussed in Section 3.3.4, most failures are small in size and large correlated failures are rare.

Performance requirements: The bursty nature of workload implies that lookups require high throughput and low response time to quickly establish a large number of connections. Since lookups are a replacement for ARP, their response time should match that of ARP, i.e., tens of milliseconds. For updates, however, the key requirement is reliability, and response time is less critical. Further, since updates are typically scheduled ahead of time, high throughput can be achieved by batching updates.

Consistency requirements: In a conventional L2 network, ARP provides eventual consistency due to ARP timeout. In addition, a host can announce its arrival by issuing

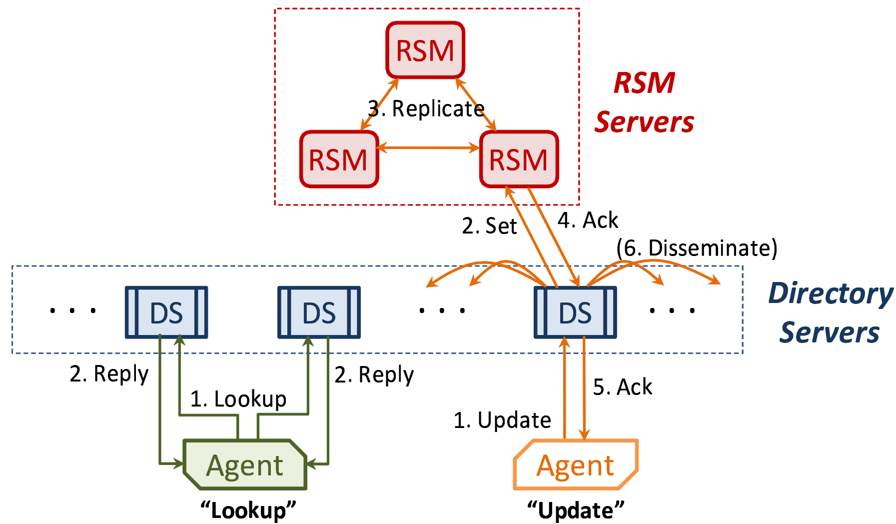


Figure 3.7: VL2 Directory System Architecture

a gratuitous ARP [66]. As an extreme example, consider live virtual machine (VM) migration in a VL2 network. VM migration requires fast update of stale mappings (AA-to-LA) as its primary goal is to preserve on-going communications across location changes. These considerations imply that weak or eventual consistency of AA-to-LA mappings is acceptable as long as we provide a reliable update mechanism.

Directory-system design

Our observations that the performance requirements and workload patterns of lookups differ significantly from those of updates led us to a two-tiered directory system architecture shown in Figure 3.7. Our design consists of (1) a modest number (50-100 servers for 100K servers) of read-optimized, replicated directory servers that cache AA-to-LA mappings and that communicate with VL2 agents, and (2) a small number (5-10 servers) of write-optimized, asynchronous replicated state machine (RSM) servers offering a strongly consistent, reliable store of AA-to-LA mappings. The directory servers

ensure low latency, high throughput, and high availability for a high lookup rate. Meanwhile, the RSM servers ensure strong consistency and durability, using the Paxos [67] consensus algorithm, for a modest rate of updates.

Each directory server caches all the AA-to-LA mappings stored at the RSM servers and independently replies to lookups from agents using the cached state. Since strong consistency is not a requirement, a directory server lazily synchronizes its local mappings with the RSM on a regular basis (e.g., every 30 secs). To achieve high availability and low latency at the same time, an agent sends a lookup to k (two in our prototype) randomly-chosen directory servers. If multiple replies are received, the agent simply chooses the fastest reply and stores it in its cache.

Directory servers also handle updates from network provisioning systems. For consistency and durability, an update is sent to only one randomly-chosen directory server and is always written through to the RSM servers. Specifically, on an update, a directory server first forwards the update to the RSM. The RSM reliably replicates the update to every RSM server and then replies with an acknowledgment to the directory server, which in turn forwards the acknowledgment back to the originating client. As an optimization to enhance consistency, the directory server can optionally disseminate the acknowledged updates to a small number of other directory servers. If the originating client does not receive an acknowledgment within a timeout (e.g., 2s), the client sends the same update to another directory server, trading response time for reliability and availability.

Ensuring eventual consistency: Since AA-to-LA mappings are cached at directory servers and at VL2 agents' cache, an update can lead to inconsistency. To resolve inconsistency without wasting server and network resources, our design employs a reactive cache-update mechanism to ensure both scalability and performance at the same time. The cache-update protocol leverages a key observation: a stale host mapping needs to

be corrected only when that mapping is used to deliver traffic. Specifically, when a stale mapping is used, some packets arrive at a stale LA – a ToR which does not host the destination server anymore. ToRs forward such non-deliverable packets to a directory server, triggering the directory server to selectively correct the stale mapping in the source server's cache via unicast.

3.5 Evaluation

In this section we evaluate VL2 using a prototype running on an 80 server testbed and commodity switches. Our goals are two-fold. First, we want to show that VL2 can be built from components available today and Second, our implementation meets the objectives described in Section 3.1.

The testbed is built using a Clos network topology, similar to Figure 3.5, consisting of 3 Intermediate switches, 3 Aggregation switches and 4 ToRs. The Aggregation and Intermediate switches have 24 10Gbps Ethernet ports, of which 6 ports are used on the Aggregation switches and 3 ports on the Intermediate switches. The ToRs switches have 4 10Gbps ports and 24 1Gbps ports. Each ToR is connected to two Aggregation switches via 10Gbps links, and to 20 servers via 1Gbps links. Internally, the switches use commodity merchant silicon ASICs: Broadcom ASICs 56820 and 56514. To enable detailed analysis of the TCP behavior seen during experiments, the servers' kernels are instrumented to log TCP extended statistics [68] (e.g., congestion window (cwnd) and smoothed RTT) after each socket buffer is sent (typically 128KB in our experiments). This logging does not affect *goodput*, i.e., useful information delivered per second to the application layer.

We first investigate VL2's ability high uniform network bandwidth between servers,

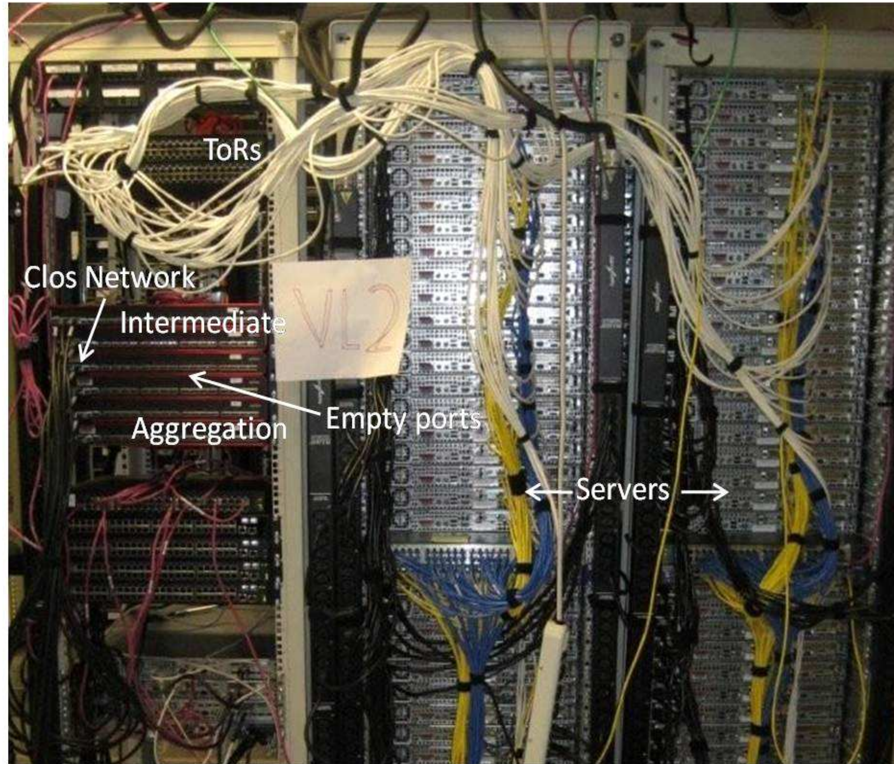


Figure 3.8: VL2 testbed comprising 80 servers and 10 switches.

then analyze performance isolation and fairness between traffic flows, measure convergence after link failures, and finally, quantify address resolution performance. Overall, our evaluation shows that VL2 provides an effective substrate for a scalable data center network: VL2 achieves (1) 93% optimal network capacity, (2) a TCP fairness index of 0.995, (3) graceful degradation under failures with fast reconvergence, and (4) handles 50K lookups/sec under 10ms for fast address resolution.

3.5.1 VL2 Uniform high capacity

A central objective of VL2 is uniform high capacity between any two servers in the data center. How closely does the performance and efficiency of a VL2 network match that of a Layer 2 switch with 1:1 over-subscription?

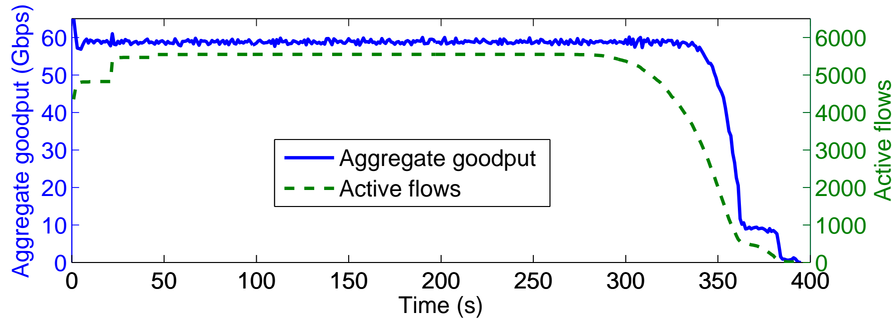


Figure 3.9: Aggregate goodput during a 2.7TB shuffle among 75 servers.

To answer this question, we consider an all-to-all *data shuffle* stress test: all servers simultaneously initiate TCP transfers to all other servers. This data shuffle pattern arises in large scale sorts, merges and join operations in the data center. We chose this test because, in our interactions with application developers, we learned that many use such operations with caution, because the operations are highly expensive in today's data center network. However, data shuffles are required and if data shuffles can be efficiently supported, it could have large impact on the overall algorithmic and data storage strategy.

We create an all-to-all data shuffle traffic matrix involving 75 servers. Each of 75 servers must deliver 500MB of data to each of the 74 other servers - a shuffle of 2.7 TB from memory to memory. [1]

Figure 3.9 shows how the sum of the goodput over all flows varies with time during a typical run of the 2.7 TB data shuffle. All data is carried over TCP connections, all of which attempt to connect beginning at time 0. VL2 completes the shuffle in 395 s. During the run, the sustained utilization of the core links in the Clos network is about 86%. For the majority of the run, VL2 achieves an aggregate goodput of 58.8 Gbps. The goodput is very evenly divided among the flows for most of the run, with a fairness index

[1] We chose 500MB files rather than 100MB files (the most common flow size seen in our measurements) to extend the period during which all 5,550 flows are sending simultaneously – some flows start late due to connection timeout on first attempt.

between the flows of 0.995 [69] where 1.0 indicates perfect fairness (mean goodput per flow 11.4 Mbps, standard deviation 0.75 Mbps). This goodput is more than an order of magnitude improvement over our existing network constructed using traditional design.

How close is VL2 to the maximum achievable throughput in this environment? To answer this question, we compute the goodput efficiency for this data transfer. The goodput efficiency of the network for any interval of time is defined as the ratio of the sent goodput summed over all interfaces divided by the sum of the interface capacities. An efficiency of 1.0 would mean that all the capacity on all the interfaces is entirely used carrying useful bytes from the time the first flow starts to when the last flow ends.

To calculate the goodput efficiency, two sources of inefficiency must be accounted for. First, to achieve a performance efficiency of 1.0, the server network interface cards must be completely full-duplex: able to both send and receive 1 Gbps simultaneously. Measurements show our interfaces are able to support a sustained rate of 1.8 Gbps (summing the sent and received capacity), introducing an inefficiency of 1.8/2.0 = 10%. The sources of this inefficiency include TCP ack overhead and artifacts of operating system and device driver implementations. In addition, there is the overhead of packet headers. In the VL2 design, packet headers (including the encapsulation headers) account for 6% inefficiency for standard Ethernet MTU of 1,500 Bytes. Therefore, our current testbed has an intrinsic inefficiency of 16% resulting in maximum achievable goodput for our testbed of 84%.

Taking the above into consideration the VL2 network achieves an efficiency of (75 * .84) / 58.8 = 93%. This combined with the fairness index of .995 demonstrates that VL2 promises to achieve uniform high bandwidth across all servers in the data center.

3.5.2 Performance isolation

One of the primary objectives of VL2 is *agility*, which we define as the ability to assign any server, anywhere in the data center to any service (3.1). Achieving agility critically depends on providing sufficient performance isolation between services so that if one service comes under attack or a bug causes its servers to spray packets, it does not adversely impact the performance of other services.

The promise of performance isolation in VL2 rests on the mathematics of Valiant Load Balancing — that any traffic matrix that obeys the hose model is sprayed evenly across the network (through randomization) to prevent any persistent hot spots. Rather than have VL2 perform admission control or rate shaping to ensure the traffic offered to the network conforms to the hose model, we instead rely on TCP to ensure that each flow offered to the network is rate-limited to its fair share of its bottleneck. Further, VL2 relies on ECMP to split traffic in equal ratios to intermediate switches. Because ECMP does flow-level splitting, coexisting elephant and mice flows might get split unevenly at smaller time scales.

Thus, the two key questions for performance isolation are — whether TCP reacts sufficiently quickly to control the offered rate of flows, and whether our implementation of Valiant Load Balancing splits traffic evenly across the network. In the following, we describe experiments that evaluate these two aspects of VL2's design.

Does TCP obey the hose model?

In this experiment, we add two services to the network. The first service has 18 servers allocated to it and each server starts a single TCP transfer to one other server at time 0 and these flows last for the duration of the experiment. The second service starts with one server at 60 seconds and a new server is assigned to it every 2 seconds for a total of 19

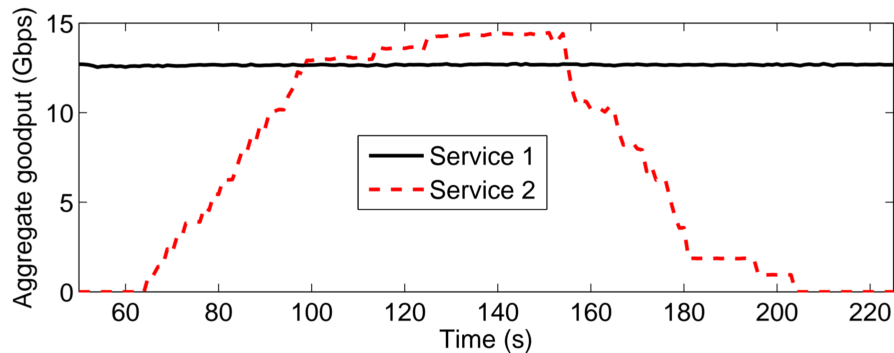


Figure 3.10: Aggregate goodput of two services with servers intermingled on the TORs. Service one's goodput is unaffected as service two ramps traffic up and down.

servers. Every server in service two starts an 8GB transfer over TCP as soon as it starts up. Both the services' servers are intermingled among the 4 TORs to demonstrate agile assignment of servers.

Figure 3.10 shows the aggregate goodput of both services as a function of time. As seen in the figure, there is no perceptible change to the aggregate goodput of service one as the flows in service two start up or complete, demonstrating performance isolation when the traffic consists of large long-lived flows. Through extended TCP statistics, we inspected the congestion window size (cwnd) of service one's TCP flows, and found that the flows fluctuate around their fair share momentarily due to service two's activity but then stabilize quickly.

We would expect that a service sending unlimited rate UDP traffic might violate the hose model and hence performance isolation. We do not observe such UDP traffic in our data centers, although, techniques such as STCP to make UDP "TCP friendly" are well known if needed [70]. However, large numbers of short TCP connections (mice), which *are* common in DCs (Section 3.3), have the potential to cause problems similar to UDP as each flow can transmit small bursts of packets as it begins slow start. Intuitively, the bursts of small connections threaten to reduce goodput of long lived flows, as the mice

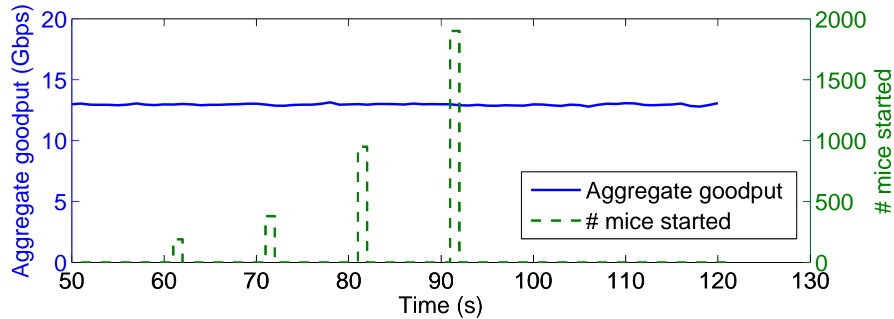


Figure 3.11: Aggregate goodput of service one as service two creates bursts containing successively more short TCP connections.

may capture an unfairly large fraction of the small buffers in VL2's switches.

To evaluate this aspect, we conduct a second experiment with service one sending long lived TCP flows, as in experiment one. Servers in service two create bursts of short TCP connections (1 to 20 KB), each burst containing progressively more connections.

Figure 3.11 shows the aggregate goodput of the service one's flows along with the total number of TCP connections by service two versus time. Again, service one's goodput is unaffected by service two's activity. We inspected the cwnd of service one's TCP flows and found only brief fluctuations due to service two's activity.

The above two experiments demonstrate TCP's natural enforcement of the hose model. Even though service one's flows could have taken more bandwidth in the network, TCP limited them to their receivers' interface capacity, thereby leaving spare capacity in the network for service two to ramp up and down without impacting service one's goodput.

VLB fairness

To evaluate the effectiveness VL2's implementation of Valiant Load Balancing in splitting traffic evenly across the network, we created an experiment on our 75-node testbed with

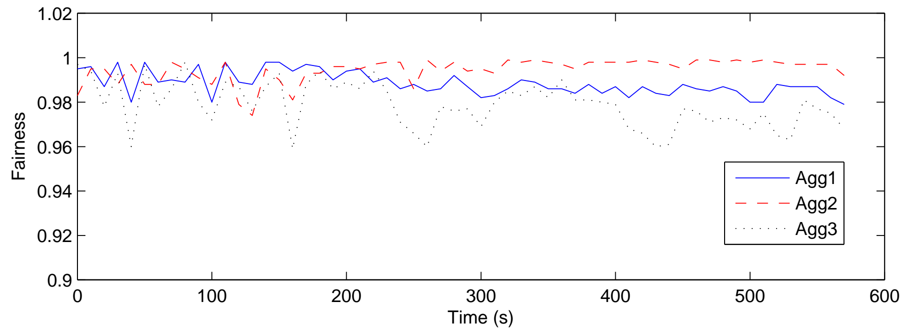


Figure 3.12: Fairness measures how evenly flows are split to intermediate switches from aggregation switches. Average utilization is for links between Aggregation and Intermediate switches.

traffic characteristics extracted from the DC workload of Section 3.3. Each server initially picks a value from the distribution of number of concurrent flows and maintains this number of flows throughout the experiment. At the start, or after a flow completes, it picks flow rate(s) from the associated distribution and starts the flow(s). Because of flow aggregation happening at the Aggregation switches, it is sufficient to check the split ratios at each Aggregation switch to each Intermediate switch. We do this by collecting SNMP counters at 10 second intervals for all links from Aggregation to Intermediate switches.

In Figure 3.12, for each Aggregation switch, we plot Jain's fairness index [69] for the traffic to Intermediate switches as a time series. The average utilization of links was between 10% and 20%. As shown in the figure, the average VLB split ratio fairness index is greater than .98 for all Aggregation switches over the duration of this experiment. We get such high fairness because there are enough flows at the Aggregation switches that randomization benefits from statistical multiplexing. This evaluation validates that our implementation of VLB is an effective mechanism for preventing hot spots in a data center network.

In summary, the even splitting of traffic in VLB, when combined with TCP's confor-

mance of the hose model, provides sufficient performance isolation to achieve agility.

3.5.3 Convergence after link failures

As discussed in Section 3.3, interface flaps account for 28% of network failures. Our discussions with network engineers revealed that many of these are due to software and hardware bugs, which manage to slip through processes for testing and hardening the system. VL2 mitigates the threat of such bugs by simplifying the network control and data plane and relying on existing, mature OSPF routing implementation. In this section, we evaluate VL2's response when a link or a switch failure does happen, which could be the result of the routing protocol or the network management process converting a link flap to a link failure.

We begin an all-to-all data shuffle and then disconnect links between Intermediate and Aggregation switches until only one Intermediate switch remains connected and the removal of one additional link would partition the network. According to our study of failures, this type of mass link failure has never occurred in our data centers, but we use it as an illustrative stress test.

Figure 3.13 shows a time series of the aggregate goodput achieved by the flows in the data shuffle, with the times at which links were disconnected and then reconnected marked by vertical lines. The figure shows OSPF is re-converging quickly (sub-second) after each failure. Both Valiant Load Balancing and ECMP work as expected, and the maximum capacity of the network gracefully degrades. OSPF timers delay restoration after links are reconnected, but restoration does not interfere with traffic and the aggregate goodput returns to its previous level.

This experiment also demonstrates the behavior of VL2 when the network is structurally oversubscribed (meaning the Clos network has less capacity than the capacity of

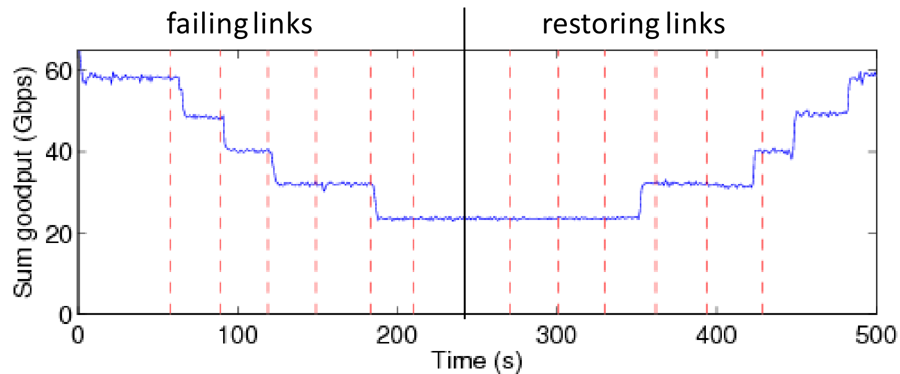
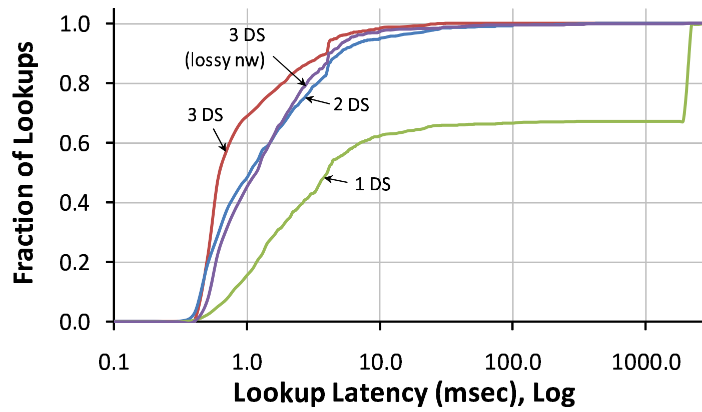


Figure 3.13: Aggregate goodput as all links to switches Intermediate1 and Intermediate2 are unplugged in succession and then reconnected in succession. Approximate times of link manipulation marked with vertical lines. Network re-converges in < 1s after each failure and demonstrates graceful degradation.

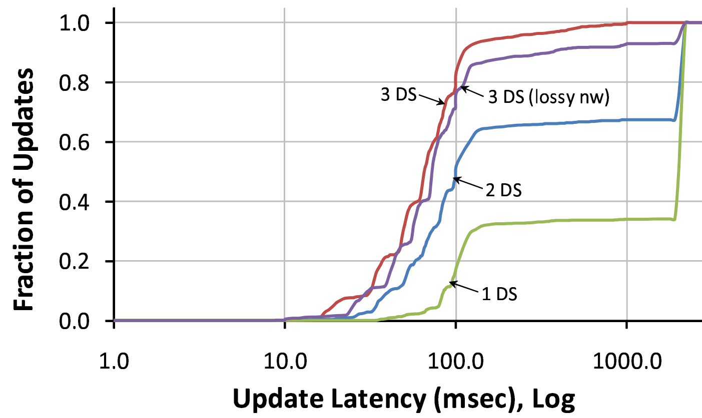
the links from the ToRs). For the over-subscription ratios between 1:1 and 3:1 created during this experiment, the VL2 continues to carry the all-to-all traffic at roughly 90% of maximum efficiency, indicating that the traffic spreading in VL2 is making full use of the available capacity.

3.5.4 Directory-system performance

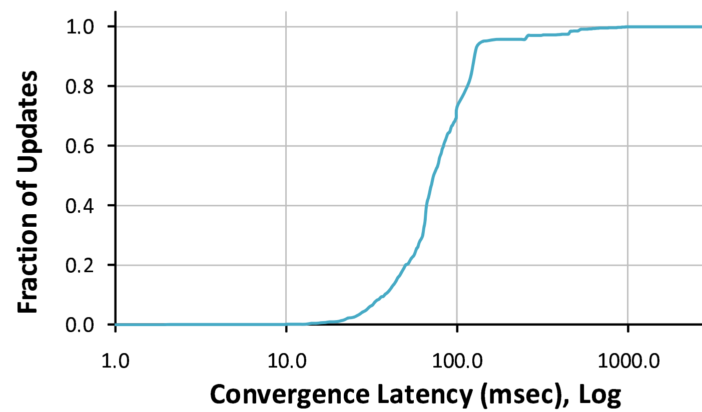
Finally, we evaluate the performance of the VL2 directory system which provides the equivalent semantics of ARP in layer 2. We perform this evaluation through macro- and micro-benchmark experiments on the directory system. We run our prototype on up to 50 machines: 3-5 RSM nodes, 3-7 directory server nodes, and the remaining nodes emulating multiple instances of VL2 agents generating lookups and updates. In all experiments, the system is configured so that an agent sends a lookup request to two directory servers chosen at random and accepts the first response. An update request is sent to a directory server chosen at random. The response timeout for lookups and updates is set to two seconds to measure the worst-case latency. To stress test the directory system, the VL2



(a)



(b)



(c)

Figure 3.14: The directory system provides high throughput and fast response time for lookups and updates

agent instances generate lookups and updates following a bursty random process, emulating storms of lookups and updates. Each directory server retrieves all the mappings (100K) from the RSM every 30 seconds.

Our evaluation supports four main conclusions. First, the directory system provides high throughput and fast response time for lookups; three directory servers can handle 50K lookups/sec with latency under 10ms (99th percentile latency). Second, the directory system can handle updates at rates significantly higher than expected churn rate in typical environments: three directory servers can handle 12K updates/sec within 600ms (99th percentile latency). Third, our system is incrementally scalable; each directory server increases the processing rate by about 17K for lookups and 4K for updates. Finally, the directory system is robust to component (directory or RSM servers) failures and offers high availability under network churns.

Throughput: In the first micro-benchmark, we vary the lookup and update rate and observe the response latencies (1st, 50th and 99th percentile). We observe that a directory system with three directory servers handles 50K lookups/sec within 10ms which we set as the maximum acceptable latency for an "ARP request". Up to 40K lookups/sec, the system offers a median response time of $<$ 1ms. Updates, however, are more expensive, as they require executing a consensus protocol [67] to ensure that all RSM replicas are mutually consistent. Since high throughput is more important than latency for updates, we batch updates over a short time interval (e.g., 50ms). We find that three directory servers backed by three RSM servers can handle 12K updates/sec within 600ms and about 17K updates/sec within 1s.

Scalability: To understand the incremental scalability of the directory system, we measured the maximum lookup rates (ensuring sub-10ms latency for 99% requests) with 3, 5, and 7 directory servers. The result confirmed that the maximum lookup rates increases

linearly with the number of directory servers (with each server offering a capacity of 17K lookups/sec). Based on this result, we estimate the worst case number of directory servers needed for a 100K server data center. Using the concurrent flow measurements (Figure 3.3), we use the baseline of 10 correspondents per server in a 100s window. In the worst case, all 100K servers may perform 10 simultaneous lookups at the same time resulting in a million simultaneous lookups per second. As noted above, each directory server can handle about 17K lookups/sec under 10ms 99th percentile. Therefore, handling this worst case will require a modest-sized directory system of about 60 servers (0.06% of the entire servers).

Resilience and availability: We examine the effect of directory server failures on latency. We vary the number of directory servers while keeping the workload constant at a rate of 32K lookups/sec and 4K updates/sec (a higher load than expected for three directory servers). In Figure 3.14(a), the lines for one directory server show that it can handle 60% of the lookup load (19K) within 10ms. The spike at two seconds is due to the timeout value of 2s in our prototype. The entire load is handled by two directory servers, demonstrating the system's fault tolerance. Additionally, the lossy network curve shows the latency of three directory servers under severe (10%) packet losses between directory servers and clients (either requests or responses), showing the system ensures availability under network churns.

For updates, however, the performance impact of the number of directory servers is higher than updates because each update is sent to a single directory server to ensure correctness. Figure 3.14(b) shows that failures of individual directory servers do not collapse the entire system's processing capacity to handle updates. The step pattern on the curves is due to a batching of updates (occurring every 50ms). We also find that the primary RSM server's failure leads to only about 4s delay for updates until a new primary

is elected, while a primary's recovery or non-primary's failures/recoveries do not affect the update latency at all.

Fast reconvergence and robustness: Finally, we evaluate the convergence latency of updates i.e., the time between when an update occurs until a lookup response reflects that update. As described in Section 3.4.3, we minimize convergence latency by having each directory server pro-actively send its committed updates to other directory servers. Figure 3.14(c) shows that the convergence latency is within 100ms for 70% updates and 99% of updates have convergence latency within 530 ms.

3.6 Discussion

In this section, we address several remaining concerns about the VL2 architecture, including whether other traffic engineering mechanisms might be better suited to the DC than Valiant Load Balancing; the cost of a VL2 network; and the cost and viability of cabling VL2.

Optimality of VLB: As noted in Section 3.4, VLB uses randomization to cope with volatility, potentially sacrificing some performance for a best-case traffic pattern by turning all traffic patterns (including both best-case and worst-case) into the average case. This performance loss will manifest itself as the utilization of some links being higher than they would under a more optimal traffic engineering system. To quantify the increase in link utilization VLB will suffer, we compare VLB's maximum link utilization with that achieved by other routing strategies on a full day's traffic matrices (TMs) from the DC traffic data reported in Section 3.3.1.

We first compare to *adaptive routing*, which routes each TM separately so as to minimize the maximum link utilization for that TM — essentially upper-bounding the best

performance that real-time adaptive traffic engineering could achieve. Second, we compare to *best oblivious routing* over all TMs so as to minimize the maximum link utilization. (Note that VLB is just one among many oblivious routing strategies.) For adaptive and best oblivious routing, the routings are computed using respective linear programs in Cplex. The overall utilization for a link in all schemes is computed as the maximum utilization over all routed TMs.

The results show that for the median utilization link in each scheme, VLB performs about the same as the other two schemes. For the most heavily loaded link in each scheme, VLB's link capacity usage is about 17% higher than that of the other two schemes. Thus, evaluations on actual data center workloads show that the simplicity and universality of VLB costs relatively little capacity when compared to much more complex traffic engineering schemes.

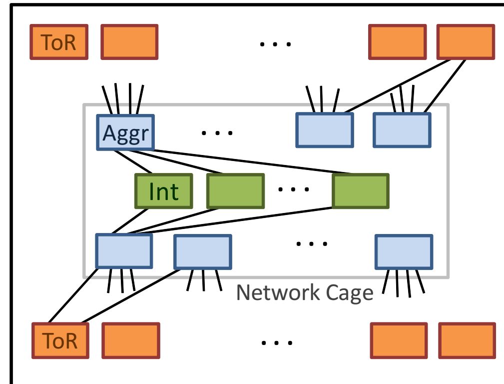
Cost and Scale: With the range of low-cost commodity devices currently available, the VL2 topology can scale to create networks with no over-subscription between all the servers of even the largest data centers. For example, switches with 144 ports ($D = 144$) are available today for $75K, enabling a network that connects 100K servers using the topology in Figure 3.5 and up to 200K servers using a slight variation. Using switches with $D = 24$ ports (which are available today for $10K each), we can connect about 3K servers. Comparing the cost of a VL2 network for 35K servers with a conventional one found in one of our data centers shows that a VL2 network with no over-subscription can be built for the same cost as the current network that has 1:240 over-subscription. Building a conventional network with no over-subscription would cost roughly 14x the cost of a equivalent VL2 network with no over-subscription. We find the same ballpark factor of 14-20 cost difference holds across a range of over-subscription ratios from 1:1 to 1:23. (We use street prices for switches in both architectures and leave out ToR and cabling

costs.) There is some savings to be had by building an oversubscribed VL2 network, as a VL2 network with 1:23 over-subscription costs 70% less than a non-oversubscribed VL2 network, but the savings is probably not worth the loss in performance.

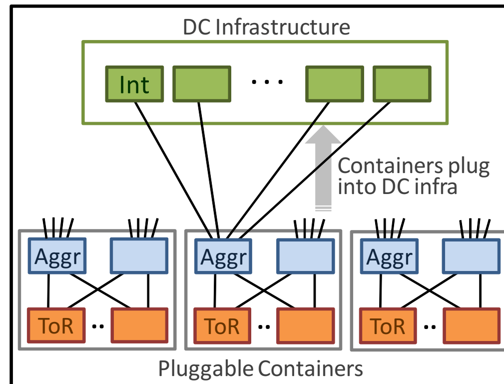
Cabling and Deployment: A major concern for every network topology is the ability to realize the cabling required. The VL2 topology in Figure 3.5 maps easily to a number of common and anticipated deployment scenarios. Consider the use of 10G SFP+ fiber optic cables for all network links (the cost of each cable is roughly $190, dominated by the cost of the SFP+ optical transceiver at each end). Given that the 10G end-ports of a link cost about $500 each, we estimate the cabling cost to be $190/1000 = 19\%$ of total system cost. Actual calculations show that for each of these deployment scenarios, the *total cabling cost is 12% of the network equipment cost* (including ToR costs).

Layout Designs: Figure 3.15(a) shows a layout of a VL2 network into a conventional open floor plan data center. The ToRs and server racks surround a central "network cage" and connect using copper or fiber cables, just as they do today in conventional data center layouts. The aggregation and intermediate switches are laid out in close proximity inside the network cage, allowing use of copper cables for their interconnection (copper cable is lower cost, thicker, and has low distance reach vs. fiber). The number of cables inside the network cage can be reduced by a factor of 4 (and their total cost by a factor of about 2) by bundling together four 10G links into a single cable using the QSFP+ standard.

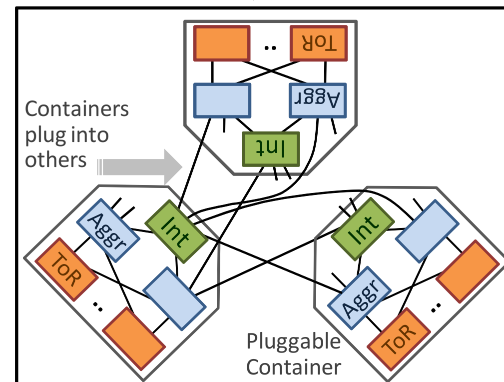
Modularization of the data center via containerization is a recent trend [71]. Figure 3.15(b) shows how the server racks, ToRs, and pairs of Aggregation switches can be packaged into containers that plug into the Intermediate switches, the latter forming part of the DC infrastructure. This design requires bringing one cable bundle from each container to the data center spine. As the next logical step, we can move the Intermediate switches into the containers themselves to realize a fully "infrastructure-less" and "containerized"



(a)



(b)



(c)

Figure 3.15: Three layouts for VL2: (a) Conventional DC floor layout, (b) Container-based layout with intermediate switches part of DC infrastructure, and (c) Fully "containerized" layout. (External connectivity, servers racks, and complete wiring not shown.)

data center – this layout is shown in Figure 3.15(c). This design requires running one cable bundle between each pair of containers C1 and C2 – the bundle will carry links that connect the aggregation switches in C1 to the intermediate switch in C2 and vice-versa.

3.7 Related Work

Commercial Networks: Data Center Ethernet (DCE) [72] by Cisco and other switch manufacturers share the goal of increasing network capacity through multi-path with VL2. However, these industry efforts are primarily focused on consolidation of IP and storage area network (SAN) traffic, which is rare in cloud-service data centers, as they are built on distributed file systems. Due to the requirement to support loss-less traffic, their switches need much bigger buffers (tens of MBs) than commodity Ethernet switches do (tens of KBs), hence driving their cost higher.

Scalable routing: Locator/ID Separation Protocol [17] from IETF proposes "map-and-encap" as a key principle to achieve scalability and mobility in Internet routing. VL2's control-plane takes a similar approach (i.e., demand-driven host-information resolution and caching) but adapted to data center environment and implemented on end hosts.

SEATTLE [73] proposes a distributed host-information resolution system running on switches to enhance Ethernet's scalability. VL2 takes an end host based approach to this problem, which allows its solution to be implemented today, independent of the switches being used. Furthermore, SEATTLE does not provide scalable data plane primitives, such as multi-path, which are critical for scalability and increasing utilization of network resources.

Data-center network designs: DCell [74] proposes a highly-dense interconnection

network for data centers by incorporating end systems with multiple network interfaces into traffic forwarding and routing. VL2 shares a similar philosophy of leveraging design options available at servers, however, uses servers only to control the way traffic is routed but not for forwarding. Furthermore, DCell incurs significant cabling complexity that may limit incremental growth.

Fat-tree [75] and Monsoon [10] also propose building a data center network using commodity switches and a Clos topology. Monsoon is designed on top of layer 2 and reinvents fault-tolerant routing mechanisms already established at layer 3. Fat-tree relies on a customized routing primitive that does not yet exist in commodity switches. VL2, in contrast, achieves hot-spot-free routing and scalable layer-2 semantics using forwarding primitives available today and minor, application-compatible modifications to host operating systems. Further, our experiments using traffic patterns from a real data center show that random flow spreading leads to a network utilization fairly close to the optimum, obviating the need for a complicated and expensive optimization scheme suggested by Fat-tree.

Valiant Load Balancing: Valiant introduced VLB as a randomized scheme for communication among parallel processors interconnected in a hypercube topology [65]. Among its recent applications, VLB has been used inside the switching fabric of a packet switch [76]. VLB has also been proposed, with modifications and generalizations [63, 62], for oblivious routing of variable traffic on the Internet under the hose traffic model [64].

3.8 Summary

The key to creating economical data centers is enabling agility – the ability to assign any server to any service – but the network in today's data centers directly inhibits agility. We argue that to enable agility, the network should meet three objectives: uniform high capacity, performance isolation, and layer-2 semantics.

In this chapter we present the VL2 network architecture that meets these objectives. It gives each service the illusion that all its servers are plugged into a single layer 2 switch, regardless of where the servers are actually connected in the topology. VL2 provides high throughput, hot-spot free routing, and performance isolation through Valiant Load Balancing on a Clos topology. The VL2 directory system achieves high throughput and fast response times, and only requires about 60 nodes for a data center of 100K servers. VL2 embraces the opportunity to customize the server operating system in the data center which allows us to build VL2 by leveraging robust networking technologies working today.

We implemented all components of VL2 and created a working prototype interconnecting 80 servers using commodity switches. Experiments with two data-center services showed that churns (e.g., dynamic re-provisioning of servers, change of link capacity, and micro-bursts of flows) have little impact on TCP goodput. Using the flow statistics measured in an operational 1,500-server cluster to drive the workload, we validated that VL2's implementation of Valiant Load Balancing splits flows evenly and VL2 achieves high TCP fairness. Our prototype network shuffles 2.7 TB of data across 75 servers in 395 seconds, achieving an efficiency of 93% with a TCP fairness index of 0.995 showing that VL2 delivers high uniform capacity.

Chapter 4

Relaying: A Scalable Routing Architecture for Virtual Private Networks

In Chapters 2 and 3, we proposed network architectures that require novel functions to be implemented in routers, switches, or end hosts. While being helpful on a mid- to long-term basis, such an approach offers little help to network administrators who want to turn an existing operational network into a scalable and efficient self-configuring one *today*.

Addressing this kind of problem requires different approaches. First, it is critical to ensure that a new solution (i.e., network architecture) can be built with router/switch/end-host functions available today. Second, more importantly, a substantial amount of effort has to be spent on facilitating the deployment and operation of the new solution. This encompasses various tasks, including offering mechanisms that ensure backwards-compatibility (with end hosts and neighboring networks), devising algorithms that help administrators to make optimal operational decisions, building tools that implement such

algorithms, and evaluating the algorithms and tools with real data from a target network. Taking virtual private networks as an example, this chapter addresses all these questions on *immediately-available*, scalable, and self-configuring network architectures.

Enterprise customers are increasingly adopting VPN service that offers direct any-to-any reachability among the customer sites via a provider network. Unfortunately this direct reachability model makes the service provider's routing tables grow very large as the number of VPNs and the number of routes per customer increase. As a result, router memory in the provider's network has become a key bottleneck in provisioning new customers.

This chapter proposes *Relaying*, a scalable VPN routing architecture that the provider can implement simply by modifying the configuration of routers in the provider network, without requiring changes to the router hardware and software. Relaying substantially reduces the memory footprint of VPNs by choosing a small number of hub routers in each VPN that maintain full reachability information, and by allowing non-hub routers to reach other routers through a hub.

Deploying Relaying in practice, however, poses a challenging optimization problem that involves minimizing router memory usage by having as few hubs as possible, while limiting the additional latency due to indirect delivery via a hub. We first investigate the fundamental tension between the two objectives and then develop algorithms to solve the optimization problem by leveraging some unique properties of VPNs, such as sparsity of traffic matrices and spatial locality of customer sites. Extensive evaluations using real traffic matrices, routing configurations, and VPN topologies demonstrate that Relaying is very promising and can reduce routing-table usage by up to 90%, while increasing the additional distances traversed by traffic by only a few hundred miles, and the backbone bandwidth usage by less than 10%.

We begin this chapter in Section 4.1 by giving an overview of the conventional VPN architecture, as well as motivating Relaying. Then in Section 4.2, we offer a brief introduction to the problem background and desirable properties that any solutions for the problem should offer. Subsequently we present our measurement results and motivate Relaying in Section 4.3. Then we describe our baseline Relaying scheme in Section 4.4 and explore the broad solution space with the baseline Relaying scheme in Section 4.5. In Sections 4.6 and 4.7, we formulate problems of finding practical Relaying configuration, propose algorithms to solve the problems, and evaluate the algorithms. Finally, we discuss the implementation and deployment issues in Section 4.8, briefly review related work in Section 4.9, and conclude the chapter in Section 4.10.

4.1 Motivation and Overview

VPN service allows enterprise customers to interconnect their sites via dedicated, secure tunnels that are established over a provider network. Among various VPN architectures, layer-3 MPLS VPN [77] offers direct any-to-any reachability among all sites of a customer without requiring the customer itself to maintain full-mesh tunnels between each pair of sites. This benefit of any-to-any reachability makes each customer VPN highly scalable and cost-efficient, leading to the growth of the MPLS VPN service at an extremely rapid pace. According to the market researcher IDC, the MPLS VPN market was worth $16.4 billion in 2006 and is still growing fast [78]. By 2010, it is expected that nearly all medium-sized and large businesses in the United States will have MPLS VPNs in place.

The any-any reachability model of MPLS VPNs imposes a heavy cost on the providers' router memory resources. Each provider edge (PE) router in a VPN *provider*

network (see, e.g., Figure 4.1a) is connected to one or more different customer sites, and each customer edge (CE) router in a site announces its own address blocks (i.e., routes) to the PE router it is connected to. To enable *direct* any-to-any reachability over the provider network, for each VPN, each PE router advertises all routes it received from the CE routers that are directly connected to it, to all other PEs in the same VPN. Then, the other PEs keep those routes in their VPN routing tables for later packet delivery. Thus, the VPN routing tables in PE routers grow very fast as the number of customers (i.e., VPNs) and the number of routes per customer increase. As a result, router memory space required for storing VPN routing tables has become a key bottleneck in provisioning new customers.

We give a simple example to illustrate how critical the memory management problem is. Consider a PE with a network interface card with OC-12 (622 Mbps) bandwidth that can be channelized into 336 T1 (1.544 Mbps) ports - this is a very common interface card configuration for PEs. This interface can serve up to 336 different customer sites. It is not unusual that a large company has hundreds or even thousands of sites. For instance, a large convenience store chain in the U.S. has 7,200 stores. Now, suppose the PE in question serves one retail store of the chain via one of the T1 ports. Since each of the 7,200 stores announces at least two routes (one for the site, and the other for the link connecting the site and the backbone), that single PE has to maintain at least 14,400 routes just to maintain any-any connectivity to all sites in this customer's VPN. On the other hand, a router's network interface has a limited amount of memory that is specifically designed for fast address look-up. Today's state-of-the-art interface card can store at most 1 million routes, and a mid-level interface card popularly used for PEs can hold at most 200 to $300K$ routes. Obviously, using 7.2% $(14,400/200K)$ of the total memory for a single site that accounts for only at most 0.3% of the total capacity (1 out of 336

T1 ports) leads to very low utilization; having only 14 customers that are similar to the convenience store can use up the entire interface card memory, while 322 other ports are still available. Even if interface cards with larger amounts of memory become available in the future, since the port-density of interfaces also grows, this resource utilization gap remains.

4.1.1 Relaying: Don't keep it if you don't need it

Fortunately, in reality, every customer site typically does not communicate with every other site in the VPN. This is driven by a number of factors including *i*) most networking applications today are predominantly client-server applications, and the servers (e.g., database, mail, file servers, etc.) are almost always centrally located at a few customer sites, and *ii*) enterprise communications typically follow corporate structures and hierarchies. In fact, a measurement study based on traffic volumes in a large VPN provider's backbone shows that traffic matrices (i.e., matrices of traffic volumes between each pair of PEs) in VPNs are typically very sparse, and have a clear hub-and-spoke communication pattern [79, 80]. We also observed similar patterns by analyzing our own flow-level traffic traces. Hence, PE routers nowadays install more routes than they actually need, perhaps much more than they frequently need.

This sparse communication behavior of VPNs motivates a router-memory saving approach that *installs only a smaller number of routes* at a PE, while still *maintains any-to-any connectivity between customer sites*. In this chapter, we propose *Relaying*, a scalable VPN routing architecture. Relaying substantially reduces the memory footprint of VPNs by *selecting a small number of hub PEs that maintain full reachability information, and by allowing non-hub PEs to reach other routers only through the hubs.* To be useful in practice, however, Relaying needs to satisfy the following requirements:

- **Bounded penalty:** The performance penalty associated with indirect delivery (i.e., detouring through a hub) should be properly restricted, so that the service quality perceived by customers does not get noticeably deteriorated and that the workload posed on the provider's network does not significantly increase either. Specifically, both *i*) additional latency between communicating pairs of PEs, and *ii*) the increase of load on the provider network should be insignificant on average and be strictly bounded within the values specified in SLAs (Service Level Agreements) in the worst case.

- **Deployability:** The solution should be immediately deployable, work in the context of existing routing protocols, require no changes to router hardware and software, and be transparent to customers.

To bound the performance penalty and to reduce the memory footprint of routing tables at the same time, we need to choose a *small* number of hub PEs out of all PEs, where the hub PEs originate or receive *most* traffic within the VPN. Specifically, we formulate this requirement as the following optimization problem. For each VPN whose traffic matrices, topology, and indirection constraints (e.g., maximum additional latency, or total latency) are given, *select as small a number of hubs as possible, such that the total number of routes installed at all PEs is minimized, while the constraints on indirect routing are not violated.* Note that, unlike conventional routing studies that typically limit overall stretch (i.e., the ratio between the length of the actual path used for delivery and the length of the shortest path), we instead bound the additional (or total) latency of each *individual* path. This is because an overall stretch is often not quite useful in directly quantifying the performance impact on applications along each path, and hence hard to be derived from SLAs. Moreover, most applications are rather tolerant to the

small increase of latency, but the perceived quality of those applications drastically drops beyond a certain threshold which can be very well specified by an absolute maximum latency value, rather than a ratio (i.e., stretch).

To solve this optimization problem, we first explore the fundamental trade-off relationship between the number of hubs and the cost due to the relayed delivery. Then, we propose algorithms that can strictly limit the increase of individual path lengths and can reduce the number of hubs at the same time. Our algorithms exploit some unique properties of VPNs, such as sparse traffic matrices and spatial locality of customer sites. We then perform extensive evaluations using real traffic matrices, route advertisement configuration data, and network topologies of hundreds of VPNs at a large provider. The results show that Relaying can reduce routing table sizes by up to 90%. The cost for this large saving is the increase of individual communication's unidirectional latency only by at most 2 to 3 ms (i.e., the increase of each path's length by up to a few hundred miles), and the increase of backbone resource utilization by less than 10%. Moreover, even when we assume a full any-to-any conversation pattern in each VPN, rather than the sparse patterns that are monitored during a measurement period, our algorithms can save more than 60% of memory for moderate penalties.

This chapter makes four contributions: *i*) We propose Relaying, a new routing architecture for MPLS VPNs that substantially reduces memory usage of routing tables; *ii*) we formulate an optimization problem of determining a hub set, and assigning hubs to the remaining PEs in a VPN; *iii*) we develop practical algorithms to solve the hub selection problem; and *iv*) we extensively evaluate the proposed architecture and algorithms with real traffic traces, routing configuration, and topologies from hundreds of operational VPNs.

4.2 Background

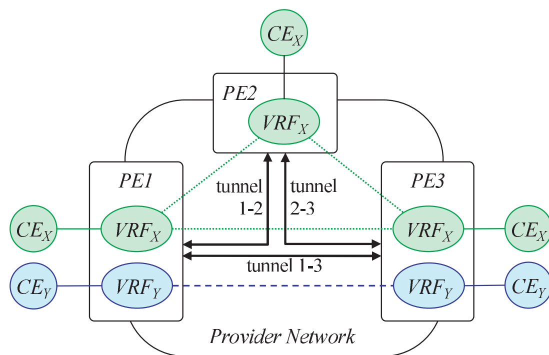
In this section, we first provide some background on MPLS VPN and then introduce terms we use in later sections. We also describe what properties a memory saving solution for VPNs should possess. Finally we briefly justify our Relaying architecture.

4.2.1 How MPLS VPN works

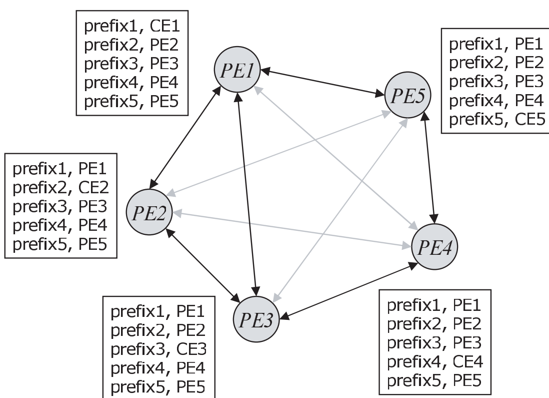
Layer 3 MPLS VPN is a technology that creates virtual networks on top of a shared MPLS backbone. As shown in Figure 4.1a, a PE can be connected to multiple Customer Edge (CE) routers of different customers. Isolating traffic among different customers is achieved by having distinct Virtual Routing and Forwarding (VRF) instances in PEs. Thus, one can conceptually view a VRF as a virtual PE that is specific to a VPN [1]. Given a VPN, each VRF locally populates its VPN routing table either with statically configured routes (i.e., subnets) pointing to incident CE routers, or with routes that are learned from the incident CE routers via BGP [81]. Then, these local routes are propagated to other VRFs in the same VPN via Multi-Protocol Border Gateway Protocol (MP-BGP) [82]. Once routes are disseminated correctly, each VRF learns all customer routes of the VPN. Then, packets are directly forwarded from a source to a destination VRF through a label-switched path (i.e., tunnel). PEs in a region are physically located at a single POP (Point of Presence) that houses all communication facilities in the region.

Figure 4.1b illustrates an example VPN provisioned over five PEs. Each PE's routing table is shown as a box by the PE. We assume that PE i is connected to CE i which announces prefix i. PE i advertises prefix i to the other PEs via BGP, ensuring reachability to CE i. To offer the direct any-to-any reachability, each PE stores every route advertised

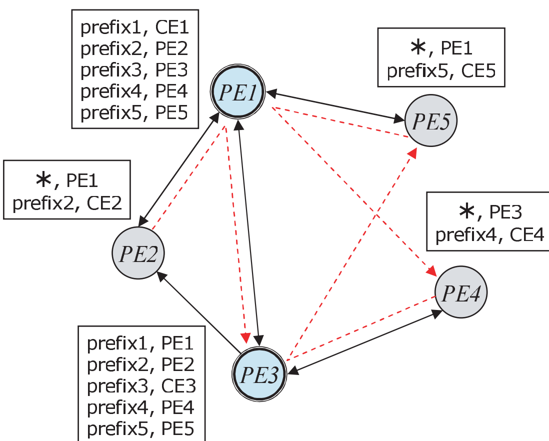
[1] We also use "PE" to denote "VRF" when we specifically discuss about a single VPN.



(a)



(b)



(c)

Figure 4.1: (a) MPLS VPN service with three PEs; two customer VPNs (X, Y) exist, (b) Direct reachability, (c) Reachability under Relaying.

by the other PEs in its local VRF table. In this example, thus, each PE keeps five route entries, leading to 25 entries in total across all PEs. The arrows illustrate a traffic matrix. Black arrows represent active communications between pairs of PEs that are monitored during a measurement period, whereas gray ones denote inactive communications.

Specifically, our Relaying architecture aims to reduce the size of a FIB (Forwarding Information Base), a data structure storing route entries. A FIB is also called a forwarding table and is optimized for fast look-up for high speed packet forwarding. Due to performance and scalability reasons, routers are usually built with several FIBs each of which is located in a very fast memory on a line card (i.e., network interface card). Unfortunately, the size of a line-card memory is limited, and increasing its size is usually very hard due to various constraints, such as packet forwarding rate, power consumption, heat dissipation, spatial restriction, etc. For example, some line-card models use a special hardware, such as TCAM (Ternary Content Addressable Memory) or SRAM [83], which is much more expensive and hard to be built in a larger size than regular DRAMs are. Even if a larger line-card memory was available, upgrading all line cards in the network with the large memory may be extremely costly. In MPLS VPN, a VRF is a virtual FIB specific to a VPN and resides in a line-card memory along with other VRFs configured on the same card. Beside the VRFs, line-card memory also stores packet filtering rules, counters for measurement, and sometimes the routes from the public Internet as well, which collectively make the FIB-size problem even more challenging.

4.2.2 Desirable properties of a solution

To ensure usefulness, a router memory saving architecture for VPNs should satisfy the following requirements.

1. **Immediately deployable:** Routing table growth is an imminent problem to providers; a solution should make use of router functions (either in software or hardware) and routing protocols that are available today.

2. **Simple to implement:** A solution must be easy to design and implement. For management simplicity, configuring the solution should be intuitive as well.

3. **Transparent to customers:** A solution should not require modifications to customer routers.

We deliberately choose Relaying as a solution because it satisfies these requirements. Relaying satisfies goal *1* because the provider can implement Relaying only via router configuration changes (see Section 4.8 for details). It also meets goal *3* since a hub maintains full reachability, allowing spoke-to-spoke traffic to be directly handled at a hub without forwarding it to a customer site that is directly connected to the hub. Ensuring goal *2*, however, shapes some design principles of Relaying which we will discuss in the following sections. Here we briefly summarize those details and justify them.

Relaying classifies PEs into just two groups (hubs and spokes) and applies a simple "all-or-one" table construction policy to the groups, where hubs maintain "all" routes in the VPN, and spokes store only "one" default route to a hub (the details are in Section 4.4). Although we could save more memory by allowing each hub to store a disjoint fraction of the entire route set, such an approach inevitably increases complexity because the scheme requires a consistency protocol among PEs.

For the same reason, we do not consider incorporating cache-based optimizations. When using route caching, each spoke PE can store a small fraction of routes (in addition to the default route, or without the default route) that might be useful for future packet delivery. Thus any conversation whose destination is found in the cache does not take

an indirect path. Despite this benefit, a route caching scheme is very hard to implement because we have to modify routers, violating goal *1*. Specifically, we need to design and implement *i*) a resolution protocol to handle cache misses, and *ii*) a caching architecture (e.g., route eviction mechanism) running in router interface cards. Apart from the implementation issues, the route caching mechanism itself is generally much harder to correctly configure than Relaying is, violating goal *2*. For example, to actually reduce memory usage, we need to fix a route cache's size. However, a fixed-sized cache is vulnerable to a sudden increase of the number of popular routes due to the changes in the customer side or malicious attempts to poison a cache (e.g., scanning). To avoid thrashing in these cases we have to either dynamically adjust cache size, or have to allow some slackness to buffer the churns; neither is satisfactory because the former introduces complexity, and the latter lowers memory saving effect.

Goal *2* also leads us to another important design decision, namely individual optimization of VPNs. That is, in our Relaying model, a set of Relaying configuration (i.e., the set of hubs) for a VPN does not depend on other VPNs. Thus, for example, we do not select a VRF in a PE as a hub at the expense of making other VRFs in the same PE spokes, neither do we choose a VRF as a spoke to make other VRFs in the same PE hubs. This design decision is critical because VPNs are dynamic. If we allowed the dependency among different VPNs, having a new VPN customer or deleting an existing customer might alter the Relaying configuration of other VPNs, leading to a large re-configuration overhead. Moreover, this independence condition also allows network administrators to customize each VPN differently by applying different optimization parameters to different VPNs.

4.3 Understanding VPNs

In this section, we first briefly describes the data set used throughout the chapter. Then we present our measurement results from a large set of operational VPNs. By analyzing the results, we identify key observations that motivate Relaying.

4.3.1 Data sources

VPN configuration, VRF tables, and network topology: We use configuration and topology information of a large VPN service provider in the U.S. which has, at least, hundreds of customers. VPNs vary in size and in geographical coverage; smaller ones are provisioned over a few PEs, whereas larger ones span over hundreds of PEs. The largest VPN installs more than $20,000$ routes in each of its VRFs. Specifically, from this VPN configuration set, we obtain the list of PEs with which each VPN is provisioned, and the list of prefixes each VRF advertises to other VRFs. We also obtain the list of routes installed in each VRF under the existing routing configuration. From the topology, we obtain the location of each PE and POP, the list of PEs in each POP, and inter-POP distances.

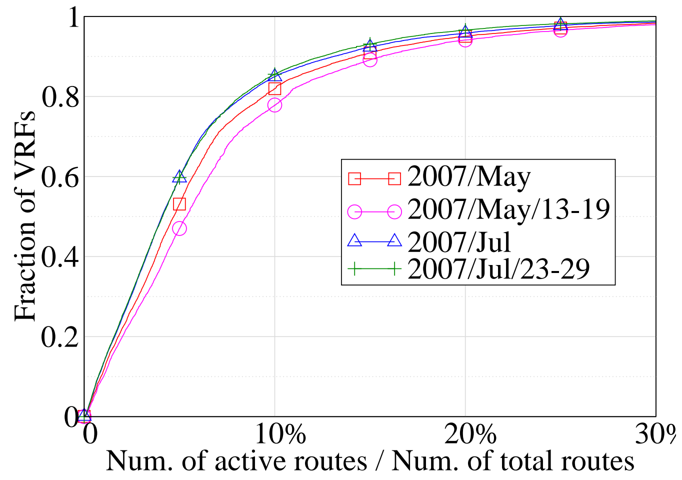
Traffic matrices: We use traffic matrices each of which describes PE-to-PE traffic volumes in a VPN. These matrices are generated by analyzing real traffic traces captured in the provider backbone over a certain (usually longer than a week) period. The traffic traces are obtained by monitoring the links of PEs facing the core routers in the backbone using Netflow [84]. Thus, the source PE of the flow is obvious, while the destination is also available from the tunnel end point information in flow records. Unless otherwise specified, the evaluation results shown in the following sections are based on a week-long traffic measurements obtained in May, 2007.

4.3.2 Properties enabling memory saving

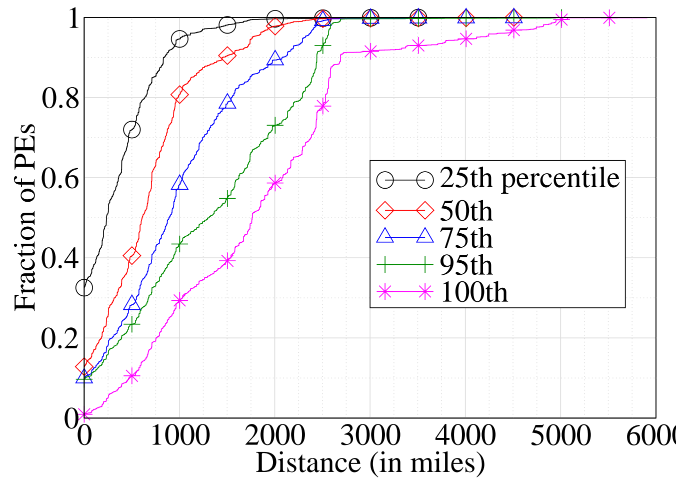
Through the analysis of the measurement results, we make the following observations about the MPLS VPNs. These properties allow us to employ Relaying to reduce routing tables.

Sparse traffic matrices: A significant fraction of VPNs exhibit *hub-and-spoke* traffic patterns, where a majority of PEs (i.e., spokes) communicate *mostly* with a small number of highly popular PEs (i.e., hubs). Figure 4.2a shows the distributions of the number of active prefixes (i.e., destination address blocks that are actually used during a measurement period) divided by the number of total prefixes in a VRF. We measure the distributions during four different measurement periods, ranging from a week to a month. The curves show that, for most VRFs, the set of active prefixes is much smaller than the set of total prefixes. Across all measurement periods, roughly 80% (94%) of VRFs use only 10% (20%) of the total prefixes stored. The figure also confirms that the sets of active prefixes are stable over different measurement periods. By processing these results, we found out that the actual amount of memory required to store the active route set is only 3.9% of the total amount. Thus, if there was an ideal memory saving scheme that precisely maintain only those prefixes that are used during the measurement period, such a scheme would reduce memory usage by 96.1%. This number sets a rough guideline for our Relaying mechanism.

Spatial locality of customer sites: Sites in the same VPN tend to be clustered geographically. Figure 4.2b shows the distributions of the distance from a VRF to its i-th percentile closest VRF. For example, the 25th percentile curve shows that 80% of VRFs have 25% of the other VRFs in the same VPN within 630 miles. According to the 50-th percentile curve, most (81%) VRFs have at least half of the other VRFs in the same VPN within



(a)



(b)

Figure 4.2: (a) CDFs of the proportion of active prefixes in a VRF, (b) CDFs of the distance to the i-th percentile closest VRF

$1,000$ miles. Thus, a single hub can serve a large number of nearby PEs, leading to the decrease of additional distances due to Relaying.

PE's Freedom to selectively install routes: A PE can choose not to store and advertise every specific route of a VPN to a CE as long as it maintains reachability to all the other sites (e.g. via a default route). Indeed, this does not affect a CE's reachability to all other sites because a CE's only way to reach other sites is via its adjacent PE(s) of the same (and sole) VPN backbone. Furthermore, this CE does not have to propagate all the routes to other downstream customers. However, a CE might still be connected to multiple PEs for load-balancing or backup purpose. In that case, the same load-balancing or backup goals are still achieved if all the adjacent PEs are selected as hubs or all are selected as spokes at the same time so that all the PEs announce the same set of routes to the CE. Note that this property does not hold for the routers participating in the Internet routing, where it is common for customers to be multi-homed to multiple providers or to be transit providers themselves.

4.4 Overview of Relaying

The key properties of VPN introduced in the previous section collectively form a foundation for Relaying. In this section, we first define the Relaying architecture, and then introduce detailed variations of the Relaying mechanism.

4.4.1 Relaying through hubs

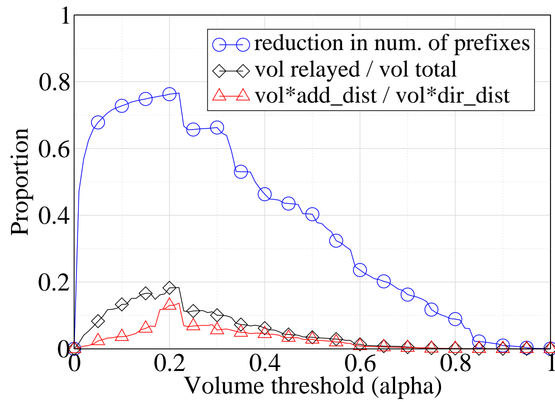
In Relaying, PEs are categorized into two different groups: *hubs* and *spokes*. A hub PE maintains full reachability information, whereas a spoke PE maintains the reachability for the customer sites that are directly attached to it and a *single default route* pointing to

one of the hub PEs. When a spoke needs to deliver packets destined to non-local sites, the spoke forwards the packets to its hub. Since every hub maintains full reachability, the hub that received the relayed packets can then directly deliver them to correct destinations. Multi-hop delivery across hubs is not required because every hub maintains the same routing table.

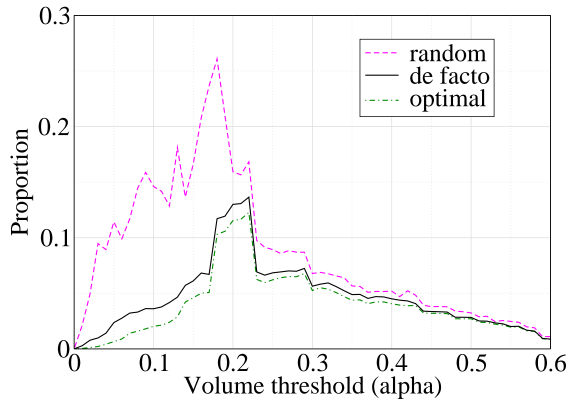
This mechanism is illustrated in Figure 4.1c. Assuming the traffic pattern shown in Figure 4.1b is stable, one may choose PE1 and PE3 as hubs. This leads to 16, rather than 25, route entries in total. Although the paths of most active communications remain unaffected (as denoted by solid edges), this Relaying configuration requires some communications (dotted edges) be detoured through hubs, offering *indirect* reachability. This indirect delivery obviously inflates some paths' length, leading to the increase of latency, additional resource consumption in the backbone, and larger fate sharing. Fortunately, reducing this side effect is possible when one can build a set of hubs that originates or receive most traffic within the VPN. Meanwhile, reducing the memory footprint of routing tables requires the hub set to be as small as possible. In the following sections, we show that composing such a hub set is possible.

4.4.2 Hub selection vs. hub assignment

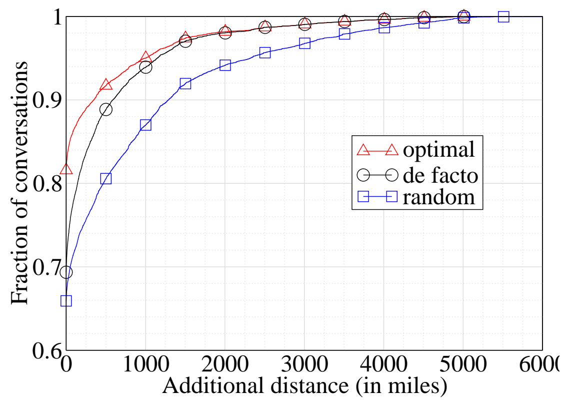
Relaying is composed of two different sub-problems: *hub selection* and *hub assignment* problems. Given a VPN, a *hub selection* problem is a decision problem of selecting each PE in the VPN as a hub or a spoke. On the other hand, a *hub assignment* problem is a matter of deciding which hub a spoke PE should use as its default route. A spoke must use a single hub consistently because, by definition, a PE cannot change its default route for each different destination. To implement Relaying, we let each hub advertise a default route (i.e., `0.0.0.0/0`) to spoke PEs via BGP. Thus, in practice, the BGP routing logic



(a)



(b)



(c)

Figure 4.3: (a) Gain and cost (de facto asgn.), (b) Sum of the products of volume and additional distance, (c) CDF of additional distances ($\alpha = 0.1$)

running at each PE autonomously solves the hub assignment problem. Since all updates for the default route are to be equivalent for simplicity, each PE chooses the closest hub in terms of IGP (Interior Gateway Protocol) distance. We call this model the *de facto* hub assignment strategy.

In order to assess the effect of the de facto strategy on path inflation, we compare it with some other hub assignment schemes, including random, optimal, and algorithm-specific assignment. The *random* assignment model assigns a random hub for each non-local destination. In the *optimal* assignment scheme, we assume that each PE chooses the best hub (i.e., the one minimizing the additional distance) for each non-local destination. Note that this model is impossible to realize because it requires global view on routing. Finally, the *algorithm-specific* assignment is the assignment plan that our algorithm generates. This plan is realistic because it assumes a single hub per spoke, not per destination.

4.5 Baseline Performance of Relaying

To investigate fundamental trade-off relationship between the gain (i.e., memory saving) and cost (i.e., increase of path lengths and the workload in the backbone due to detouring), we first explore a simple, light-weight strategy to reduce routing tables. Despite its simplicity, this strategy saves router memory substantially with only moderate penalties. Note that the Relaying schemes we propose in later sections aim to outperform this baseline approach.

4.5.1 Selecting heavy sources or sinks as hubs

The key problem of Relaying is building a right set of hubs. Fortunately, spatial locality of traffic matrices hints us that forcing a spoke PE to communicate only through a hub might increase memory saving significantly, without increasing the path lengths of most conversations. Thus, we first investigate the following simple hub selection strategy, namely *aggregate volume-based hub selection*, leveraging the sparsity of traffic matrices.

For a PE p_i in VPN v, we measure the aggregate traffic volume to and from the PE. We denote a_i^{in} to be the aggregated traffic volume received by p_i from all customer sites directly connected to p_i, and a_i^{out} to be the aggregated traffic volume sent by p_i to all customer sites directly attached to p_i. In VPN v, if $a_i^{in} \geq \alpha \sum_j a_j^{in}$ or $a_i^{out} \geq \alpha \sum_j a_j^{out}$ where α is a tunable parameter between 0 and 1 inclusively, then we choose p_i as a hub in v.

Although we could formulate this as an optimization problem to determine the optimal value of α (for a certain VPN or for all VPNs) minimizing a multi-objective function (e.g., a weighted sum of routing table size and the amount of traffic volume relayed via hubs), this approach lead us to two problems. First, it is hard to determine a general, but practically meaningful multi-objective utility function especially when each of the objectives has a different meaning. Second, the objectives (e.g., memory saving) are not convex, making efficient search impossible. Instead, we perform numerical analysis with varying values of α and show how table size and the amount of relayed traffic volume varies across different α values. Since there are hundreds of VPNs available, exploring each individual VPN with varying α values broadens the solution space impractically large. Thus we apply a common α value for all VPNs.

4.5.2 Performance of the hub selection

Performance metrics: To assess Relaying performance, we measure four quantities: *metric-i*) the number of routing table entries reduced, *metric-ii*) the amount of traffic that is indirectly delivered through hubs, *metric-iii*) the sum of the products of traffic volume and additional distance by which the traffic has to be detoured, and *metric-iv*) the additional distance of each conversation's forwarding path. For easier representation, we normalize the first three metrics. *Metric-i* is normalized by the total number of routing entries before Relaying, *metric-ii* is normalized by the amount of total traffic in the VPN, and *metric-iii* is normalized by the sum of the products of traffic volume and direct (i.e., shortest) distance. We consistently use these metrics throughout the rest of the chapter.

The meanings of these metrics are as followings. *Metric-i* quantifies our scheme's *gain* in memory saving, whereas *metric-ii, iii* and *iv* denote its *cost*. Specifically, *ii* and *iii* show the increase of workload on the backbone. On the other hand, *iv* shows the latency inflation of individual PE-to-PE communications. Note that we measure the latency increase in distance (i.e., miles) because, in the backbone of a large tier-one network, propagation delay dominates a path latency. Due to the speed of light and attenuation, a mile in distance roughly corresponds to 11.5 usec of latency in time. Thus, increasing a path length by 1000 (or 435) miles lead to the increase of unidirectional latency roughly by 11.5 (5) msec.

Relaying results: Figure 4.3a shows the gain and cost of the aggregate volume-based hub selection scheme across different values of the volume threshold α. As soon as we apply Relaying (i.e., $\alpha > 0$), all three quantities increase because the number of hubs decreases as α increases. Note, however, that the memory saving increases very fast, whereas the amount of relayed traffic and its volume-mile product increases modestly. If we assume

a sample utility function that is an equally weighted sum of the memory saving and the relayed traffic volume, the utility value (i.e., the gap between the gain and the cost curves) is maximized where α is around 0.1 to 0.2. When α passes 0.23, however, the memory saving begins to decrease fast because a large value of α fails to select hubs in some VPNs, making those VPNs revert to the direct reachability architecture between every pair of PEs. This also makes the cost values decrease as well.

Figure 4.3b shows how different hub assignment schemes affect the cost (specifically, the increase of workload on the backbone manifested by the sum of the products of volume and additional distance). Note that we do not plot the gain curve because it remains identical regardless of which hub assignment scheme we use. First, the graph shows that the overall workload increased by Relaying with either the de facto or the optimal assignment is generally low (less than 14% for any α). Second, the de facto assignment only slightly increases the workload on the backbone (around 2%) in the sweet spot ($0.1 < \alpha < 0.2$), compared to the optimal (but impractical) scheme. The increase happens because the de facto scheme forces a spoke to use the closest hub consistently, and that closest hub might not be the spoke's popular communication peer. Nevertheless, this result indicates that choosing the closest hub is effective in reducing the path inflation.

Although the sum of the volume-mile products is reasonably small, the increased path lengths can be particularly detrimental to some *individual traffic flows*. Figure 4.3c shows, for all communicating pairs in all VPNs, how much additional distances the Relaying scheme incurs. The figure shows latency distributions when using Relaying (with $\alpha = 0.1$) for three different hub assignment schemes: optimal, de facto, and random. For example, when using Relaying with the de facto assignment scheme, roughly 70% of the communicating pairs still take the shortest paths, whereas around 94% of the pairs experience additional distances of at most 1000 miles (i.e., the increase of unidirectional

latency by up to 11.5 msec). Unfortunately this means that some 6% of the pairs suffer from more than 1000 miles of additional distances, which can grow in the worst case larger than 5000 miles (i.e., additional 60 msec or more unidiretionally). To those communications, this basic Relaying scheme might be simply unacceptable, as some applications' quality may drastically drop. Unfortunately, the figure also shows that even the optimal hub assignment scheme does not help much in reducing the particularly large additional path lengths. *To remove the heavy tail, we need a better set of hubs*.

4.6 Latency-constrained Relaying

Relaying requires spoke-to-spoke traffic to traverse an indirect path, and therefore increases paths' latency. However, many VPN applications such as VoIP are particularly delay-sensitive and can only tolerate a strictly-bounded end-to-end latency (e.g., up to 250 ms for VoIP). SLAs routinely specify a tolerable maximum latency for a VPN, and violations can lead to adverse business consequences, such as customers' loss of revenue due to business disruptions and corresponding penalties on the provider.

The simple baseline hub selection scheme introduced in Section 4.5 does not factor in the increase of path latencies due to relaying. Thus, we next formulate the following optimization problem, namely *latency-constrained Relaying (LCR) problem*, of which goal is to minimize the memory usage of VPN routing tables subject to a *constraint on the maximum additional latency of each path*. Note that we deliberately bound individual paths' additional latency, rather than the overall stretch, because guaranteeing a certain hard limit in latency is more important for applications. For example, increasing a path's latency from 30 msec (a typical coast-to-coast latency in the U.S.) to 60 leads to the stretch of only 2, whereas the additional 30 msec can intolerably harm a VoIP call's quality. On

the other hand, increasing a path's latency from 2 msec to 10 may be nearly unnoticeable to users, even though the stretch factor in this case is 5.

4.6.1 LCR problem formulation

We first introduce the following notation. Let P denote the set of PE routers $P = \{p_1, p_2, ..., p_n\}$ in VPN v. We define two matrices: *i*) Conversation matrix $C = (c_{i,j})$ that captures the historical communication between the routers in P, where $c_{i,j} = 1$ if $i \neq j$ and p_i has transmitted traffic to p_j during the measurement period, and $c_{i,j} = 0$ otherwise; and *ii*) latency matrix $L = (l_{i,j})$ where $l_{i,j}$ is unidirectional communication latency (in terms of distance) from p_i to p_j. $l_{i,i} = 0$ by definition. Let $H = \{h_1, ..., h_m\}$ $(m \leq n)$ be a subset of P denoting the hub set. Finally, we define mapping $M : P \rightarrow H$ that determines a hub $h_j \in H$ for each $p_i \in P$.

LCR is an optimization problem of determining a smallest H (i.e., hub selection) and a corresponding mapping M (i.e., hub assignment), such that in the resulting Relaying solution, every communications between a pair of VRFs adhere to the maximum allowable additional latency (in distance) threshold θ. Formally,

$$\begin{aligned} \text{min} \quad & |H| \\ \text{s.t.} \quad & \forall s, d \text{ whose } c_{sd} = 1, \\ & l_{s,M(s)} + l_{M(s),d} - l_{s,d} \leq \theta \end{aligned}$$

Other variations of the above formulation include bounding either the maximum total one-way distance, or both the additional and the total distances. We do not study these variations due to the following reasons. First, bounding the additional distance is a stricter condition than bounding only the total distance is. Thus, our results in the following sections provide lower bounds of memory saving and upper bounds of indirection penalties.

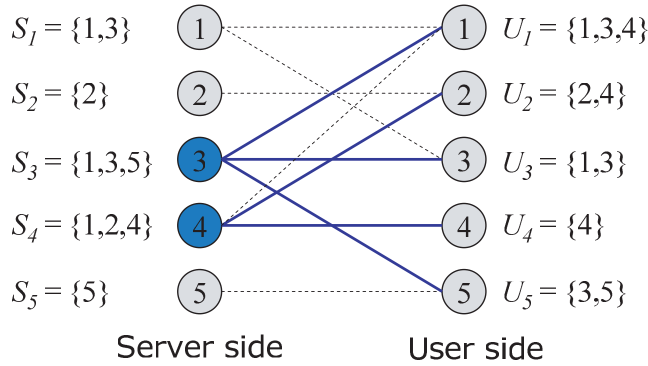
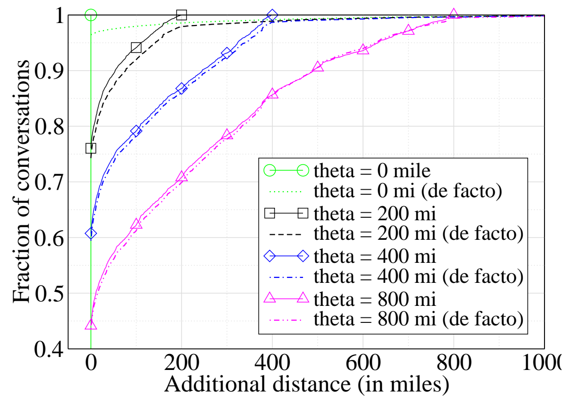


Figure 4.4: A sample serve-use relationship

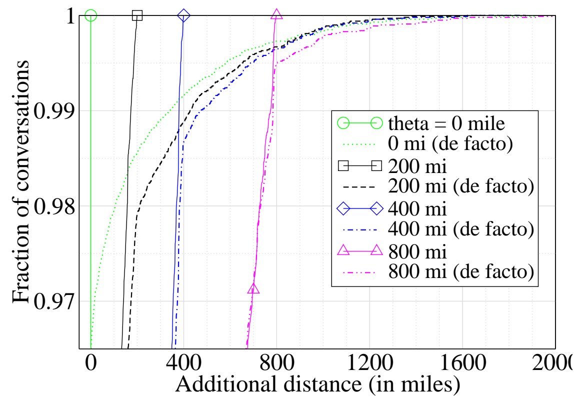
Second, when bounding total and additional distances, the total distance threshold must be larger than the maximum direct distance. However, this maximum direct distance often results from a small number of outlier conversations (e.g., communication between Honolulu and Boston in the case of the U.S.), making the total distance bound ineffective for most common conversations.

Considering the any-to-any reachability model of MPLS VPNs, we could accommodate the possibility that any PE can potentially communicate with any other PEs in the VPN, even if they have not in the past. Thus, we can solve the LCR problem using a *full-mesh* conversation matrix $C^{full} = (c^{full}_{i,j})$, where $\forall i, j \ (i \neq j) \ c^{full}_{i,j} = 1, \ c^{full}_{i,i} = 0$. There is trade-off between using the usage-based matrices (C) and full-mesh matrices (C^{full}). Using C^{full} imposes stricter constraints, potentially leading to lower memory saving. The advantage of this approach, however, is that the hub selection would be oblivious to the changes in communication patterns among PEs, obviating periodical re-adjustment of the hub set.

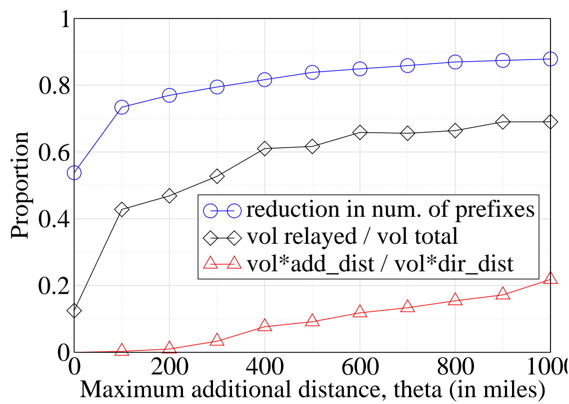
Unfortunately, the LCR problem is NP-hard, and we provide this proof in the paper containing the extended version of this chapter [85]. Hence, we propose an approximation



(a)



(b)



(c)

Figure 4.5: LCR performance with usage-based C (a) CDF of additional distances, (b) CDF of additional distances (zoomed in), (c) Gain and cost

algorithm explained in the following subsection.

4.6.2 Algorithm to solve LCR

In this section, we outline our solution to the *LCR* problem. The pseudo-code of this algorithm is also given in our extended version [85]. We solve a LCR problem through a two-staged approach. The first stage builds *serve-use* relationships among PEs. Given a pair of PEs $(p_i, p_j) \in P$, we say that p_i can *serve* p_j, or conversely, that p_j can *use* p_i, if all conversations in C from p_j to other PEs in the VPN can be routed via p_i as a hub without violating the latency constraint. For each PE p_i in P, we builds two sets: *i*) the *serve* set S_i composed of PEs that p_i can serve as a hub, and *ii*) the *use* set U_i composed of PEs that p_i, as a spoke, can use for relaying. The serve-use relationships among PEs can be represented as a bipartite graph. Figure 4.4 shows a sample serve-use relationship graph of a VPN with five PEs. Each node on the left bank represents $p_i \in P$ in its role as a possible hub, and on the right bank, each node represents p_i in its role as a potential spoke. The existence of serve-use relationship between $(p_i, p_j) \in P$ is represented by an edge between p_i on the left bank and p_j on the right bank.

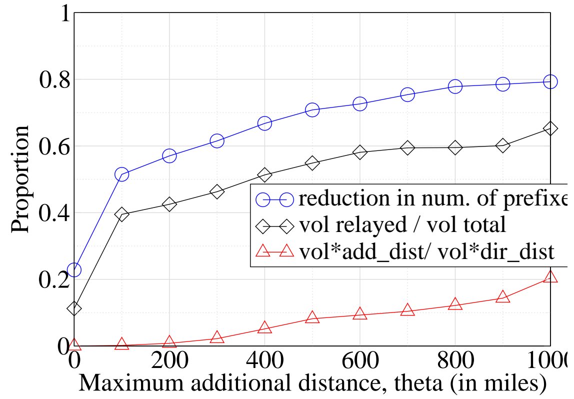
Our original LCR problem now reduces to finding the smallest number of nodes on the left bank (i.e., hubs) that can serve every node on the right bank. This is an instance of the set cover problem, which is proven to be NP-complete [86], and we use a simple greedy approximation strategy to solve this. At each step, the LCR algorithm *i*) greedily selects p_i from the left bank whose serve set S_i is the largest, *ii*) remove p_i from the left bank, *iii*) remove all p_j $(j \in S_i)$ from the right bank and update the mapping function M to indicate that p_j's assigned hub is p_i, and *iv*) revise the serve-use relationships for the remaining PEs in the graph. The above step is repeated until no PE remains on the right bank.

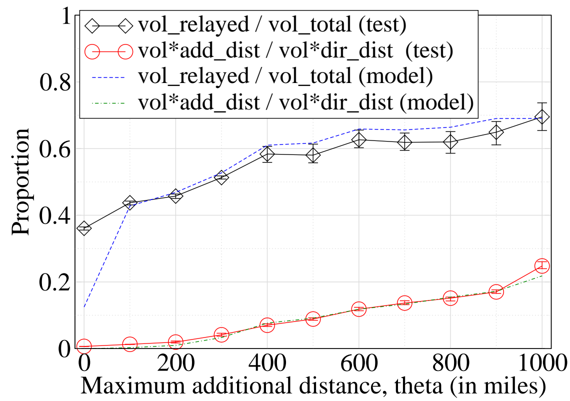
The LCR algorithm can be easily extended to solve the alternative problems mentioned above (i.e., bounding total latency, or both additional and total latency) only by re-defining the semantics of "serve" and "use" relationship. Note also that the LCR algorithm assigns a single hub for each spoke PE, and each spoke PE is assumed to use the single hub consistently for all packets. Thus, the hub assignment plan generated by the LCR algorithm is implementable, if not as simple as the de facto assignment scheme based on BGP anycasting as described in Section 4.4. For example, a VPN provisioning system can configure each spoke PE's BGP process to choose a specific route advertisement from a corresponding hub.

4.6.3 Solutions with usage-based matrices

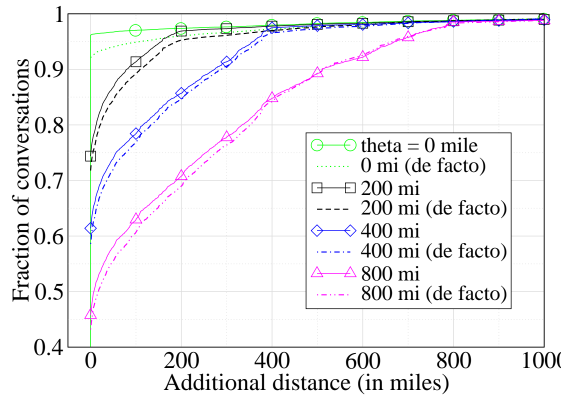
We run the LCR algorithm individually for each VPN. The results shown in later sections are the aggregate of individual solutions across all VPNs.

Bounding additional distances: Figure 4.5a shows the CDFs of additional distances of all conversation pairs in C across all VPNs for varying θ values. Solid lines show the additional distance distributions when using the algorithm-specific hub assignment decisions (i.e., assignment plans computed by the LCR algorithm), whereas the dotted lines represent the distributions when spoke PEs choose the closest hubs in terms of the IGP metric (i.e., the de facto hub assignment). Note, however, that for a given θ, both solid and dotted curves are generated with the same hub set. For some spoke PEs, the hub determined by the LCR algorithm might be different from the closest hub that the de facto assignment scheme would choose. Thus, when one uses the hub sets selected by the LCR algorithm accompanied with the de facto hub assignment scheme, some pairs of PEs might experience additional distances larger than θ. This is why solid lines always conform the θ values, whereas dotted lines have tails extending beyond theta.





(a)



(b)

(c)

Figure 4.6: (a) LCR performance with C^{full}, (b) Robustness results (costs), (c) Robustness results (CDF of additional distances with C)

However, the fraction of PE pairs experiencing additional distances larger than θ is small. We re-scale Figure 4.5a to magnify the tails and present it as Figure 4.5b. For a reasonable value of θ (e.g., additional 400 miles, which are commensurate with 5 msec increase in latency), only 1.3% of PE pairs experience additional distances larger than θ. Meanwhile, the fraction of pairs exposed to unbearably larger additional distances (e.g., $1000+$ miles, or more than 11.5 msec in latency) is only 0.2%, which amounts to a few tens of pairs. These results suggest that in practice, administrators can use the simpler de facto assignment scheme in general and can configure only those small number of exceptional PEs as per specific assignment plans dictated by the LCR algorithm.

Memory saving: We next investigate the memory saving from our latency constrained hub selection. Figure 4.5c shows the gain (i.e., memory saving) and cost (i.e., the increase of workload in the backbone) under LCR for a range of θ. Both the gain and cost curves increase with θ because a larger value of θ makes it possible for a smaller number of hubs to serve all the spokes.

The results convey a number of promising messages. For example, to achieve roughly 80% memory saving, a conversation (i.e., VRF pairs) need to tolerate additional distance of only up to 320 miles, which corresponds to the increase of unidirectional latency by just 3 ms. Moreover, when conversations can bear at most 1000 miles of path inflation, Relaying can reduce routing table memory by roughly 88%. Note that this amount of memory saving is only 8% worse than that of the ideal memory saving scheme mentioned in Section 4.3.2, and is better than that of the aggregate volume-based scheme (Figure 4.3a) with any choices of the volume threshold α.

A surprising result is that, even if we do not allow any additional distance (i.e., $\theta = 0$), the relaying scheme can reduce routing table memory consumption by around a hefty 54%. By specifically analyzing these penalty-free cases, we found out three reasons

for this huge, free-of-charge benefit. First, a significant number of PEs communicate mostly with a hub PE. In the case of the traffic whose source or destination is a hub, our relaying scheme does not introduce any additional distance penalty as long as the hubs are correctly identified and assigned. This case accounts for roughly 45% of the entire penalty-free conversations. Second, a hub can sometimes be located on the shortest path from a source to a destination (38%). For example in the United States, relaying traffic between San Diego and San Francisco through Los Angeles might not incur any detour in effect because physical connectivity between the two cities might be already configured that way. Third, for availability purposes, a VPN is often provisioned over multiple PEs located at the same POP (17%). Thus if one of the PEs in a POP is chosen as a hub, all the other PEs at the same POP can use the hub PE for relaying and avoid any (practically meaningful) distance penalties.

Indirection penalty: Figure 4.5c shows that the latency-constrained Relaying requires substantially more traffic to be relayed compared to the base line relaying scheme shown in Figure 4.3a. (Note that we use the same metrics described in Section 4.5.2.) This is because the LCR algorithm does not take into account the relative traffic volumes associated with the different PE-to-PE conversations while making either the hub selection or hub assignment decisions; the LCR algorithm selects a PE as a hub as long as the PE can relay the largest number of other PEs without violating the latency constraint. However, Figure 4.5c also shows that the sum of the relayed volume and additional distance products is a relatively small compared to the corresponding sum of the volume and direct distance products. This is because, even when relaying is needed, the algorithm limits the additional distances below the small θ values. Hence, the *practical impact* of relaying (e.g., the increase of resource usage in the provider network) is much less severe than it is suggested by the sheer amount of relayed traffic. Also, we confirmed that using the

de facto hub assignment, rather than the LCR algorithm's hub assignment plan, increases the aggregate costs very little. This is because the LCR algorithm does not necessarily choose heavy sources or sinks as hubs, leaving only little room to improve/worsen the indirection penalties via different hub assignments.

4.6.4 Solutions with full-mesh matrices

We next consider the performance of the LCR when using the full-mesh conversation matrices. Figure 4.6a shows the gain and the cost curves. While we select the hubs based on the full-mesh matrices, to evaluate the penalties due to relaying we use the historical PE-to-PE conversations (including volumes) that are monitored during the measurement period (May 13 - 19, 2007).

The results are encouraging, even though the conversation matrices are much denser in this case. At the expense of incurring additional distances of up to roughly 480 miles (corresponding only to roughly 5 ms in unidirectional latency), we can reduce the memory consumption by nearly 70%. Interestingly, we can still save roughly 23% of memory even with no additional distance. This is is because, given a PE, it is sometimes possible to have a hub lying on every shortest path from the PE to each other PE. For example, on a graph showing the physical connectivity of a VPN, if a spoke PE has only one link connecting itself to its neighbor, and the neighbor is the hub of the spoke PE, delivering traffic through the neighbor does not increase any paths' lengths originating from the spoke PE. We also note that the aggregate costs (i.e., relayed traffic volume and its volume-mile product) are slightly reduced compared to the results derived from usage-based matrices. This is because hub sets for full-mesh matrices are bigger than those for usage-based matrices. We also confirmed that the lengths of individual paths are correctly bounded within θ for all cases.

For network administrators, these full-mesh results are particularly encouraging for the following reasons. First, even when customers' networking patterns drastically change from todays hub-and-spoke style to a more peer-to-peer-like style (e.g., by pervasively deploying VoIP or P2P-style data sharing applications), the relaying scheme can still save significant amount of memory. Second, it has the additional attraction that the relaying solution needs to be computed and implemented only once and for all. There would be no need to track the changing conversation matrices, unlike the usage-based case.

4.6.5 Robustness

We next explore how our solution performs under traffic dynamics. Figure 4.6b shows the results when we apply a *fixed* solution set derived from the traffic measurements in a particular *model* week (May 13 - 19, 2007) to the traffic for 8 different *test* weeks in May to August, 2007. We assume that VRFs added later on after the model week maintains full reachability.

The solid curves in Figure 4.6b depict the aggregate cost during the test weeks. We apply the usage-based solutions – both the hub selection and assignment results – from the model week to the traffic matrices of the test weeks. Error bars represent the maximum and minimum cost across all 8 weeks with each value of θ, whereas the curve connects the averages. For comparison, we also plot the cost curves for the model week using dotted lines. We do not plot the gain curves because we use the same hub set for all the weeks. The results are promising; the aggregate cost curves for the test weeks are close to those of the model week for all choices of θ except 0. The exception when $\theta = 0$ occurs because the strict latency condition leads to a solution set that is very sensitive to traffic pattern changes. The tight error bars show that the aggregate cost is stable with little

variance across the 8 test weeks. We found similar results with the full-mesh solution of the model week, and omit the graphs in the interest of space.

Figure 4.6c shows the distribution of additional distances when we apply the usage-based solutions from the model week to one (July 23 - 29, 2007) of the test weeks. The solid lines show the additional distances when we use both the hub selection and assignment plans of the model week. The dotted lines represent the distances when combining the hub selection results with de facto hub assignment. The graph is similar to Figure 4.5a, meaning that the site-to-site latency distribution remains fairly stable over the test weeks. However, the fraction of communication pairs whose additional distance is larger than the specified θ increases by at most 3%, leaving heavier tails. Note that, due to traffic dynamics, just using the hub assignment results of the model week (i.e., solid curves) does not guarantee the conformance to θ in subsequent weeks because the conversation matrix changes. However, the fraction of such cases is small. In the case of $\theta = 400$, only less than 2.5% of conversation pairs do not meet the latency constraint. We verified that these tails are removed when using the full-mesh solutions of the test week. In conclusion, the latency-constrained hub selection and assignment scheme generates robust solutions.

4.7 Latency-constrained, Volume-sensitive Relaying

In addition to bounding additional latency within a threshold, we also want to reduce additional resource consumption in the backbone required for Relaying. We view the sum of volume and additional distance products is one of the relevant metrics to measure the load a backbone has to bear. This requirement motivates another optimization problem, namely *latency-constrained, volume-sensitive Relaying (LCVSR)*.

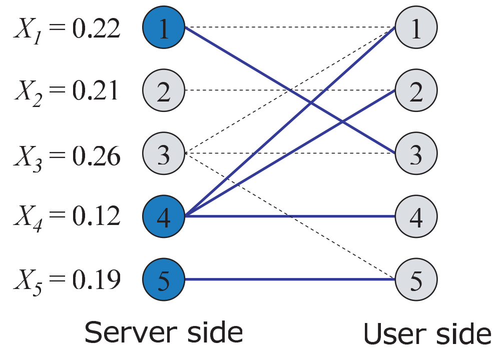


Figure 4.7: A sample serve-use relationship with penalties

4.7.1 LCVSR problem formulation

To formulate this problem, we define a volume matrix $V = (v_{i,j})$ of a VPN where $v_{i,j}$ denotes the amount of traffic volume that p_i sends to p_j. Obviously, $v_{i,j} = 0$ when $c_{i,j} = 0$. Now we consider the following problem:

$$\begin{aligned} \text{min} \quad & |H|, \textstyle\sum_{\forall s,d} v_{s,d} \cdot (l_{s,M(s)} + l_{M(s),d} - l_{s,d}) \\ \text{s.t.} \quad & \forall s, d \text{ whose } c_{s,d} = 1, \\ & l_{s,M(s)} + l_{M(s),d} - l_{s,d} \leq \theta \end{aligned}$$

To provide robust solutions for the worst case, we can also utilize the full-mesh conversation matrix C^{full}. Note, however, that we still need to use the usage-based volume matrix V as well because we cannot correctly assume the volumes of the conversations that have never taken place. Thus, a hub set generated by this formulation can serve traffic between any pairs of PEs without violating the latency constraint, and keeps the amount of relayed volume relatively small as long as the volume matrices considered are similar to the usage-based matrices. Hence, assuming that the future communication between two PEs who have never spoken to each other in the past generates relatively small amount of traffic, this full-mesh solution might still work effectively.

4.7.2 Algorithm to solve LCVSR

We outline our LCVSR algorithm in this section. The pseudo-code is given in [85]. Minimizing $|H|$ conflicts with minimizing $\sum_{\forall s,d} v_{s,d} \cdot (l_{s,M(s)} + l_{M(s),d} - l_{s,d})$, making the problem hard to solve. For example, to reduce the additional resource consumption to zero, every PE should become a hub, leading to the maximum value of $|H|$. Note, however, that this does not mean that the additional resource consumption can be minimized only and if only the hub set is maximal. In fact, for some C, it is possible to minimize the additional resource consumption with a very small $|H|$ – one trivial example is the case where there is only one popular PE, and all the other PEs communicate only with the popular PE. Although we cannot minimize both the objectives at the same time in general, coming up with a unified objective function composed of the two objectives functions (e.g., a weighed sum of the two objectives) is not a good approach either because the two objectives carry totally different meanings. Thus, we propose a simple heuristic to build a reasonably small hub set that reduces the relayed volume.

Our algorithm for LCVSR works similarly to the algorithm for the LCR problem. Thus, the first stage of the algorithm builds the same serve-use relationships among PEs. However, during the process, the algorithm also computes a *penalty* value for each PE. The penalty X_i of PE p_i is defined to be the sum of the volume and additional distance products of all conversations in C, assuming p_i is a sole hub in the VPN. That is, $X_i = \sum_{\forall s,d} v_{s,d} \cdot (l_{s,i} + l_{i,d} - l_{s,d})$. Figure 4.7 illustrates a sample serve-use relationship graph with five PEs, where each PE is annotated with its penalty value. With these serve-use relationships along with penalties, the algorithm chooses hubs.

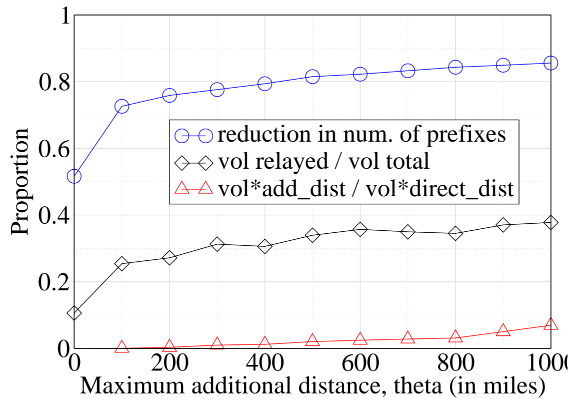
Due to the two conflicting objectives, at each run of the hub selection process, the algorithm has to make a decision that reduces either one of the two objectives, but not necessarily both. We design our algorithm to choose a PE from the server side (i.e.,

the left bank) that has *the smallest penalty value*, rather than the largest serve set size. The intuition behind this design is that choosing a PE with the smallest penalty is often conducive to reducing *both* objectives, if not always. By definition, a penalty value X_i of PE p_i decreases when each product (i.e., $v_{s,d} \cdot (l_{s,i} + l_{i,d} - l_{s,d})$) becomes smaller. Now suppose a PE that communicates with a large number of other PEs; we call such a PE has a high *span*. The penalty of the high-span PE is usually lower because the high-span PE is on the shortest path of many conversations, making many of the volume-distance products zero. Thus, our algorithm tends to choose PEs with higher spans. The key is that, fortunately, a PE with a higher span *also* has a large serve set (i.e., S_i) because it can serve as a hub a large number of PEs it communicates without violating the additional distance constraint. The remaining part (removing the chosen PE, and revising the serve-use relationships) is identical to the previous algorithm. We repeat the process until no PE remains on the user side.

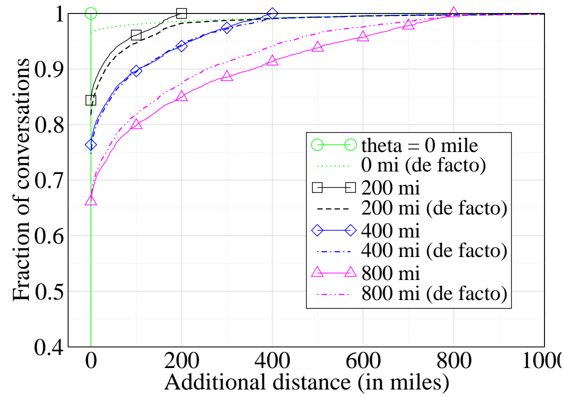
4.7.3 Solutions with usage-based matrices

The benefit of LCVSR over LCR is shown in Figure 4.8a. The figure indicates that LCVSR algorithm significantly reduces indirection penalties compared to the LCR algorithm. The amount of relayed traffic volume increases much more slowly than does it with the LCR algorithm and never exceed 40% of the total volume for any θ below 1000 miles. In comparison, in Figure 4.5c, the same cost curve lies above the 40% line for almost all θ, reaching nearly 70% when $\theta = 1000$. This decrease in relayed volume also reduces the sum of volume and additional distance products. For a reasonable choice of θ (e.g., 400 miles), the sum of the volume and additional mile products is only 1.2% of the corresponding total.

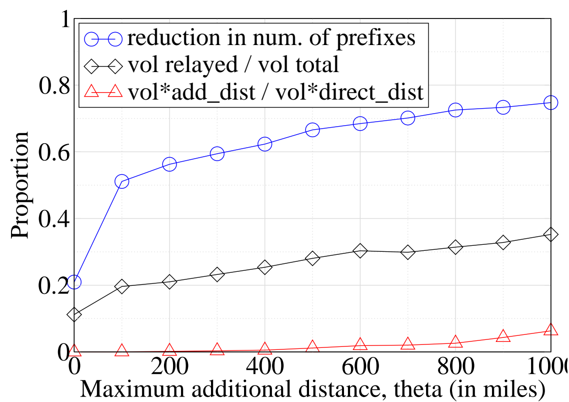
Figure 4.8a also shows the LCVSR can still substantially reduce router memory us-



(a)



(b)



(c)

Figure 4.8: LCVSR performance (a) Gain and cost (with C), (b) CDF of additional distances (with C) (c) Gain and cost (with C^{full})

age, and also that the amount of memory saving is marginally lower than the results of LCR. Specifically, a comparison with Figure 4.5c reveals that the saving for each θ is lower by only 1 to 3%. While, the lower saving is somewhat expected given that LCVSR does not explicitly aim to minimize $|H|$, the small difference in saving suggests that the LCVSR is still very effective in identifying memory-efficient solutions.

In conclusion, the LCVSR scheme results in very modest increase of backbone workload, while enabling dramatic memory saving. We also confirmed that combining the LCVSR with the simpler-to-implement de facto assignment scheme, instead of the assignment dictated by the algorithm, marginally affects the aggregate cost.

We also note that the distribution of path lengths is biased towards lower values for the LCVSR compared to the LCR, as evidenced by comparing the each CDF curve in Figure 4.8b to the corresponding curve in Figure 4.5a. In particular we note that for any given θ, a significantly larger fraction of communications have no path inflation. This reduced inflation is probably a consequence of the LCVSR algorithm tending to choose PEs with higher span values, rather than those PEs whose locations are qualified to serve a larger number of other PEs, as hubs. Using the de facto assignment scheme, however, also adversely impacts a few (e.g., 0.9% of the total communication pairs in the case of $\theta = 400$) individual pairs, leading to the increase of additional distances beyond θ.

4.7.4 Solutions with full-mesh matrices

Here we compare LCVSR with LCR, both using full-mesh conversation matrices. Recall that a full-mesh conversation matrix of a VPN factor in the latency for conversations between every PE pair. Hence, for a given PE, the number of hub PEs that can serve the given PE tends to be smaller than the corresponding number with the usage-based conversation matrix. This in turn suggests that hub sets are generally larger in the full-

mesh case. This intuition is confirmed in Figure 4.8c which shows that the hub set size indeed increases. However, we can still save a substantial 22 to 75% of memory for reasonable values of θ in the few hundred miles range. This corresponds to marginally lower (about $1-5\%$) memory saving compared to the LCR algorithm (see Figure 4.6a). On the other hand, the cost of Relaying reduces significantly under LCVSR (about 50% in terms of the amount of relayed volume, and 75 to 95% in terms of the sum of volume and additional distance products) compared to LCR.

4.7.5 Robustness

We next evaluate how a specific LCVSR solution (i.e., a hub selection and assignment results generated by the LCVSR algorithm) performs as conversation and volume matrices change over time. We use data from the same model and test weeks as in Section 4.6.5.

Figure 4.9 presents the aggregate cost during the test weeks with the solid curves and error bars. We apply the usage-based solution (both hub selection and assignment results) derived from the model week to the conversation and volume matrices of the test weeks. For comparison, dotted curves show the corresponding indirection penalties for the model week. First, the figure shows that the increase of backbone workload (i.e., the sum of volume and additional distance products), while still higher than the model week, is quite small and has low variability. For all values of θ, this quantity remains below 12% (below 9% when $\theta \leq 800$). Hence, the actual additional network load in the test weeks is still very low. However, the results also indicate that the amount of relayed traffic volume itself can be substantially higher in the test weeks compared to that in the model week. For example for $\theta = 400$, the relayed volume is higher by 17 to 106%, depending on weeks. This higher and markedly variable amount of relayed volume can be attributed to the fact that LCVSR uses the traffic matrix of the model week in its hub selection, and

that different weeks will have some variability in the volumes of common conversation pairs. Similar results are found when using the full-mesh solutions of the model week, but we do not show them to save space. Despite the increase of relayed traffic volume, we confirmed that the distributions of additional distances during the test weeks are similar to those of the test week (i.e., curves in Figure 4.8b), except that tails become slightly heavier in a similar fashion shown as in the LCR results (Section 4.6.5 and Figure 4.6c).

The fact that the distributions of the additional distances do not change much over the test weeks might seem to be conflicting with the fact that the aggregate relayed volume significantly increases during the test weeks. By manually investigating the traffic patterns of the model and test weeks, we figure out that this happens because conversation matrices (C) are more stable than volume matrices (L) are. For example, suppose PE p_1 communicates with PE p_2 during the model week (i.e., $c_{1,2} = 1$), and both p_1 and p_2 are not hubs (i.e., traffic from p_1 to p_2 is relayed). Note that during the test weeks traffic from p_1 to p_2 never experience additional distances larger than θ because our algorithm guarantees this for all pairs of PEs who communicated during the model week. Now let us consider the volume $v_{1,2}$ of the traffic from p_1 to p_2. When $v_{1,2}$ increases during the test weeks, compared to $v_{1,2}$ of the model week, the fraction of the relayed traffic volume during the test weeks eventually increases, leading to the effect shown in Figure 4.9 without affecting the additional distance distributions.

4.8 Implementation and Deployment

Implementing relaying is straightforward and does not introduce complexity into the existing VPN infrastructure. Given a set of hubs for each VPN, network administrators can easily migrate from the conventional VPN architecture to the relaying architecture only

by slightly modifying PE routers' configuration. Meanwhile, relaying traffic through a hub is handled solely by the hub, without affecting the CEs or links that are incident to the hub. Moreover, both initial deployment and periodic re-adjustment of Relaying do not incur any service disruption.

4.8.1 Implementing Relaying with BGP

PE routers use BGP to disseminate customer prefixes. For scalability, PE routers multiplex route updates of different VPNs over a set of shared BGP sessions established among the PEs. Isolation between different VPNs is then achieved via *route targets* (RTs) and *import policies*. Specifically, all routes of VPN p are tagged with its own route target RT_p when they are advertised. When PEs receive route updates tagged with RT_p, they import those routes only into p's VRF table VRF_p as dictated by p's import policy. Note that implementing this conventional VPN routing architecture requires only one RT and one import policy for each VPN.

Our Relaying scheme can be easily implemented by introducing in each VPN *i*) two RTs to distinguish customer routes from the default routes (`0.0.0.0/0`), and *ii*) two different import policies for hubs and spokes. This mechanism is illustrated in Figure 4.10. Each PE – regardless of whether it is a hub or a spoke – in VPN p advertises customer routes tagged with RT_p^c. Additionally, each hub in p advertises a default route tagged with RT_p^d. Upon receiving updates, a spoke PE imports routes tagged with RT_p^d only, whereas a hub PE imports routes tagged with RT_p^c only. As a result, a VRF table VRF_p^{spoke} at a spoke PE is populated only with the default routes advertised by other hubs and the routes of the locally-attached customer sites. Contrastingly, a hub's VRF table VRF_p^{hub} contains all customer routes in the VPN. Note that this routing architecture ensures for a hub to directly handle relayed traffic without forwarding it to a locally-attached customer

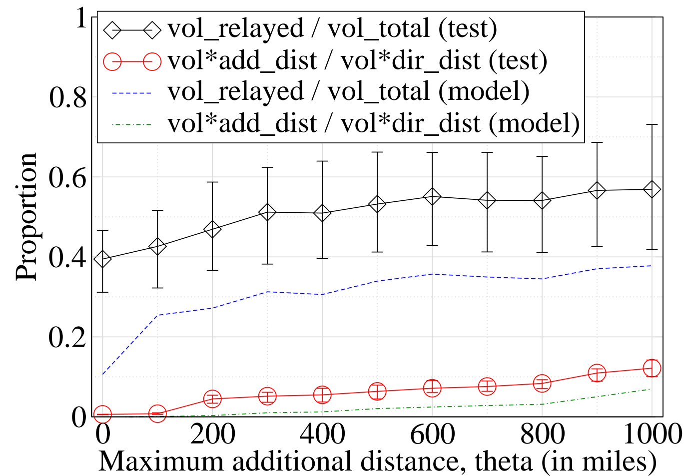


Figure 4.9: Robustness results during test weeks

site because hubs maintain full reachability information. Hence, implementing Relaying does not require any modification to customers.

The mechanism described above naturally implements the de facto hub assignment scheme because each spoke individually selects the closest hub in terms of IGP metric. However, to implement a specific hub assignment plan that our algorithm generates, we need to customize the import policy of each spoke's VRF differently. There may be multiple options enabling such customization, including local preference adjustment.

Unfortunately there is one exceptional case where this implementation guideline should not be directly applied. When a customer site announces the default route on its own (to provide Internet gateway service to other sites, for example), the PE connected to the customer site must be chosen as a hub regardless of our algorithm's decision. By comparing our algorithms' results with the VPN routing configuration, we found that most (97.2%) PEs that are connected to those default-route-advertising sites are anyway chosen as hubs because those sites usually have higher span values or generate/receive large amount of traffic volume. Thus, the penalty due to those exceptional cases are very

low.

4.8.2 Managing Relaying-enabled VPNs

Run-time complexity: The complexity or our algorithms is $O(n^3)$ per VPN where n is the number of PEs in a VPN. A sparse usage-based traffic matrix often allows much better run times in practice – $O(n^2)$ especially when each spoke communicates with a small constant number of hubs. Actually, given hundreds of VPNs each of which spans some tens to hundreds of PEs, running our prototype script with usage-based traffic matrices measured over one week takes less than an hour. With full-mesh matrices, the completion time goes up to tens of hours. Thus, assuming a simple configuration automation tool set [87], running our algorithms and correspondingly re-configuring PEs can be easily performed as a weekly management operation. With further optimization (e.g., coding in C language), daily adjustment might be possible as well.

Reconfiguration overhead: To assess the incremental re-configuration overhead, we measured how stable hub sets are across different measurement/re-configuration windows. We used two different sizes (i.e., a week and a month) of windows. For 9 weeks beginning May 07, 2007, we measured how many hub PEs out of those chosen for a window are also chosen as hubs in the following window. The results are shown in Table 4.1. Overall, the hub sets are quite stable; more than 94% (91%) percent of PEs chosen for a week (month) remain as hubs in the following week (month). The results also confirms the followings: *i*) Using a smaller measurement window (i.e., a shorter re-configuration interval) ensures higher stability, *ii*) full-mesh solutions are more stable than usage-based solutions, and *iii*) the LCR solutions are more stable than the LCVSR solutions.

Availability: One concern about utilizing Relaying is the potential decrease of availability

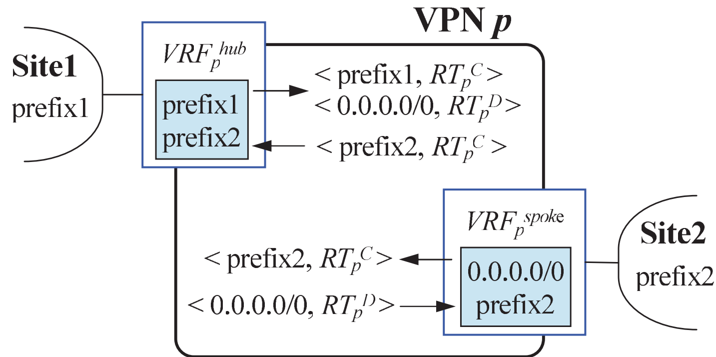


Figure 4.10: BGP configuration for Relaying

Table 4.1: Proportions (in percentage) of hubs that remain as hubs across two windows (averaged across all windows and θ)

	LCR-usage	*LCR-full*	*LCVSR-usage*	*LCVSR-full*
weekly	96.4	99.2	94.3	98.4
monthly	91.2	97.8	91.0	97.3

due to the small number of hubs. When one of the multiple hubs in a VPN goes down, our Relaying scheme ensures that each spoke using the unreachable hub immediately switches over to another hub. Thus, by having at least two hubs located at different POPs, we can significantly increase a VPN's availability. Although our hub selection algorithms do not consider this constraint (i.e., $|H| \geq 2$), most large VPNs almost always have multiple hubs anyway; 98.3% of VPNs which are provisioned over more than 10 PEs have at least 2 hubs. Moreover, our algorithms in itself ensure that each of those hubs are located at different POPs.

Nevertheless, extending the algorithms to explicitly factor in this additional constraint for every VPN is also straight forward. When the algorithm chooses only one hub for a VPN, we can make the algorithm additionally choose the second best hub in terms of the algorithm's hub selection criterion that does not reside in the same POP as the

first hub. We modified our algorithms this way and measured memory saving effect. The reduction of memory saving due to this modification was only 0.5% at most, and this penalty decreases as θ becomes smaller because a smaller θ naturally chooses more number of hubs per VPN.

4.9 Related Work

To the knowledge of the author, ou in this chapter is the first study focusing on a scalable L3 VPN routing architecture that reduces the the amount of state information stored in routing tables and yet strictly bounds individual path lengths.

Designing a scalable routing architecture for *the general intra- or inter-domain IP routing*, however, is an active research field where several interesting results are available, including CRIO [18], and LISP [17]. One key mechanism commonly used by these works is *indirection*. An indirection mechanism divides the large address space used for end host (or subnet) identification into several small fractions. Then each router stores the reachability information about one of the fractions, rather than the entire address space, leading to smaller routing tables. When a router receives a packet destined to a fraction for which the router is not responsible, the packet is first forwarded via tunneling to a router that is responsible for the fraction, and then finally to the actual destination. Each architectural model mentioned above suggests unique mechanisms to systematically divide the original address space into fractions, and to map a fraction to a router. For example, CRIO uses large super-prefixes and BGP for these purposes, whereas LISP encompasses several variations, including super-prefix-based, DNS-based, or ICMP-based mapping models. However, none of these models suggest specific algorithms that can generate complete indirection configuration satisfying parameterized performance con-

straints. Also, all these models propose caching for reducing path inflation, whereas our approach avoids caching for simplicity and implementability.

Flat routing architectures, such as ROFL [21] and UIP [88], also reduces the amount of state at the expense of increasing path stretch. These works leverage on DHTs (Distributed Hash Tables) to avoid storing individual route entries for each destination. Unfortunately, these solutions are not immediately available and lacks simplicity. Also it is unclear how we can utilize these flat routing schemes in a VPN environment where names (i.e., prefixes) are not flat. For example, a router cannot determine which part of a given destination address it should hash unless it knows the destination prefix. Apart from these, the flat routing schemes are not practically suitable for realizing a constrained indirection architecture because path stretch in those architectures is unlimited in the worst case, and paths may often change as the set of destinations (i.e., prefixes) change.

Understanding the unique nature of VPNs and suggesting a better (e.g., more efficient or scalable) provisioning architecture leveraging the unique nature of VPNs has been of interest to many researchers recently. A study on estimating PE-to-PE traffic matrix from aggregate volume data gathered at the edge [79] has identified the strong “hub-and-spoke” traffic patterns. They used the estimation model to suggest a more efficient admission control scheme [80]. Unfortunately, estimated traffic matrices are not suitable for making relaying decisions since hub selection is sensitive to the conversation matrices, rather than the volumes matrices.

4.10 Summary

The large memory footprint of L3 VPN services is a critical, impending problem to network service providers. As a solution, this chapter suggests Relaying, a highly scalable

VPN routing architecture. Relaying enables routers to reduce routing tables significantly by offering indirect any-to-any reachability among PEs. Despite this benefit, there are two practical requirements that must be considered. First, from customer sites' point of view, end-to-end communication latency over a VPN should not increase noticeably. Second, for the service provider's sake, Relaying should not significantly increase the workload on the backbone.

Reflecting these requirements, we formulate two hub selection and assignment problems and suggest practical algorithms to solve the problems. Our evaluation using real traffic matrices, routing configuration, and VPN topologies draws the following conclusions: *i*) When one can allow the path lengths of common conversations to increase by a few hundred miles (i.e., a few msec in unidirectional latency) at most, Relaying can reduce memory consumption by 80 to 90%; *ii*) even when enforcing the additional distance limit to every conversation, rather than only common ones, Relaying can still save 60 to 80% of memory with the increase of unidirectional latency by around 10 msec at most; and *iii*) it is possible, at the same time, to increase memory saving, tightly bound the increase of workload on the backbone, and bound additional latency of individual conversations.

Our Relaying technique is readily deployable in today's network, works in the context of existing routing protocols, requires no changes to either router hardware and software, or to the customer's network. Network administrators can implement Relaying by modifying routing protocol configuration only. The entire process of Relaying configuration can be easily automated, and adjusting the configuration does not incur service disruption.

In this chapter, we focused on techniques that did not require any new capabilities or protocols. The space of alternatives increases if we relax this strict backwards-compatibility assumption. One interesting possibility involves combining caching with

Relaying, where Relaying is used as a resolution scheme to handle cache misses. Another revolves around having hubs keep smaller non-overlapping portions of the address space, rather than the entire space, and utilizing advanced resolution mechanisms such as DHTs. We are exploring these as part of ongoing work, and the SEATTLE architecture introduced in Chapter 2 will serve as a good model.

Chapter 5

Conclusion

Configuration is the Sisyphean task of network management, which burdens administrators with a huge workload and complexity only to maintain the status quo. This dissertation took an architectural approach toward self-configuring networks that do not compromise other indispensable features for wide deployment, such as scalability and efficiency. This chapter begins by summarizing the contributions in Section 5.1, then suggests avenues for future work in Section 5.2, and finally concludes the dissertation in Section 5.3.

5.1 Summary of Contributions

While sharing the same high-level goal of ensuring self-configuration without sacrificing scalability and efficiency, the specific network architectures introduced in previous chapters take different approaches as to how and where new functions are implemented, which specific aspects of self-configuration, scalability, and efficiency they address, and so forth. In this section, we first summarize the key results of the three network architec-

tures with focus on how and to what extent those architectures achieve this dissertation's goal – self-configuration, scalability, and efficiency. Then we recapitulate how our key principles play pivotal roles in those architecture in ensuring the goals.

5.1.1 Scalable and efficient self-configuring networks can be made practical

Table 5.1 gives an overview of how specifically each architecture in this dissertation addresses the issues of self-configuration, scalability, and efficiency, and what kind of key results each architecture ensures. Altogether, these results demonstrate that scalable and efficient self-configuring networks can be made practical.

Table 5.1: Specific aspects of self-configuration, scalability, and efficiency in the proposed architectures

	Self-configuration	**Scalability**	**Efficiency**
SEATTLE	Ensure reachability without requiring addressing and routing configuration	Decrease control-plane overhead, allowing an Ethernet network to grow an order of mag. larger	Improve link utilization and reduce convergence latency as well
VL2	Obviate configuration for addressing, routing, and traffic engineering	Allow a DC to host hundreds of thousands of servers without over-subscription	Enable dynamic service re-provisioning, increasing server and link utilization
Relaying	Retain self-configuring semantics for VPN customers	Allow existing routers to serve an order of mag. more number of VPNs	Only slightly increase end-to-end latency and traffic workload

Self-configuration

Both SEATTLE and VL2 obviate the need for configuration for most frequent, labor-intensive, and yet complex administrative tasks. More specifically, SEATTLE ensures host-to-host reachability without requiring any addressing and routing configuration.

VL2 takes a further step by not only guaranteeing reachability, but also avoiding congestion in a configuration-free fashion. In addition, Relaying retains the same self-configuring capability as the conventional VPN architecture – allowing individual customer sites to autonomously choose and alter their own address blocks, and letting routers in the provider network self-learn and disseminate that information.

Scalability

In SEATTLE and VL2, the main technical principle enabling self-configuration is flat addressing of end hosts. When dealing with a large number of hosts, however, disseminating and storing non-aggregatable hosts' information can lead to a huge workload in the control plane. SEATTLE effectively solves this problem by partitioning – assigning only a fraction of the entire host information to each switch. This scheme allows a SEATTLE network to grow by more than an order of magnitude larger than a conventional Ethernet network can. While VL2 also improves control-plane scalability through its own non-partitioning approach (i.e., the scalable directory-service system), its novelty and emphasis lie on data-plane scalability achieved via random traffic spreading. Specifically, this allows cloud-service providers to build a huge data-center network using only commodity components. Finally, Relaying substantially reduces the overall memory footprint needed to store customer-routing information and thus enables a VPN provider to host nearly an order of magnitude more customers immediately.

Efficiency

SEATTLE switches run a link-state routing protocol and deliver traffic through shortest paths, rather than through a single spanning tree. This reduces routing-convergence latency and improves link utilization by a huge factor. VL2 basically offers the same benefits, because its switch-level routing mechanism is identical to that of SEATTLE.

Additionally, the random traffic spreading used in VL2 can maintain links' utilization at a uniformly high level and ensure a huge server-to-server capacity. Eventually this mechanism enables agility (i.e., capability to frequently re-provision services over different sets of machines without causing any configuration update or control-plane overhead), which eliminates pod-level resource fragmentation and helps maintain servers' utilization at a uniformly high level as well. Together, all these features can greatly improve the statistical multiplexing gain of a data center. Finally in Relaying, the hub selection algorithm guarantees that any traffic between customer sites is subject to only a small, bounded increase of end-to-end latency. Another variation of this algorithm can additionally bound the increase of traffic workload in the provider network resulting from indirect forwarding. In the end, the overall networking performance, perceived by customers, in a Relaying-enabled VPN would remain equivalent to that in a conventional VPN.

5.1.2 Principles and applications

Earlier in this dissertation (Section 1.2), we introduced three technical principles useful for designing scalable and efficient self-configuring networks. Table 5.2 summarizes how those principles are repeatedly utilized in each of the architectures introduced in this dissertation.

Flat addressing

In SEATTLE, hosts identify themselves using their flat and permanent MAC addresses, and the network also delivers traffic based on those addresses. This ensures exactly the same plug-and-play semantics as Ethernet, guaranteeing backwards-compatibility for end hosts in enterprises. Servers in a VL2 network also utilize permanent, location-

Table 5.2: Key principles and the varying applications of the principles

	Flat Addressing	**Traffic Indirection**	**Usage-based Optimization**
SEATTLE	MAC-address-based routing	Forwarding traffic through resolver switches	Caching host info at ingress switches, and reactive cache update
VL2	Separating hosts' names from their locations	Forwarding traffic through randomly-chosen indirect paths	Utilizing ARP, and reactive cache update
Relaying	Location-independent site addressing	Forwarding traffic through hub routers	Popularity-driven hub selection

independent names to communicate with one another. Since a server's name does not change regardless of which physical or virtual machine the server is provisioned on, VL2 can allow services to be frequently re-provisioned over different sets of machines without causing any configuration change at hosts or switches. Unlike SEATTLE and VL2, Relaying utilizes flat addressing at the level of individual VPN-customer sites, rather than of individual hosts. This gives each site complete freedom to choose its own address blocks and thus allows the network to cope with frequent customer-information churn – which naturally occur due to various business activities, such as branch opening or consolidation, corporate merger and acquisition, altering VPN-service providers – with little configuration overhead.

Traffic indirection

SEATTLE switches retrieve a host's information by forwarding only the first few packets to the host through a resolver switch. To lower complexity and increase scalability when dealing with traffic to less popular or highly-mobile hosts, switches can also forward such traffic always through resolvers. On the other hand, in VL2, switches forward traffic through randomly-chosen indirect paths all the time, and doing so enables the network

to avoid congestion with neither gross over-provisioning nor esoteric traffic engineering. Finally in Relaying, traffic between two unpopular customer sites is always forwarded through a hub router connected to a popular site. This greatly reduces overall memory footprint in provider routers, allowing them to serve an order of magnitude more customer VPNs without hardware or software upgrade.

Usage-based optimization

In SEATTLE, host-information is reactively fetched from a resolver when incoming traffic arrives at ingress switches. Resolved host information is also cached at ingress switches to reduce resolution overhead and end-to-end latency for subsequent packets. When updating cached entries for consistency, switches again employ a reactive approach that can correct only those entries that are actually used by traffic. All these "usage-driven" approaches substantially improve the overall efficiency of a network, in terms of memory footprint in switches, end-to-end performance, control-plane overhead to disseminate host information, etc. VL2 employs the same approaches except that source hosts, rather than ingress switches, resolve and cache host information. Finally, in Relaying, the hub selection algorithm chooses hub routers by taking into account actual traffic patterns among customer sites. This helps reduce the number of hub routers – thus overall memory consumption as well – while bounding the impact on the end-to-end performance those actual conversations experience.

5.2 Future Work and Open Issues

Given the importance of enabling hands-free network operation in various networking environments and at various scales, we believe the following issues deserve further investigation.

Internet-wide flat addressing: Despite potential benefits including mobility and security, enabling flat addressing at Internet scale remains an extremely challenging goal due to scalability and efficiency concerns. We believe that the multi-level one-hop resolution scheme introduced in Chapter 2 (Section 2.3) can offer an initial architecture toward this goal. Our preliminary analysis results indicate that it may still be feasible to achieve scalability and efficiency to such a tremendous extent while ensuring robustness, backwards compatibility, and conformance with existing inter-domain-routing policies remain as challenging future work. The reason these issues become all the more important at Internet scale is because, unlike in edge networks, different independent participants in the Internet (i.e., autonomous systems) are often not cooperating with or even competing against one another. As such, forwarding or resolving through a randomly chosen intermediary network could cause various security, performance, and economic concerns.

Scalable router: Internet routers in the default-free zone are suffering from the inflation of the amount of routing state and churn. Deployment of IPv6 and VPNs stands to worsen this situation even further. While solutions for this problem are being proposed [17, 19], those schemes rely on inter-router coordination and thus require complicated control- and management-plane protocols to maintain consistency across distributed routers, increasing configuration complexity. Instead, it may be interesting to develop a purely stand-alone solution that achieves the same goal, greatly simplifying configuration and deployment. The feasibility of this argument is grounded on the observation that network interface cards in an individual router can collectively offer enough memory capacity to hold the entire set of routes in the Internet, whereas conventional routers utilize the memory resources in the least efficient fashion by forcing the entire set of routes to be redundantly kept on every interface card.

Self-adjusting host and switch groups: For scalability and security purposes, hosts in a network are often grouped into VLANs. Deciding which hosts belong to which VLANs, however, is mostly a manual process relying on administrators' intuition and understanding of performance, host-mobility patterns, and access-control policies. This can lead to either sub-optimal resource utilization across different VLANs or frequent updates of VLAN-membership settings. Thus it would be highly interesting to design a network that self-adjusts VLAN membership and boundaries. By allowing VLANs to be dynamically defined for varying workloads, or by making VLAN boundaries elastic, it would be possible to attain higher efficiency and performance while lowering configuration complexity. Similar approaches could also be extended for defining switch groups, such as areas in link-state routing protocols and regions in the multi-level SEATTLE design introduced in Chapter 2 (Section 2.3).

Novel traffic engineering: Hot spots in a network have traditionally been resolved by traffic engineering and end-to-end congestion control. The former approach is often embodied by adjusting traffic-forwarding paths – tuning routing-protocol parameters, or explicitly setting up specific paths for individual types of traffic. The latter approach achieves its goal by regulating traffic-forwarding rates at end hosts. While the VL2 work in Chapter 3 introduces a novel way of eliminating hot spots, it would also be very exciting to investigate the potentials of other kinds of mechanisms. Among those, two emerging primitives – anycast and live process cloning/migration – seem particularly intriguing. With these primitives it might be possible to move heavy traffic sources and sinks, or even to split them into multiple entities running on different machines and adjust workload distributions over the machines. This approach has several unique benefits over the conventional schemes: effectiveness (as it directly manipulates individual traffic sources and destinations), efficiency (as it essentially increases the capacity share for the

resource-hungry application), and low management complexity (as it does not require frequent update of routing-protocol parameters and thus eliminates transient forwarding anomalies during convergence).

Robust self-configuring networks: Ensuring robust operation of a self-configuring network can be particularly challenging when the network includes faulty or un-trusted entities. This is because such an entity can deliberately or inadvertently compromise both the integrity and availability of a routing system. Moreover, in the case of a malicious participant, it can even hide itself with a fabricated (spoofed) self-configured identity. Potentially promising principles deserving further exploration include self-certifying identifiers, replication of routing and host information, and quorum-based resolution. A resulting architecture will be useful in securing conventional layer-2 networks, which suffer from the vulnerabilities due to flooding and broadcasting.

5.3 Concluding Remarks

Administrators in today's large operational networks need self-configuring network architectures. To run emerging applications, such as cloud or utility computing, a self-configuring network is also paramount, because such a network can substantially reduce service-management workload and increase resource utilization. For real-world deployment, however, self-configuring networks must be scalable and efficient all at once.

As part of a larger effort to re-design networks with this goal in mind, this dissertation *i*) presented key technical principles useful for designing and developing a scalable and efficient self-configuring network; *ii*) proposed a highly-scalable network architecture that combines Ethernet's plug-and-play capability and IP's efficiency (SEATTLE); *iii*) developed a data-center network architecture that ensures tremendous server-to-server

capacity and support for agility, substantially increasing a data-center's overall resource utilization (VL2); and *iv*) presented an alternative VPN routing architecture that, once deployed, immediately allows a VPN provider to host an order of magnitude more customers without any router hardware or software upgrade (Relaying).

At the same time, all the solutions proposed in this dissertation are highly practical and can be rapidly prototyped and deployed – in fact, Relaying can be immediately deployed. VL2 and Relaying have passed pre-deployment tests in laboratory settings and are expected to be rolled out for real-world deployment – for a large public cloud-service provider (VL2) and for large corporate VPNs served by a tier-1 provider in the U.S. (Relaying). SEATTLE is also available as several independent prototypes implemented by different research groups.

While we took a comprehensive approach in this dissertation, we do not claim completeness. In fact, we are aware that this is just the first step towards the wide deployment of self-configuring networks. That means there are many other types of networks upon which self-configuration can (and should) be achieved without compromising scalability and efficiency. Good examples include the Internet itself, various content-distribution networks, the networks interconnecting distributed small-scale data centers, etc. In light of the lessons learned from this dissertation, we believe the design principles and architectural primitives we proposed will benefit the development of appropriate architectures for those networks.

Bibliography

[1] ResearchAndMarket, "Demystifying Opex and Capex Budgets - Feedback from Operator Network Managers," 2007. `http://www.researchandmarkets.com/reports/448691/demystifying_opex_and_capex_budgets_feedback`.

[2] Lightreading, "Capex vs. Opex," 2002. `http://www.lightreading.com/document.asp?doc_id=22223`.

[3] W. Enck, T. Moyer, P. McDaniel, S. Sen, P. Sebos, S. Spoerel, A. Greenberg, Y. Sung, S. Rao, and W. Aiello, "Configuration Management at Massive Scale: System Design and Experience," *IEEE J. Selected Areas in Communications, Special Issue on Network Infrastructure Configuration*, April 2009.

[4] R. Mahajan, D. Wetherall, and T. Anderson, "Understanding BGP misconfigurations," in *Proc. ACM SIGCOMM*, August 2002.

[5] T. G. Griffin and G. Wilfong, "On the correctness of iBGP configuration," in *Proc. ACM SIGCOMM*, August 2002.

[6] N. Feamster and H. Balakrishnan, "Detecting BGP configuration faults with static analysis," in *Proc. USENIX Networked Systems Design and Implementation*, 2005.

[7] Z. Kerravala, "Configuration management delivers business resiliency." The Yankee Group, November 2002.

[8] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Meyers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang, "A clean slate 4D approach to network control and management," in *ACM SIGCOMM Computer Communication Review*, 2005.

[9] H. Ballani and P. Francis, "CONMan: A Step Towards Network Manageability," in *Proc. ACM SIGCOMM*, 2007.

[10] A. Greenberg, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "Towards a Next Generation Data Center Architecture: Scalability and Commoditization," in *Proc. of PRESTO Workshop at SIGCOMM*, 2008.

[11] "HP Openview." http://www.openview.hp.com/.

[12] "IBM Tivoli." http://www.ibm.com/software/tivoli/.

[13] "Microsoft Operations Manager." http://www.microsoft.com/mom/.

[14] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and J. van der Merwe, "Design and Implementation of a Routing Control Platform," in *Proc. USENIX Networked Systems Design and Implementation*, 2005.

[15] H. Yan, D. A. Maltz, T. S. E. Ng, H. Gogineni, H. Zhang, and Z. Cai, "Tesseract: A 4D network control plane," in *Proc. USENIX Networked Systems Design and Implementation*, April 2007.

[16] Y. Sung, S. Rao, G. Xie, and D. Maltz, "Towards Systematic Design of Enterprise Networks," in *Proc. ACM Conference on emerging Networking EXperiments and Technologies*, 2008.

[17] D. Farinacci, V. Fuller, D. Meyer, and D. Lewis, "Locator/ID Separation Protocol (LISP)." Internet draft, work in progress, March 2009.

[18] X. Zhang, P. Francis, J. Wang, and K. Yoshida, "Scaling IP Routing with the Core Router-Integrated Overlay," in *Proc. IEEE International Conference on Network Protocols*, November 2006.

[19] H. Ballani, P. Francis, T. Cao, and J. Wang, "Making Routers Last Longer with ViAggre," in *Proc. USENIX Networked Systems Design and Implementation*, April 2009.

[20] C. Kim, M. Caesar, A. Gerber, and J. Rexford, "Revisiting Route Caching: The World Should be Flat," in *Proc. Passive and Active Measurement*, 2009.

[21] M. Caesar, T. Condie, J. Kannan, K. Lakshminarayanan, and I. Stoica, "ROFL: Routing on Flat Labels," in *Proc. ACM SIGCOMM*, September 2006.

[22] L. De Carli and Y. Pan and A. Kumar and C. Estan and K. Sankaralingam, "Flexible Lookup Modules for Rapid Deployment of New Protocols in High-speed Routers," in *Proc. ACM SIGCOMM*, 2009. to appear.

[23] D. Pall, "Faster Packet Forwarding in a Scalable Ethernet Architecture." TR-812-08, Princeton University, January 2008. Computer Science Department Technical Report.

[24] M. Arregoces and M. Portolani, *Data Center Fundamentals*. Cisco Press, 2003.

[25] S. Halabi, *Metro Ethernet*. Cisco Press, 2003.

[26] H. Hudson, "Extending access to the digital economy to rural and developing regions." Understanding the Digital Economy, The MIT Press, Cambridge, MA, 2002.

[27] A. Gupta, B. Liskov, and R. Rodrigues, "Efficient routing for peer-to-peer overlays," in *Proc. USENIX Networked Systems Design and Implementation*, March 2004.

[28] W. Aiello, C. Kalmanek, P. McDaniel, S. Sen, O. Spatscheck, and J. van der Merwe, "Analysis of communities of interest in data networks," in *Proc. Passive and Active Measurement*, March 2005.

[29] R. Perlman, "Rbridges: Transparent routing," in *Proc. IEEE INFOCOM*, March 2004.

[30] "IETF TRILL Working Group." http://tools.ietf.org/wg/trill/, 2009.

[31] A. Myers, E. Ng, and H. Zhang, "Rethinking the service model: scaling Ethernet to a million nodes," in *Proc. ACM Workshop on Hot Topics in Networks*, November 2004.

[32] S. Sharma, K. Gopalan, S. Nanda, and T. Chiueh, "Viking: A multi-spanning-tree Ethernet architecture for metropolitan area and cluster networks," in *Proc. IEEE INFOCOM*, March 2004.

[33] T. Rodeheffer, C. Thekkath, and D. Anderson, "SmartBridge: A scalable bridge architecture," in *Proc. ACM SIGCOMM*, August 2000.

[34] C. Kim and J. Rexford, "Revisiting Ethernet: Plug-and-play made scalable and efficient," in *Proc. IEEE Workshop on Local and Metropolitan Area Networks*, June 2007.

[35] S. Ray, R. A. Guerin, and R. Sofia, "A distributed hash table based address resolution scheme for large-scale Ethernet networks," in *Proc. IEEE International Conference on Communications*, June 2007.

[36] B. Ford, "Unmanaged Internet Protocol: Taming the edge network management crisis," in *Proc. ACM Workshop on Hot Topics in Networks*, November 2003.

[37] M. Caesar, M. Castro, E. Nightingale, A. Rowstron, and G. O'Shea, "Virtual Ring Routing: Network routing inspired by DHTs," in *Proc. ACM SIGCOMM*, September 2006.

[38] R. Perlman, *Interconnections: Bridges, Routers, Switches, and Internetworking Protocols*. Addison-Wesley, second ed., 1999.

[39] M. Allman, V. Paxson, and J. Terrell, "A brief history of scanning," in *Proc. Internet Measurement Conference*, October 2007.

[40] Dartmouth Institute for Security Technology Studies, "Problems with broadcasts." `http://www.ists.dartmouth.edu/classroom/crs/arp_broadcast.php`.

[41] R. King, "Traffic management tools fight growing pains," June 2004. `http://www.thewhir.com/features/traffic-management.cfm`.

[42] "IEEE Std 802.1Q - 2005, IEEE Standard for Local and Metropolitan Area Network, Virtual Bridged Local Area Networks," 2005.

[43] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana, "Internet indirection infrastructure," in *Proc. ACM SIGCOMM*, August 2002.

[44] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-area cooperative storage with CFS," in *Proc. ACM Symposium on Operating Systems Principles*, October 2001.

[45] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web," in *Proc. ACM Symposium on Theory of Computing*, 1997.

[46] B. Godfrey, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica, "Load balancing in dynamic structured P2P systems," in *Proc. IEEE INFOCOM*, March 2003.

[47] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley, "The design and implementation of an intentional naming system," in *Proc. ACM Symposium on Operating Systems Principles*, December 1999.

[48] J. Moy, *OSPF: Anatomy of an Internet Routing Protocol.* Addison-Wesley, 1998.

[49] R. Hinden, "Virtual Router Redundancy Protocol (VRRP)." RFC 3768, April 2004.

[50] Ethereal, "Gratuitous ARP." `http://wiki.ethereal.com/Gratuitous_ARP`.

[51] R. Droms, "Dynamic Host Configuration Protocol." Request for Comments 2131, March 1997.

[52] C. Tengi, J. Roberts, J. Crouthamel, C. Miller, and C. Sanchez, "autoMAC: A Tool for Automating Network Moves, Adds, and Changes," in *Proc. USENIX Large Installation System Administration Conference*, 2004.

[53] D. Hucaby and S. McQuerry, *Cisco Field Manual: Catalyst Switch Configuration*. Cisco Press, 2002.

[54] R. Pang, M. Allman, M. Bennett, J. Lee, V. Paxson, and B. Tierney, "A first look at modern enterprise traffic," in *Proc. Internet Measurement Conference*, October 2005.

[55] N. Spring, R. Mahajan, and D. Wetherall, "Measuring ISP topologies with Rocketfuel," in *Proc. ACM SIGCOMM*, August 2002.

[56] "Skitter." `http://www.caida.org/tools/measurement/skitter`.

[57] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. Kaashoek, "The Click modular router," in *ACM Trans. Computer Systems*, August 2000.

[58] M. Handley, E. Kohler, A. Ghosh, O. Hodson, and P. Radoslavov, "Designing extensible IP router software," in *Proc. USENIX Networked Systems Design and Implementation*, May 2005.

[59] J. Hamilton, "Cooperative Expendable Micro-Slice Servers (CEMS): Low Cost, Low Power Servers for Internet-Scale Services," in *Proc. of Conference on Innovative Data Systems Research*, 2009.

[60] K. Claffy, H. Braun, and G. Polyzos, "A parameterizable methodology for Internet traffic flow profiling," *IEEE J. Selected Areas in Communications*, vol. 13, 1995.

[61] Y. Zhang and Z. Ge, "Finding critical traffic matrices," in *Proc. IEEE International Conference on Dependable Systems and Networks*, June 2005.

[62] R. Zhang-Shen and N. McKeown, "Designing a Predictable Internet Backbone Network," in *Proc. ACM Workshop on Hot Topics in Networks*, 2004.

[63] M. Kodialam, T. V. Lakshman, and S. Sengupta, "Efficient and Robust Routing of Highly Variable Traffic," in *Proc. ACM Workshop on Hot Topics in Networks*, 2004.

[64] N. G. Duffield, P. Goyal, A. G. Greenberg, P. P. Mishra, K. K. Ramakrishnan, and J. E. van der Merwe, "A flexible model for resource management in virtual private network," in *Proc. ACM SIGCOMM*, 1999.

[65] W. J. Dally and B. Towles, *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers, 2004.

[66] D. C. Plummer, "An ethernet address resolution protocol – or – converting network protocol addresses." RFC 826, November 1982.

[67] L. Lamport, "The part-time parliament," *ACM Trans. Computer Systems*, vol. 16, pp. 133–169, 1998.

[68] M. Mathis, J. Heffner, and R. Raghunarayan, "TCP Extended Statistics MIB." RFC 4898, 2007.

[69] R. Jain, *The Art of Computer Systems Performance Analysis*. John Wiley and Sons, Inc., 1991.

[70] M. Handley, S. Floyd, J. Padhye, and J. Widmer, "Tcp friendly rate control (tfrc): Protocol specification." RFC 3348, 2003.

[71] J. Hamilton, "An architecture for modular data centers," in *Third Biennial Conference on Innovative Data Systems Research*, 2007.

[72] Cisco, "Data Center Ethernet." http://www.cisco.com/go/dce.

[73] C. Kim, M. Caesar, and J. Rexford, "Floodless in SEATTLE: A Scalable Ethernet Architecture for Large Enterprises," in *Proc. ACM SIGCOMM*, August 2008.

[74] C. Guo, H. Wu, K. Tan, L. Shiy, Y. Zhang, and S. Luz, "DCell: A Scalable and Fault-Tolerant Network Structure for Data Centers," in *Proc. ACM SIGCOMM*, 2008.

[75] M. Al-Fares, A. Loukissas, and A. Vahdat, "A Scalable, Commodity Data Center Network Architecture," in *Proc. ACM SIGCOMM*, 2008.

[76] C. Chang, D. Lee, and Y. Jou, "Load balanced Birkhoff-von Neumann switches, part I: one-stage buffering," *IEEE HPSR*, 2001.

[77] E. Rosen and Y. Rekhter, "BGP/MPLS IP Virtual Private Networks." RFC 4364, February 2006.

[78] IDC, "U.S. IP VPN services 2006-2010 forecast." `http://www.idc.com/getdoc.jsp?containerId=201682`.

[79] S. Raghunath, K. K. Ramakrishnan, S. Kalyanaraman, and C. Chase, "Measurement Based Characterization and Provisioning of IP VPNs," in *Proc. Internet Measurement Conference*, October 2004.

[80] S. Raghunath, S. Kalyanaraman, and K. K. Ramakrishnan, "Trade-offs in Resource Management for Virtual Private Networks," in *Proc. IEEE INFOCOM*, March 2005.

[81] Y. Rekhter, T. Li, and S. Hares, "A Border Gateway Protocol (BGP-4)," *RFC 4271*, January 2006.

[82] T. Bates, R. Chandra, D. Katz, and Y. Rekhter, "Multiprotocol Extensions for BGP-4." RFC 2283, 1998.

[83] Cisco, "Cisco Line Cards, engine 0,1,2,3,4." `http://www.cisco.com/en/US/products/hw/routers/ps167/products_tech_note09186a00801e1dbe.shtml`.

[84] B. Claise, "Cisco Systems NetFlow Services Export Version 9." Request for Comments 3954, October 2004.

[85] C. Kim, A. Gerber, C. Lund, D. Pei, and S. Sen, "Scalable VPN Routing via Relaying." Technical Report, November 2007. AT&T TD-794M29.

[86] R. Karp, "Reducibility among combinatorial problems," in *Complexity of Computer Computations*, pp. 85–103, 1972.

[87] W. Enck, P. McDaniel, S. Sen, P. Sebos, S. Spoerrel, A. Greenberg, S. Rao, and W. Aiello, "Configuration Management at a Massive Scale: System Design and Experience," in *Proc. USENIX Annual Technical Conference*, 2007.

[88] B. Ford, "Unmanaged Internet Protocol: Taming the edge network management crisis," in *ACM SIGCOMM Computer Communication Review*, 2004.