UC San Diego
UC San Diego Electronic Theses and Dissertations

Title
Scalable, efficient, and fault-tolerant data center networking

Permalink
https://escholarship.org/uc/item/83d5p6vm

Authors
Walraed-Sullivan, Meg

Walraed-Sullivan, Meg

Publication Date
2012

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA, SAN DIEGO

Scalable, Efficient, and Fault-Tolerant Data Center Networking

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science

by

Meg Walraed-Sullivan

Committee in charge:

Professor Amin Vahdat, Chair
Professor Keith Marzullo, Co-Chair
Professor Sujit Dey
Professor Tara Javidi
Professor Alex Snoeren

2012

Copyright
Meg Walraed-Sullivan, 2012
All rights reserved.

The dissertation of Meg Walraed-Sullivan is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

____

____

____

____
Co-Chair

____
Chair

University of California, San Diego

2012

DEDICATION

For Sean, without whom this dissertation would not exist.

EPIGRAPH

The only way of finding the limits of the possible
is by going beyond them into the impossible.
—Arthur C. Clarke

If you can't explain it simply,
you don't understand it well enough.
—Albert Einstein

Any intelligent fool can make things bigger,
more complex, and more violent.
It takes a touch of genius – and a lot of courage –
to move in the opposite direction.
—Albert Einstein

TABLE OF CONTENTS

Signature Page iii

Dedication iv

Epigraph v

Table of Contents vi

List of Figures x

List of Tables xii

List of Code Listings xiii

List of Common Systems and Distributed Computing Acronyms xiv

List of Acronyms Introduced in this Dissertation xvi

Acknowledgements xvii

Vita xix

Abstract of the Dissertation xx

Chapter 1 Introduction 1
- 1.1 Data Center Networking Today 2
- 1.2 Challenges in Data Center Networking 3
  - 1.2.1 Scalable, Fault-Tolerant Topologies 4
  - 1.2.2 Scalable Addressing and Communication 6
  - 1.2.3 Formalizing Label Assignment 8
- 1.3 Hypothesis 9
- 1.4 Contributions 9
  - 1.4.1 Aspen Trees: Tuning Scalability and Fault Tolerance 9
  - 1.4.2 ALIAS: Scalable Addressing and Communication . 10
  - 1.4.3 The Decider/Chooser Protocol in ALIAS 11
- 1.5 Organization 11
- 1.6 Acknowledgment 12

Chapter 2 Background: Data Center Networks 13
- 2.1 Topology 13
- 2.2 Addressing and Communication 16
- 2.3 Fault Tolerance 18

2.4 Management 19
2.5 Nomenclature 19
2.6 Acknowledgment 23

Chapter 3 Scalability versus Fault Tolerance in Aspen Trees 24
3.1 Failures in Traditional Fat Trees 25
3.2 Designing Aspen Trees 27
3.2.1 Assumptions 29
3.2.2 Aspen Tree Generation 29
3.3 Aspen Tree Properties 33
3.3.1 Fault Tolerance 34
3.3.2 Number of Switches Needed 34
3.3.3 Number of Hosts Supported 36
3.3.4 Hierarchical Aggregation 37
3.4 Leveraging Fault Tolerance: Routing Around Failures 38
3.4.1 Communication Protocol Overview 39
3.4.2 Propagating Failure Notifications 40
3.5 Wiring the Tree: Striping 42
3.6 Evaluation 44
3.6.1 Convergence versus Scalability 44
3.6.2 Recommended Aspen Trees 48
3.7 Related Work 50
3.7.1 Alternative Routing Techniques 50
3.7.2 Backup Paths 51
3.7.3 High Performance Computing Topologies 52
3.8 Summary 53
3.9 Acknowledgment 53

Chapter 4 ALIAS: Scalable, Decentralized Label Assignment for Data Centers 54
4.1 ALIAS 55
4.1.1 Environment 56
4.1.2 Protocol Overview 57
4.1.3 Multi-Path Support 59
4.2 Protocol 60
4.2.1 Level Assignment 60
4.2.2 Label Assignment 62
4.2.3 Relabeling 67
4.2.4 M-Graphs 69
4.3 Communication 71
4.3.1 Routing 71
4.3.2 Forwarding 72
4.3.3 End-to-End Communication 73
4.3.4 Peer Links 75

4.3.5 Switch Modifications 77
4.4 Implementation 77
4.4.1 Mace Implementation 78
4.4.2 NetFPGA Testbed Implementation 79
4.5 Evaluation 79
4.5.1 Storage Requirements 80
4.5.2 Control Overhead 80
4.5.3 Compact Forwarding Tables 82
4.5.4 Convergence Time 84
4.6 Related Work 86
4.7 Summary 87
4.8 Acknowledgment 87

Chapter 5 A Randomized Algorithm for Label Assignment in Dynamic Networks 88
5.1 ALIAS Details 89
5.2 The Label Selection Problem 90
5.2.1 The Label Selection Problem with Consensus 92
5.3 The Decider/Chooser Protocol 94
5.3.1 Bounding the Channels 98
5.4 Analysis of the Decider/Chooser Protocol 102
5.4.1 Proof of Correctness of DCP 102
5.4.2 Model Checking DCP 104
5.5 Performance of the Decider/Chooser Protocol 105
5.5.1 Analyzing DCP Performance 105
5.5.2 Simulating DCP Performance 108
5.6 DCP in Data Center Labeling 109
5.6.1 Distributing the Chooser 110
5.6.2 The Decider/Chooser Protocol in ALIAS 112
5.6.3 Eliminating M-Graphs in ALIAS 115
5.7 From DCP to ALIAS Coordinate Selection 117
5.7.1 ALIAS and DCP Review 117
5.7.2 Computing Hypernodes 118
5.7.3 L_1-Coordinate Assignment: Basic DCP 122
5.7.4 L_2-coordinate Assignment: Distributed DCP 128
5.7.5 Derivation Summary 135
5.8 The Decider/Chooser Protocol in Wireless Networks 137
5.9 Related Work 138
5.10 Summary 139
5.11 Acknowledgment 140

Chapter 6 Single Label Selection for ALIAS Hosts 141
6.1 Background and Environment 142
6.2 ALIAS Protocol Modifications 145
6.2.1 Selecting a Single Label 146
6.2.2 Propagating Single Label Selections 148
6.2.3 Combining Single Label Forwarding Table Entries . 150
6.3 Effects of Single Label Selection on ALIAS 151
6.3.1 Effect on Multi-Path Support 151
6.3.2 Effect on Peer Link Usage 154
6.3.3 Effect on Reactivity to Topology Dynamics 156
6.4 Building Forwarding Tables with Single Label Selection . . 157
6.4.1 Super Tables 158
6.4.2 Creating Forwarding Tables from Super Tables . . . 167
6.5 Evaluation 167
6.6 Summary 173

Chapter 7 Conclusions and Future Work 175
7.1 Summary 175
7.2 Open Problems and Future Work 177
7.2.1 Data Center Network Topologies 178
7.2.2 Scalable Communication 179
7.2.3 Formalizing Data Center Protocols 180
7.3 Acknowledgment 180

Bibliography 182

LIST OF FIGURES

Figure 1.1: Data Center Topology with Address Assignments 4

Figure 2.1: Multi-Rooted Tree Topology 14
Figure 2.2: Fat Tree Topology 15

Figure 3.1: Packet Travel in a 4-Level, 4-Port Fat Tree 26
Figure 3.2: Modifying a 3-Level, 4-Port Fat Tree to Have 1-Fault Tolerance at L_3 28
Figure 3.3: Examples of 4-Level, 6-Port Aspen Trees 35
Figure 3.4: 4-Level, 4-Port Aspen Tree with FTV= <0,1,0> 39
Figure 3.5: 4-Level, 4-Port Aspen Tree with FTV= <1,0,0> 42
Figure 3.6: Striping Examples for a 3-Level, 4-Port Tree 43
Figure 3.7: Host Removal and Convergence Time vs. Fault Tolerance 46
Figure 3.8: Convergence vs. Scalability for 4 and 5-Level, 16-Port Aspen Trees 47
Figure 3.9: Convergence vs. Scalability for 3-Level, 32 and 64-Port Aspen Trees 49

Figure 4.1: Sample Multi-Rooted Tree Topology 56
Figure 4.2: ALIAS Applied to a Sample Topology: Level and Label Assignments 58
Figure 4.3: ALIAS Hypernodes 63
Figure 4.4: Decider/Chooser Abstraction in ALIAS 65
Figure 4.5: Label Assignment: L_2-Coordinates 67
Figure 4.6: Relabeling Example 68
Figure 4.7: Example M-Graph 70
Figure 4.8: Example of Forwarding Table Entries 72
Figure 4.9: End-to-End Communication 74
Figure 4.10: Peer Link Tradeoff 76
Figure 4.11: ALIAS Architecture 78
Figure 4.12: Storage Overhead for 3-Level, 128-Port Tree 80
Figure 4.13: CDF of Startup Convergence Times 84
Figure 4.14: Relabeling Convergence Times 85

Figure 5.1: ALIAS Topology 90
Figure 5.2: Label Selection Problem Topology 90
Figure 5.3: Choosers and Shared Deciders 91
Figure 5.4: Stability Example 92
Figure 5.5: Example Consensus Scenarios 93
Figure 5.6: $P(q,32,L)$ with $L = 32,64,128$ 107
Figure 5.7: Distributed Chooser 111
Figure 5.8: Multi-Rooted Tree Topology 113
Figure 5.9: Simple DCP in ALIAS 114
Figure 5.10: Assigning L_2-Coordinates using Distributed Choosers 115
Figure 5.11: Example M-Graph 116
Figure 5.12: Two Options for L_1-Coordinate Selection 124

Figure 5.13: Multiple AP Example 138

Figure 6.1: Fat Tree Topology 142
Figure 6.2: Labeling Conventions 143
Figure 6.3: Topology with Multiple Labels per Host 145
Figure 6.4: Combining Optimal Label Forwarding Entries 150
Figure 6.5: Multi-Path Support with Optimal Label Selection 152
Figure 6.6: Peer Links and Optimal Label Selection 154
Figure 6.7: Connectivity-Providing Peer Links and Optimal Label Selection . . 155
Figure 6.8: Optimal Label Selection and Topology Changes 157
Figure 6.9: Super Table Example Topology 159
Figure 6.10: Initial Super Table for S_{64} 159
Figure 6.11: Consolidated Super Table for S_{64} 160
Figure 6.12: Topology for Super Table Entry Examples 161
Figure 6.13: Forwarding Table Sizes for n=3, k=4 169
Figure 6.14: Forwarding Table Sizes for n=3, k=8 169
Figure 6.15: Forwarding Table Sizes for n=3, k=16 169
Figure 6.16: Forwarding Table Sizes for n=3, k=32 170
Figure 6.17: Forwarding Table Sizes for n=3, k=64 170
Figure 6.18: Forwarding Table Sizes for n=4, k=4 170
Figure 6.19: Forwarding Table Sizes for n=4, k=8 170
Figure 6.20: Forwarding Table Sizes for n=4, k=16 171
Figure 6.21: Forwarding Table Sizes for n=5, k=4 171
Figure 6.22: Forwarding Table Sizes for n=5, k=8 171
Figure 6.23: Optimal Label Entries as a Percentage of Regular Entries 173
Figure 6.24: Regular Entries vs. Number of Ports Per Switch 173

LIST OF TABLES

Table 2.1: Symbols Commonly Used throughout Dissertation 20
Table 2.2: Symbols Specific to Chapter 3 21
Table 2.3: Symbols Specific to Chapter 4 21
Table 2.4: Symbols Specific to Chapter 5 22
Table 2.5: Symbols Specific to Chapter 6 22

Table 3.1: Topology Enumeration with $k = 6$ and $n = 4$ 32
Table 3.2: Replacing S with Numerical Values 33

Table 4.1: Relabeling Cases 69
Table 4.2: Level/Coordinate Assignment Overhead 81
Table 4.3: Forwarding Entries Per Switch 83

Table 5.1: Convergence Time of DCP 109

Table 6.1: Summary of Super Table Entries 162

LIST OF CODE LISTINGS

Listing 3.1: Aspen Tree Generation Algorithm 31

Listing 4.1: ALIAS local state 61

Listing 5.1: Decider Algorithm 95
Listing 5.2: Chooser Channel Predicates and Routines (Unbounded Channels) . 96
Listing 5.3: Chooser Algorithm: Actions and State (Unbounded Channels) . . . 97
Listing 5.4: Chooser Channel Predicates and Routines (Bounded Channels) . . 100
Listing 5.5: Chooser Algorithm: Actions and State (Bounded Channels) 101
Listing 5.6: Decider Algorithm (Derivation Version) 119
Listing 5.7: Chooser Algorithm: Actions and State (Derivation Version) 120
Listing 5.8: Chooser Channel Predicates and Routines (Derivation Version) . . 121
Listing 5.9: Hypernode Computation: L_2 Switches 122
Listing 5.10: Hypernode Computation: L_1 Switches 122
Listing 5.11: Chooser Algorithm: Actions and State (Multi-HN Refinement 1) . 125
Listing 5.12: Chooser Algorithm: Actions and State (Multi-HN Refinement 2) . 127
Listing 5.13: Chooser Algorithm: Actions and State (Representative L_1 Switch) . 129
Listing 5.14: Chooser Algorithm: Actions and State (L_2 Relays) 131
Listing 5.15: Chooser Algorithm: Actions and State (All L_1 Switches) 131
Listing 5.16: Chooser Channel Predicates and Routines 1 (Distributed Chooser) . 133
Listing 5.17: Chooser Channel Predicates and Routines 2 (Distributed Chooser) . 134
Listing 5.18: Decider Algorithm (Distributed Chooser) 136

LIST OF COMMON SYSTEMS AND DISTRIBUTED COMPUTING ACRONYMS

AP Access Point

ARP Address Resolution Protocol

CPU Central Processing Unit

DAC Data center Address Configuration

DDC Data-Driven Network Connectivity

DHCP Dynamic Host Configuration Protocol

DRAM Dynamic Random-Access Memory

ECMP Equal-Cost Multi-Path Routing

FFR Fast Failure Recovery

FTE Forwarding Table Entry

GCP Graph Coloring Problem

IP Internet Protocol

IS-IS Intermediate System to Intermediate System

LDP (PortLand's) Location Discovery Protocol

MAC Media Access Control

MPTCP Multi-Path Transmission Control Protocol

OSPF Open Shortest Path First

RAM Random-Access Memory

SDN Software-Defined Networking

SEATTLE Scalable Ethernet Architecture for Large Enterprises

TCP Transmission Control Protocol

TRILL Transparent Interconnection of Lots of Links

UID Unique Identifier

VM Virtual Machine

LIST OF ACRONYMS INTRODUCED IN THIS DISSERTATION

CV Connectivity-Value

DC Don't Care value

DCA Decider/Chooser Abstraction

DCC Duplicate Connection Count

DCP Decider/Chooser Protocol

FTV Fault Tolerance Vector

HN Hypernode

LNV L_n-Value

LSP Label Selection Problem

TVM Topology View Message

ACKNOWLEDGEMENTS

It is hard to imagine a better pair of advisors than Professors Amin Vahdat and Keith Marzullo.

Through the years, Professor Vahdat has not only been willing to share his seemingly infinite set of research ideas, he has been unfailingly supportive when I've wanted to explore a new direction or idea. I've been constantly amazed by his ability to turn a vague sketch of a potential idea into a coherent, motivated research direction. One of the most important lessons I've learned from Professor Vahdat is how to see the forest through the trees. Another is that 6pt font is simply not acceptable.

Professor Marzullo introduced me to the theoretical side of Systems research. Exploring both the theoretical and applied aspects of each research question has been one of my favorite parts of this experience and I am forever indebted to him for this. I am continuously impressed by his patience and his ability for careful and precise thought, whether in the context of understanding a new algorithm, writing a 30 page protocol derivation, or somehow extracting a proof from my incoherent ramblings.

I would like to thank Dr. Doug Terry for his mentorship during my internships at Microsoft Research and thereafter. It was through his advice and teaching that I initially discovered my interest in distributed systems research. I am also grateful for the mentorship of Professor E. Gun Sirer. My decision to continue the study of computer science was largely inspired by his guidance. I thank Dr. Bruce Land for teaching me how to build systems and for taking engineering out of the text books and into the lab.

During my time at UCSD, I have been fortunate to work with some of the brightest people I have ever met. I would like to thank my fantastic co-authors: Radhika Niranjan Msyore, Malveeka Tewari and Ying Zhang. I have learned from each of them. I also wish to thank friends and collaborators at UCSD: Mohammad Al-Fares, Ryan Braud, Stephen Checkoway, Mike Conley, Nathan Farrington, Rishi Kapoor, Chris Kanich, Chip Killian, Harsha Madhyastha, John McCullough, Hein Meling, Marti Motoyama, Sivasankar Radhakrishnan, Alex Rasmussen, Cynthia Taylor and Qing Zhang. I thank the SysNet faculty, especially Dr. Kirill Levchenko, Dr. George Porter, Professor Stefan Savage, Professor Alex Snoeren, Professor Geoff Voelker and Dr. Ken Yocum, for their mentorship and guidance.

Finally, and most importantly, I am grateful to my family for their unwavering support and encouragement throughout this experience; Sean Keller for for always having my back and being my biggest champion; my parents, Susan Walraed and J. Thomas Sullivan, for years of support both prior to and during my time at UCSD – without them I would not have made it to day 1 of this program; and Kyle Walraed-Sullivan for his encouragement and for always being willing to read a paper submission (or dissertation) at a moment's notice.

I also thank Monster, Tiger and Rascal for inciting much needed laughs and smiles and for always saving their catastrophes until immediately before paper deadlines.

Chapters 1, 2, 3, and 7, in part, contain material submitted for publication as "Scalability vs. Fault Tolerance in Apsen Trees." Walraed-Sullivan, Meg; Vahdat, Amin; Marzullo, Keith. The dissertation author was the primary investigator and author of this paper.

Chapters 1, 2, 4, and 7, in part, contain material as it appears in the Proceedings of the ACM Symposium on Cloud Computing (SOCC) 2011. Walraed-Sullivan, Meg; Niranjan Mysore, Radhika; Tewari, Malveeka; Zhang, Ying; Marzullo, Keith; Vahdat, Amin. The dissertation author was the primary investigator and author of this paper.

Chapters 1, 2, 5, and 7, in part, contain material as it appears in the Proceedings of the 25th International Symposium on Distributed Computing (DISC) 2011. Walraed-Sullivan, Meg; Niranjan Mysore, Radhika; Tewari, Malveeka; Zhang, Ying; Marzullo, Keith; Vahdat, Amin. The dissertation author was the primary investigator and author of this paper.

Chapters 1, 2, 5, and 7, in part, contain material submitted for publication as "A Randomized Algorithm for Label Assignment in Dynamic Networks." Walraed-Sullivan, Meg; Niranjan Mysore, Radhika; Marzullo, Keith; Vahdat, Amin. The dissertation author was the primary investigator and author of this paper.

VITA

2003	B. S. in Electrical and Computer Engineering Cornell University
2004	M. Eng. in Electrical and Computer Engineering Cornell University
2012	Ph. D. in Computer Science University of California, San Diego

PUBLICATIONS

"ALIAS: Scalable, Decentralized Label Assignment for Data Centers." Meg Walraed-Sullivan, Radhika Niranjan Mysore, Malveeka Tewari, Ying Zhang, Keith Marzullo, Amin Vahdat. *Proceedings of the ACM Symposium on Cloud Computing (**SOCC**)*, 2011.

"Brief Announcement: A Randomized Algorithm for Label Assignment in Dynamic Networks'." Meg Walraed-Sullivan, Radhika Niranjan Mysore, Keith Marzullo, Amin Vahdat. *Proceedings of the 25th International Symposium on DIStributed Computing (**DISC**)*, 2011.

"A Randomized Algorithm for Label Assignment in Dynamic Networks'." Meg Walraed-Sullivan, Radhika Niranjan Mysore, Keith Marzullo, Amin Vahdat. *Technical Report*, 2011.

"Cimbiosys: A Platform for Content-based Partial Replication." Venugopalan Ramasubramanian, Thomas L. Rodeheffer, Douglas B. Terry, Meg Walraed-Sullivan, Ted Wobber, Catherine C. Marshall, Amin Vahdat. *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (**NSDI**)*, 2009.

"Fast encounter-based synchronization for mobile devices'." Daniel Peek, Venugopalan Ramasubramanian, Tom L. Rodeheffer, Douglas B. Terry, Meg Walraed-Sullivan, and Ted Wobber. *Proceedings of the International Conference on Digital Information Management (**ICDIM**)*, 2007.

ABSTRACT OF THE DISSERTATION

Scalable, Efficient, and Fault-Tolerant Data Center Networking

by

Meg Walraed-Sullivan

Doctor of Philosophy in Computer Science

University of California, San Diego, 2012

Professor Amin Vahdat, Chair
Professor Keith Marzullo, Co-Chair

The advent of cloud computing and the expectation of anytime availability of user data and services have brought data center design to the forefront of computer science research. Modern data centers can be massive in size, consisting of hundreds of thousands of servers and millions of virtualized end hosts. At this scale and complexity, the underlying network becomes central to data center scalability, efficiency, availability and fault tolerance.

Given the scale of today's data center networks, operators typically turn to symmetric, highly structured network topologies, sacrificing flexibility for relative simplicity. These topologies tend to have an "all or nothing" tradeoff between fault tolerance and scalability. Over these topologies, data center operators often run protocols bor-

rowed from the Internet, an environment that is drastically different from that of the data center. Because these protocols have not been built for the data center, they can operate and interact in unexpected and undesirable ways. Moreover, they are generally vetted by virtue of having survived in the Internet, rather than by formal reasoning. This makes the management burden associated with configuration, maintenance and error diagnosis for these protocols substantial, leading to compromised efficiency and availability.

The first contribution of this dissertation is the introduction of a new class of network topologies called Aspen trees. Aspen trees provide the high throughput and path multiplicity of current data center network topologies while also allowing a network operator to select a particular point on the scalability versus fault tolerance spectrum. This addresses the challenge of supporting simultaneous scalability and fault tolerance in data center networks. Next, the challenge of providing scalable and efficient communication is addressed with the design of ALIAS, a protocol for scalable, automatic and decentralized addressing and communication in the data center. Finally, this dissertation presents a formalization and proof of correctness of the fundamental building block of ALIAS, thus enabling feasible configuration and maintenance of ALIAS in the data center. This combination of tunable topology structure and tailored communication protocols enables scalable, efficient and fault-tolerant data center communication.

Chapter 1

Introduction

Today, the term “cloud computing” is a household expression. Users perform web searches that leverage a data center hosted in the cloud, they store their email and documents in the cloud, and they stream videos from the cloud to their mobile devices. The advent of cloud computing along with users’ expectations that services will be responsive, robust and available anytime has made the data center a key area for computer science research today. Companies, universities and other large organizations are building massive data centers to provide new services and to develop new technologies, and this rapid development pace shows no sign of slowing in the near future. In fact, recent trends signal that the market for data center construction will nearly double within the next ten years [12].

A modern data center contains a group of end hosts that communicate and cooperate across an interconnect of network elements —switches and routers— to perform shared tasks. These tasks might comprise the back-end for a company that provides one or more user-facing services, such as Google’s search engine, email and map services [27] or Facebook’s social networking site [24]. Or, a cloud provider might build a data center for the purpose of renting out nodes to service providers, as is the case with Amazon’s EC2 Platform [6] and Microsoft’s Windows Azure [52].

An important component of the data center is its network. A data center network is comprised of the switches and routers, wires, topology layout and network protocols that together provide an interconnect for end host communication. The characteristics of this network can be crucial to a data center’s success [2].

1.1 Data Center Networking Today

Today's data centers have a number of characteristics that give networking researchers a unique set of challenges to face. First and foremost, they can be massive in size; a modern data center can connect up to hundreds of thousands of end hosts via an interconnect of tens of thousands of switches and routers. For instance, a recent article estimates the largest data center of Amazon's EC2 platform to contain upwards of 300,000 hosts [49]. Therefore, scalability is a concern at all levels of the design, including physical layout, hardware selection and protocol design. Not only is the current scalability of the topology important, it is also imperative to keep in mind a data center's ability to scale out in order to match future needs. As cloud computing becomes more popular, service providers increasingly leverage the elasticity and the in-place services offered by data center providers in order to bring new products to market quickly [7]. Therefore, data centers have to accommodate massive scale now and also be amenable to significant growth in the future.

An important piece of a data center network is its physical topology, the interconnection of its switches and hosts. A variety of different means for interconnecting massive numbers of end hosts have been proposed. Some layouts are based on fairly regular structures, such as fat trees [4, 56], hypercube-like designs [3, 14, 29] or other regular, symmetric constructs [28, 30]. Others designs allow for more varied topologies and introduce more complicated protocols to accommodate topology discovery and host connectivity in the face of topological asymmetry [16, 42, 67]. Regardless of the layout selected, it is imperative that a data center designer utilize care in actually building and wiring the network, as erroneous cablings and mis-configuration can have disastrous effects. It is also important to have methods in place to discover and locate wiring errors and equipment failures when they occur. At the scale of the data center, this can create a need for specialized protocols even just to present a user-readable view of the current topology.

Another key characteristic of the modern data center is that each falls within a single administrative domain. That is, one entity is responsible for all decisions regarding the data center's hardware (e.g. network elements, end hosts and storage elements), software and firmware (e.g. communication protocols, distributed applications and oper-

ating systems) and physical layout. Given this, it is possible to start entirely from scratch when designing a new data center; an organization can build brand new protocols that are perfectly tailored to its needs. This can be beneficial in that it provides considerable flexibility to a data center owner. In fact, more recently, data center operators have been known to build their own hardware in addition to software, custom operating systems and networking protocols [23]. However, this flexibility can be a drawback, as not every data center owner is equipped to build everything from scratch.

To avoid building custom protocols, data center operators frequently borrow existing protocols from other types of networks, such as the Internet. An example of this is Ethernet, which for years has been a standard for data center networking. Another example is IP, which is used in recent enterprise network designs such as SEATTLE [42] and VL2 [28]. A difficulty with borrowing protocols from other types of networks is that the protocols may have been designed for a fairly different environment than that of the data center. This is compounded by the fact that an ad hoc set of protocols borrowed from various different networks may exhibit odd or unexpected interactions. The difficulty of simply borrowing existing protocols and grouping them together in a "plug-and-play" manner, along with the fact that it is impossible to expect every data center operator to design all network protocols from scratch, leads to a unique challenge for networking researchers today. It is time to step back and consider the axes along which data centers vary, and to develop data center networking protocols that meet the needs of a variety of different usage models while allowing for the ability to tune these protocols to suit a particular situation.

1.2 Challenges in Data Center Networking

Modern data center networks are often structured as indirect hierarchical topologies [66], in which servers connect to the leaves of a multi-stage switch fabric. Such networks can support hundreds of thousands of servers (and millions of virtual machines) with tens of thousands of switches [34]. The enormous size of these networks leads to a number of challenges in data center design.

In this dissertation, we consider the issues of designing a scalable and fault-tolerant data center network topology, providing scalable and efficient communication protocols, and formalizing these protocols in order to reason about their correctness and performance. Figure 1.1 shows an example of a scalable, fault-tolerant network topology. Each switch and host in the topology has been assigned an address by our provably correct and efficient communication protocols.

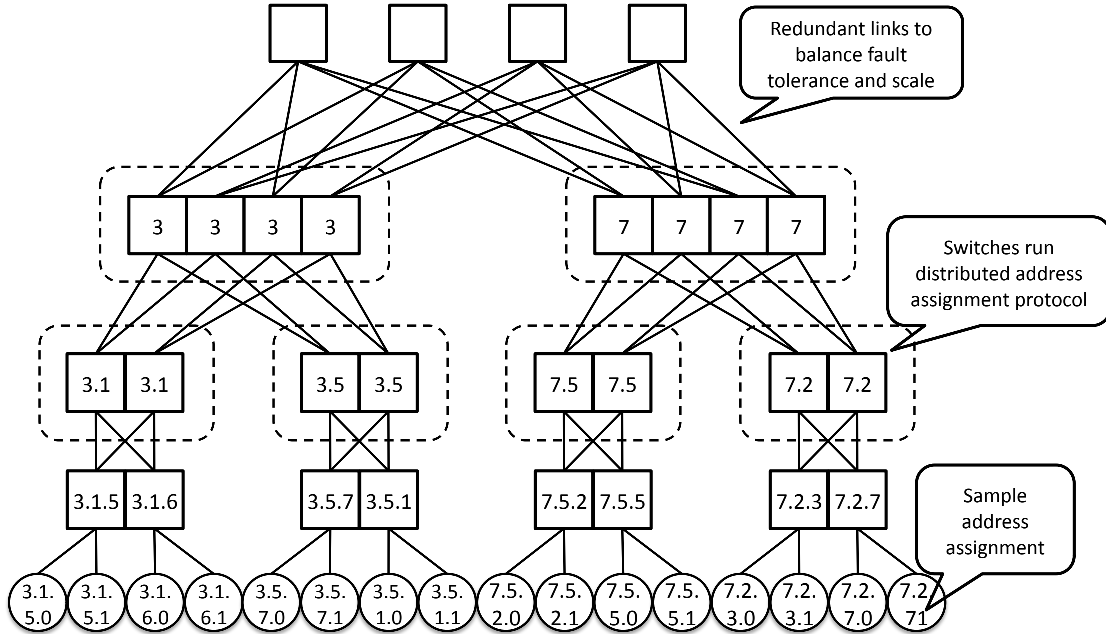


Figure 1.1: Data Center Topology with Address Assignments

1.2.1 Scalable, Fault-Tolerant Topologies

One common topology for data center interconnects is a fat tree, or Clos network [4, 19, 47, 56]. The popularity of this topology is in part due of the fat tree's support for full bisection bandwidth. In our experience, a key difficulty in the data center is handling faults in these hierarchical network fabrics.

Despite the high path multiplicity between any pair of end hosts in a traditionally defined fat tree, a single link failure can temporarily cause the loss of all packets destined to a particular set of end hosts, thus effectively disconnecting a portion of the network.

For instance, one link failure at the lowest level of a 3-level, 64-port fat tree tree can disconnect 32 hosts while a failure at the top level can affect as many as 1,024, or 1.5%, of the topology's hosts. This can drastically affect storage applications that replicate (or distribute) data across the cluster. The storage overhead required to tolerate the loss of an arbitrary 1% of hosts without rendering all replicas (or pieces) of a data item inaccessible would be quite expensive, as the item would need to be replicated at more than 1% of the topology's hosts. It is crucial then, that re-convergence periods be as short as possible.

However, the time required for updating network elements to work around failures and to use alternate paths can be substantial. For instance, the time for global re-convergence of the broadcast-based routing protocols (e.g. OSPF and IS-IS) used in today's data centers [17, 54] can be tens of seconds [48]. As each switch receives a routing update, its CPU processes the information, calculates a new forwarding table, determines a new topology, and computes corresponding updates to send to all of its neighbors. Embedded CPUs on switches are generally under-powered and slow compared to a switch's data plane [50, 53] and in practice, settings such as protocol timers can further compound these delays [45]. The processing time at each switch along the path from a failure to the farthest switches adds up quickly. Packets continue to drop during this re-convergence period, crippling applications until recovery completes. Moreover, at the scale of today's data centers, the control overhead required to broadcast updated routing information to all nodes in the topology can be significant.

Long convergence times can be unacceptable in the data center, where the highest levels of availability are required. For instance, an expectation of 5 nines (99.999%) availability translates to about 5 minutes of downtime per year, or 30 failures, if each failure requires a 10 second re-convergence time. A fat tree that supports tens of thousands of hosts has tens or even hundreds of thousands of links. Even a relatively small 64-port, 3-level fat tree has 196,608 links, and in an environment in which switch and link failures happen quite regularly, restricting the number of acceptable yearly failures to 30 is essentially impossible.

The first goal of this dissertation is to introduce a class of data center network topologies that can be tuned with respect to the following characteristics:

1. The topology should be scalable, using a relatively small interconnect of switches to connect as many end hosts as possible.
2. It should provide as much bisection bandwidth as possible in support of all-to-all communication.
3. The topology should retain the path multiplicity of fat trees. If path multiplicity is to be partially sacrificed in favor of other properties, the costs associated with providing this multiplicity should decrease correspondingly.
4. Reactions to topology changes should happen as quickly as the hardware will allow. In particular, failures should not necessitate global re-convergence of broadcast-based routing protocols.

1.2.2 Scalable Addressing and Communication

Another key issue in massive scale data center networks is address assignment as the basis for scalable routing and forwarding protocols. Currently, practitioners typically look to either Layer 2 or Layer 3 techniques for such address assignment, with starkly different tradeoffs. At Layer 2, host address assignment is trivial: it simply consists of the unique MAC address assigned to each network interface at the time of manufacture. While address assignment is simple, routing and forwarding at Layer 2 require global knowledge, broadcast and large forwarding tables. Essentially, every switch must track the location of, and maintain a forwarding table entry for, every host in the network. The overhead in convergence time for maintaining such global knowledge can be significant. Worse, the requisite number of forwarding table entries (one per host) far exceeds the capacity of modern switch hardware [56].

Layer 3 solutions address some of these challenges by assigning hierarchical, topologically meaningful addresses to end hosts. Through subnetting, hosts topologically close to one another share a common prefix in their Layer 3 addresses. With longest prefix matching forwarding, switches need only maintain a single forwarding table entry for a group of hosts that share the same prefix. At Layer 3, the number of required forwarding table entries shrinks substantially and is easily accommodated in switch hardware. However, address assignment now requires centralized and error prone

manual configuration [36, 38, 62], e.g., configuring DHCP servers and assigning subnet masks to each switch. Routing protocols still rely on broadcast, limiting scalability and increasing convergence time.

A number of recent efforts have blurred the distinction between Layer 2 and Layer 3 protocols, delivering some of the benefits of both. PortLand [56] uses a Location Discovery Protocol (LDP) to assign hierarchical addresses to hosts in a fat tree. LDP assumes a full fat tree structure for correct operation and bases address assignment on constructs inherent to this structure. A centralized controller handles the more complicated portions of address assignment as well as all routing. DAC [16] allows for more general topologies but performs all operations in a centralized controller. It also requires manual configuration initially and prior to planned changes. Additionally, DAC depends on a user-provided topology blueprint and necessitates that the physical topology match this blueprint exactly.

The biggest issue with any scheme based on centralized control is that the switching environment essentially requires an out-of-band control network for bringup and bootstrapping. That is, centralized control makes it more challenging to physically combine the data plane with the control plane. Consider the moment at which a switch first comes up. A centralized control scheme requires that the switch locate and communicate with its controller. At data center scale with tens of thousands of switches, the controller is unlikely to be physically connected to each switch. Hence, there must either be a second, physically separate control network (of substantial scale and complexity) or switches must fall back to some complex flooding/broadcast protocol to locate the central controller.

The second goal of this dissertation is to introduce a labeling and communication scheme for hierarchical data center networks that provides the following features:

1. Switches should be able to discover the necessary topology information for connectivity and communication, and to select topologically significant addresses without reliance on centralized components, manual configuration, topology blueprints or global knowledge.

2. Switches should be able to efficiently locate remote hosts and route packets without using centralized lookup or suboptimal paths.

3. The topology should quickly converge to global reachability (assuming underlying physical connectivity) after arbitrary changes. The effects of a topology change should be limited to the area immediately surrounding the change.

4. To lower the barrier to adoption, all components should run on existing hardware, without requiring modifications to end hosts.

1.2.3 Formalizing Label Assignment

A third challenge in the data center is the size and complexity of the underlying network, in terms of configuration and diagnostics. Even when laid out in the most regular of structures, a data center network can be complex, enormous and incredibly difficult to manage and maintain. When communication is disrupted, it is often difficult to pinpoint the source(s) of the problem. One reason for this is that data center networks rely on numerous different protocols that all cooperate to provide connectivity and communication. The interactions between these protocols are complex and often misunderstood, making correct configuration and error diagnosis nearly impossible [36, 38, 62].

Given these challenges, one of our key concerns in the design of ALIAS is to ensure that the protocol is provably correct. Another goal is to break the protocol down into small components, each with a clear interface and list of responsibilities. In this way, it is feasible to reason about the interactions among ALIAS components as well as those between ALIAS and other inter-operating protocols.

The third goal of this dissertation is to introduce a fundamental building block protocol for ALIAS that has the following characteristics:

1. To be usable as a basis for ALIAS, the protocol should not rely on any centralized components, global knowledge or manual configuration.

2. It should enable the scalability of ALIAS to hundreds of thousands of nodes.

3. The protocol should be practical and efficient, with low message overhead and quick convergence time.

4. It should be robust to miswirings and transient startup conditions and it should react and stabilize quickly after both failures and planned topology changes.

5. Finally, simplicity is an important requirement, and the protocol should be provably correct and easy to reason about for the purposes of configuration and failure diagnosis.

1.3 Hypothesis

This dissertation aims to show that we can have scalable, efficient and fault-tolerant communication in hierarchical data center networks, despite the data center's scale and complexity. In particular, we argue that with careful topology design and tailored communication protocols, we can overcome the following three challenges:

1. Building scalable, fault-tolerant topologies that allow network designers to tune scalability and fault tolerance tradeoffs according to the requirements for a particular situation.

2. Providing scalable and efficient addressing and communication.

3. Formalizing the underlying protocols and their interactions in order to make configuration and debugging feasible.

In the following section, we discuss our approaches to each of these challenges.

1.4 Contributions

1.4.1 Aspen Trees: Tuning Scalability and Fault Tolerance

To address the first challenge we present a new set of data center topologies that we coin *Aspen trees*. These trees are similar to the indirect hierarchical topologies found in data centers today, but allow for local failure reaction. That is, rather than waiting for global re-convergence of a broadcast-based protocol such as OSPF, Aspen trees allow the nodes immediately surrounding a link failure to route in-flight packets around the failure. To accommodate this, Aspen trees include extra, redundant links as alternate paths. These extra links take the place of links that would have enabled the topology to support more end hosts. Thus, these links reduce the scalability of the overall topology.

Additionally, Aspen trees retain the high bisection bandwidth and path multiplicity of their fat tree counterparts.

In Chapter 3, we show that Aspen trees can be tuned to have a variety of different failure reaction properties, each corresponding to a different scalability cost. The ability to tune scalability and fault tolerance tradeoffs is important as current topologies do not provide this flexibility, often forcing a data center operator to choose an unnecessarily high level of one property while nearly entirely sacrificing the other. For instance, a fat tree topology [19, 47] provides substantial scalability but at the expense of long reconvergence periods. With Aspen trees, it is possible to create a tree that meets the requirements of a particular network, at exactly the scalability cost that the network administrator is willing to pay.

1.4.2 ALIAS: Scalable Addressing and Communication

In Chapter 4, we present ALIAS, a protocol that addresses our second goal of label assignment as a basis for scalable routing and forwarding in the data center. ALIAS assigns topologically significant labels to hosts and switches, using commodity switch hardware in a decentralized, scalable and broadcast-free manner.

We have completed two implementations of ALIAS [73] and we show the protocol's correctness via model checking as well as its real-world applicability via our testbed implementation. Through our implementation and simulations, we show that ALIAS converges to correct labels quickly, with little control bandwidth and computational overhead. Most importantly, the forwarding tables used by ALIAS switches are comparable in size to those in traditional Layer 3 networks, but ALIAS does not require the centralization or manual configuration necessary for Layer 3 address assignment. This addresses our second goal of providing scalable and efficient addressing and communication, without reliance on centralized control or manual configuration.

A difference between ALIAS and other addressing schemes is that in ALIAS, hosts have multiple labels. This is because ALIAS labels reflect a host's position in the topology as well as the potentially multiple ways to reach that host. Because this is a significant departure from current practice, in Chapter 6, we explore a technique for selecting and using only a single label per host in for ALIAS routing and forwarding.

1.4.3 The Decider/Chooser Protocol in ALIAS

In Chapter 5, we formalize the smallest instance of the problem being solved by ALIAS as the Label Selection Problem (LSP). We then introduce the Decider/Chooser Protocol (DCP), a practical, randomized protocol that solves the Label Selection Problem. We show the correctness of DCP with respect to the requirements of LSP (and thus ALIAS) through proofs and via model checking. We then explore the convergence time of this probabilistic algorithm using simulations and mathematical analysis. We find that due to the random nature of the algorithm, DCP converges quite quickly, even when choosing labels from a small domain. Finally, through a series of protocol refinements, we design extensions to DCP in order to support the more complicated features of ALIAS. We present these refinements as a formal protocol derivation from the basic version of DCP to a full solution for ALIAS.

Our implementation of ALIAS [73] uses DCP and therefore includes an implementation of DCP. However, we also completed a full implementation of the basic DCP and each of its extensions for the purpose of model checking each step of the protocol derivation [72].

DCP addresses our goal of providing a building block for ALIAS that is scalable, practical, and free of global knowledge, centralized control and manual configuration. We show with proofs and model checking that DCP is robust to miswirings and transient network conditions, and our simulations demonstrate its quick stabilization after topology changes. Most importantly, our proof of the correctness of DCP and the corresponding derivation of ALIAS give us confidence in the use of ALIAS in modern data centers.

1.5 Organization

In Chapter 2, we provide the background material relevant to the three components of this thesis. We defer discussions of related work for the individual components to each component's chapter. We also include listings of symbols used throughout this dissertation. In Chapter 3, we describe a new class of data center network topologies, Aspen trees, and provide an algorithm for generating an Aspen tree based on the scala-

bility and fault tolerance requirements for a given network. In Chapter 4, we consider the issue of labeling switches and hosts in a hierarchical data center network. We describe the concept, motivation, design, implementation and evaluation of ALIAS, a protocol for labeling and communication in data center networks. Next we formalize and reason about the key components of ALIAS in Chapter 5, where we show the correctness and performance of the Decider/Chooser Protocol and use this protocol to derive a solution for ALIAS. In Chapter 6, we explore changes to the ALIAS protocol in order to support the selection of a single label per end host. Finally, Chapter 7 summarizes, discusses future work and concludes the dissertation.

1.6 Acknowledgment

Chapter 1, in part, contains material submitted for publication as “Scalability vs. Fault Tolerance in Apsen Trees.” Walraed-Sullivan, Meg; Vahdat, Amin; Marzullo, Keith. The dissertation author was the primary investigator and author of this paper.

Chapter 1, in part, contains material as it appears in the Proceedings of the ACM Symposium on Cloud Computing (SOCC) 2011. Walraed-Sullivan, Meg; Niranjan Mysore, Radhika; Tewari, Malveeka; Zhang, Ying; Marzullo, Keith; Vahdat, Amin. The dissertation author was the primary investigator and author of this paper.

Chapter 1, in part, contains material as it appears in the Proceedings of the 25th International Symposium on Distributed Computing (DISC) 2011. Walraed-Sullivan, Meg; Niranjan Mysore, Radhika; Tewari, Malveeka; Zhang, Ying; Marzullo, Keith; Vahdat, Amin. The dissertation author was the primary investigator and author of this paper.

Chapter 1, in part, contains material submitted for publication as “A Randomized Algorithm for Label Assignment in Dynamic Networks.” Walraed-Sullivan, Meg; Niranjan Mysore, Radhika; Marzullo, Keith; Vahdat, Amin. The dissertation author was the primary investigator and author of this paper.

Chapter 2

Background: Data Center Networks

In this dissertation, we consider challenges surrounding several intertwined aspects of data center networking: topology design, scalable communication, fault tolerance and management overhead. We introduce each in turn in this chapter.

2.1 Topology

When designing a data center network, one of the primary aspects to consider is the physical layout of the topology. Because of the size and complexity of modern data centers, there is often a preference for regular, symmetric topology structures. Symmetric structures enable more uniform bandwidth and latency between pairs of hosts than do their asymmetric counterparts. Frequently, a hierarchical structure is used so that addresses can be aggregated into shared prefixes for reduced forwarding state (Section 2.2). Regardless of the structure of the topology, scalability is a crucial factor. That is, the bisection bandwidth delivered by the network should increase linearly with the number of overall ports provided by the interconnect [2].

There are data center and enterprise network designs that do not rely on structural symmetry. For instance, SEATTLE [42] and DAC [16] both operate over arbitrary topologies and use distributed location discovery and manually-configured blueprints, respectively, to work around the lack of regular structure. Jellyfish [67] takes this a step farther by operating over randomly generated topologies. On the other hand, DCell [30] is based on a recursive interconnect of switches and hosts, in which both

switches and hosts provide switching for packets moving across the network. Similarly, in a BCube [29] network, hosts play a switching role in a topology that is somewhat similar to a generalized hypercube structure [14]. Most frequently, we see data centers organized hierarchically [4, 13, 18, 28, 47, 56] into two or three layers of switches; a multi-rooted tree as a common example of this layout.

As shown in Figure 2.1, multi-rooted tree is a graph in which nodes can be partitioned into levels such that each node belongs to exactly one level. We consider multi-rooted trees in which a switch connects to switches at its own level or at the levels above and below. We call connections between switches of the same level *peer links*. For the purpose of this dissertation, we will refer to the bottom level of such a tree as level L_0 and we label the remaining levels L_1 through L_n, moving up the tree. A second convention that we adopt is to denote with n the total number of levels of switches in the tree and with k the number of ports per switch. So, the example of Figure 2.1 shows a tree with $n = 3$ and $k = 6$. That is, there are three levels of switches in the tree (hosts are at L_0) and each switch has up to six ports connecting it to its neighbors. The rightmost column of Figure 2.1 shows the levels for all switches and hosts, and each node is marked with a globally unique identifier, such as S_7 or H_{15}. In a data center, a MAC address might be used as such a unique identifier.

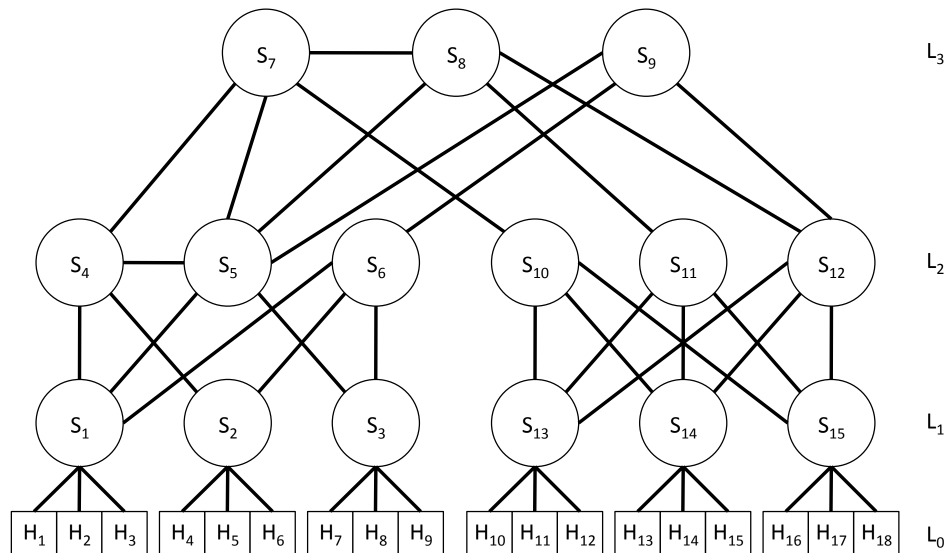


Figure 2.1: Multi-Rooted Tree Topology

In a multi-rooted tree, the number of links between level L_{i+1} and level L_i is typically less than or equal to that between level L_i and level L_{i-1}. This corresponds to techniques to over-subscribe network bandwidth moving up the topology.

A data center interconnect based on a multi-rooted tree can either be organized into an indirect or a direct hierarchy [66]. In a direct hierarchy, a host can connect to any switch in the network whereas in an indirect multi-rooted tree topology, hosts connect only to leaf switches. The example of Figure 2.1 is an indirect topology. As the indirect hierarchy is the common case in hierarchical data centers [4, 13, 18, 28, 56], we focus on this type of topology in this dissertation. In Chapter 4, we design a protocol for scalable addressing and communication in multi-rooted trees such as those shown in Figure 2.1.

A familiar example of a multi-rooted tree is a fat tree [19, 47], as exemplified by Figure 2.2. For this tree, $n = 3$ and $k = 4$.

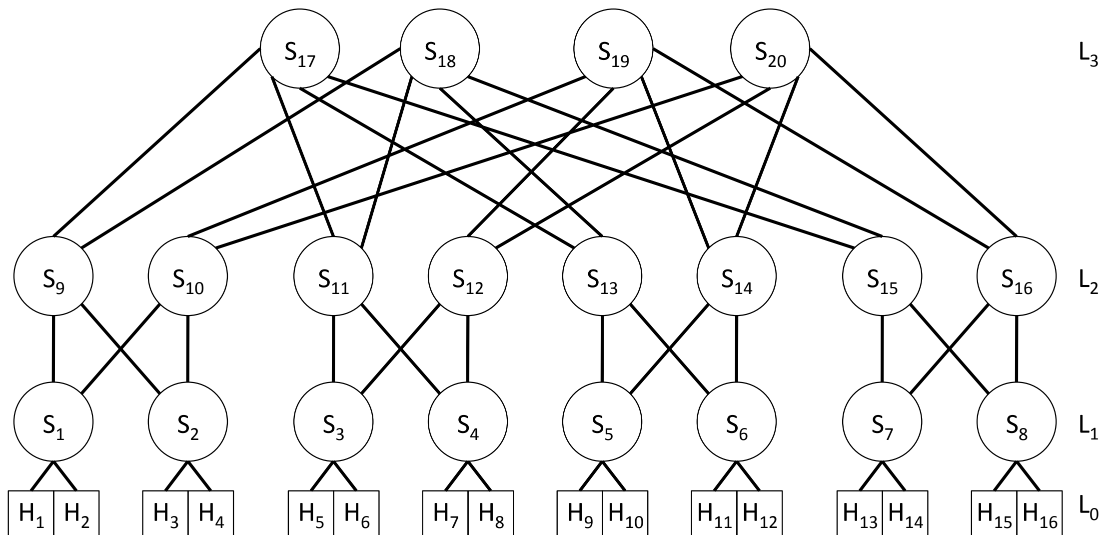


Figure 2.2: Fat Tree Topology

An interesting property of multi-rooted trees, and fat trees in particular, is that there is often significant path multiplicity between pairs of hosts. That is, any two hosts have multiple, possibly link-disjoint[1], paths via which to reach one another. For instance, in Figure 2.2, H_1 can communicate with H_{13} using path S_1-S_9-S_{17}-S_{15}-S_7 or instead via S_1-S_{10}-S_{20}-S_{16}-S_7.

[1]This excludes first-hop links.

We use the term *striping* to denote the particular pattern of interconnections between switches at adjacent levels. In Chapter 3, we show that in a fat tree wired via traditional striping methods, a data packet leaving a particular L_n switch only has a single downward path towards its destination. This limits the usability of path multiplicity, prompting our design of a new class of data center network topologies that we call *Aspen trees*. These topologies are loosely based on the fat tree and enable fast reactions to failures by including redundant links.

Addressing and communication are tightly coupled with topology layout and we discuss these concerns next.

2.2 Addressing and Communication

Another aspect of data center network design is the selection (or development) of communication protocols. With these protocols, switches determine relevant topology information and discover routes to remote switches and hosts. This enables end-to-end communication between hosts in different areas of the network.

Communication relies on the naming or addressing scheme used in a network. A switch's (or host's) address may encode information about the location of the switch (or host) in the topology, as is the case with an IP address. Alternatively, the network's addresses may be *flat*, giving no location information (e.g. MAC addresses). The communication protocols for a data center network include a mechanism for switches to locate other switches and hosts, and the design of such an *address resolution* mechanism depends on the information encoded in a switch or host address. This mechanism can rely on a centralized component with a global view of the topology or it can be distributed among the topology's switches and hosts. For instance, SEATTLE [42] uses a distributed hash table to form a directory service to look up host's locations within the network. In this dissertation, we focus on fully distributed protocols.

Once the addressing scheme and address resolution protocols have been chosen, the next step in the design of a data center network is to select the routing and forwarding protocols that enable communication between hosts. In an indirect topology, a host delivers a packet to its *ingress switch*, which then moves the packet through the network,

based on the routing and forwarding protocols, to the destination host's *egress switch*. Hosts do not play a switching role for in-flight packets in an indirect topology.

Many data center networks use routing and forwarding protocols borrowed from the Internet. For instance, Ethernet (Layer 2) has been one of the standards in data center networking for years. Spanning tree protocols can be used to discover routes, at the cost of requiring global knowledge, broadcast and large forwarding tables. In this case, the MAC addresses assigned to devices out of the box can be used as labels for host naming and identification. Still, Layer 3 protocols must be used over top of Ethernet to scale the topology to data center size. Alternatively, solutions such as PortLand [56] and TRILL [69] can also be used to extend the scalability of Ethernet. Many data centers use IP-based protocols in order to reduce the amount of forwarding state stored at each switch. In this case, protocols such as DHCP are used to assign IP addresses such that hosts physically near one another in the topology have similar addresses through shared prefixes. This reduces the forwarding state of the network's switches, as a switch can refer to a group of hosts via a single shared prefix. Link-state routing protocols such as OSPF and IS-IS are used to compute a global view of the topology at each switch.

On the other hand, some data center networks use custom routing and forwarding protocols or even modified versions of existing protocols. The most relevant example to this dissertation is the up*/down* forwarding introduced by Autonet [65]. In Autonet, packets travel upwards and then back down a hierarchical topology; these direction restrictions are introduced to avoid the occurrence of forwarding loops. More recently, PortLand [56] adopts an up-down forwarding restriction for the same reason. In fact, in a multi-rooted tree without peer links, shortest path forwarding often selects identical paths to those of up*/down* forwarding. When we discuss forwarding in Aspen trees (Chapter 3), we assume shortest path-style communication, which tends to follow an up*/down* pattern. Then in Chapter 4, we design a new addressing scheme called ALIAS that automatically assigns labels to reflect current topology conditions. Though many communication protocols can interoperate with ALIAS labels, we provide an example protocol that accommodates peer links by forwarding with an up*/across*/down* restriction.

2.3 Fault Tolerance

Another property of interest in a data center network is the speed with which the network recovers from topology changes. A topology change can be planned, as is the case when a new rack of servers is added or when a set of switches is de-commissioned. A change can also be unplanned, occurring as the result of a link or switch failure or recovery. Since unplanned changes occur quite frequently in today's data centers, it is imperative that the network react and recover in as short a time as possible, with minimal disruption to ongoing communication sessions.

Routing protocols that rely on all switches having a global view of the topology can have significant re-convergence time. An example of such a protocol is OSPF, in which link-state messages are broadcast to every node in the topology, even after a single link failure. The time to propagate this information to all switches in a large topology can be substantial. This is further exacerbated in the data center, where the embedded switch processors used to calculate global topology information are often slow and under-powered.

On the other hand, some routing protocols allow reactions to failures to occur locally. For instance, fast failure recovery techniques (FFR) [44] in WANs allow for local failure reaction. Similarly, with bounce routing techniques, switches located near a failure cooperate to route in-flight packets around the failure without involving the packets' senders. Many networks also incorporate the notion of backup paths [10, 11, 31, 32, 43, 68, 74, 75]. These paths are created either before a new flow is admitted or on-the-fly after a failure. In many cases, backup paths can significantly reduce the re-convergence period of a network.

In Chapters 3 and 4, we design new communication protocols that quickly and efficiently react to failures in Aspen trees and general multi-rooted trees, respectively. Both leverage the structure of the topology and contain the reaction to only those switches located near to the topology change.

2.4 Management

Finally, management overhead is a significant concern in the data center. Most protocols come with some source of manual configuration. For instance, DAC [16] requires the data center designer to write a blueprint for the topology and to wire the topology so that it matches the blueprint exactly. This can prove difficult, if not infeasible, given the scale and complexity of the data center. On the other hand, protocols that run over arbitrary topologies without blueprints can also require manual configuration. For instance, when IP is used with DHCP for address assignment, an administrator has to configure subnet masks and DHCP servers manually. Moreover, this configuration must be manually updated for most topology changes. As manual configuration has proven to be error prone [36, 38, 62] and difficult, there has been a flurry of research recently that aims to reduce the management burden on the data center operator. In Chapter 4 we introduce ALIAS to remove the manual configuration associated with address assignment in the data center.

2.5 Nomenclature

As there are a number of several different symbols introduced and used throughout this dissertation, we provide in Tables 2.1 through 2.5 a complete list for reference. In some cases, symbols are necessarily overloaded across chapters. Table 2.1 shows symbols used consistently throughout the dissertation. Tables 2.2 through 2.5 give chapter-specific symbols and conventions.

Table 2.1: Symbols Commonly Used throughout Dissertation

Symbol	**Usage**
n	Total number of levels in a tree, zero-based
k	Ports per switch in a topology
L_i, L_f	Any levels i or f of a tree, $i \neq f$
L_0	Bottom level of a tree
L_n	Top (root) level of a tree
s, s'	Any switches in a topology
h, h'	Any hosts in a topology
S_x	Switch with unique identifier x
H_x	Host with unique identifier x
s_x	Any switch at level L_x in a tree
hn, hn'	Any hypernodes in a topology

Table 2.2: Symbols Specific to Chapter 3

Symbol	**Usage**
S	Total number of switches at one level in a tree
p_i	Pods at level L_i
m_i	Member switches in a pod at L_i
r_i	Responsibility of an L_i switch
c_i	Connections from an L_i switch to each L_{i-1} pod

Table 2.3: Symbols Specific to Chapter 4

Symbol	**Usage**
c_i	Coordinate for a switch at L_i
L_iHN	Hypernode at L_i
p	Number of peer links permitted for traversal
S	Size of a switch's unique identifier
C	Size of a coordinate

Table 2.4: Symbols Specific to Chapter 5

Symbol	Usage
ℓ_x	Any L_x switch (code listings)
h	Hypernode iterator (code listings)
c	Any chooser
d	Any decider
c_x	Chooser with unique identifier x
d_x	Decider with unique identifier x
$\mathscr{C}$	Any distributed chooser
$C, C+, D$	Sets of choosers and deciders in proof of DCP correctness
$c.me$	Current choice of chooser c
q	Choosers that finish during a round of DCP
m	Choosers remaining immediately before a round of DCP
L	Size of the DCP coordinate domain

Table 2.5: Symbols Specific to Chapter 6

Symbol	Usage
ℓ	Any label in the topology
c_{n-1}	Coordinate for a switch at L_{n-1}
S_R	Set of switches that reach some L_1 switch using only suboptimal labels
$S_{\ell \wedge R}$	Set of switches that reach some L_1 switch using both optimal and suboptimal labels

2.6 Acknowledgment

Chapter 2, in part, contains material submitted for publication as “Scalability vs. Fault Tolerance in Apsen Trees.” Walraed-Sullivan, Meg; Vahdat, Amin; Marzullo, Keith. The dissertation author was the primary investigator and author of this paper.

Chapter 2, in part, contains material as it appears in the Proceedings of the ACM Symposium on Cloud Computing (SOCC) 2011. Walraed-Sullivan, Meg; Niranjan Mysore, Radhika; Tewari, Malveeka; Zhang, Ying; Marzullo, Keith; Vahdat, Amin. The dissertation author was the primary investigator and author of this paper.

Chapter 2, in part, contains material as it appears in the Proceedings of the 25th International Symposium on Distributed Computing (DISC) 2011. Walraed-Sullivan, Meg; Niranjan Mysore, Radhika; Tewari, Malveeka; Zhang, Ying; Marzullo, Keith; Vahdat, Amin. The dissertation author was the primary investigator and author of this paper.

Chapter 2, in part, contains material submitted for publication as “A Randomized Algorithm for Label Assignment in Dynamic Networks.” Walraed-Sullivan, Meg; Niranjan Mysore, Radhika; Marzullo, Keith; Vahdat, Amin. The dissertation author was the primary investigator and author of this paper.

Chapter 3

Scalability versus Fault Tolerance in Aspen Trees

In this chapter, we present *Aspen trees*, a class of data center network topologies that allow the network designer to tune a multi-rooted tree topology with respect to the tradeoffs between convergence time and scalability. Our goal with Aspen trees is to eliminate excessive periods of host disconnection in the data center. It is unrealistic to limit the number of failures sufficiently to meet the stringent availability requirements of the data center. Therefore, we consider the problem of drastically reducing the convergence time for each individual failure. We do so by modifying fat tree topologies to enable *local failure reactions*. Instead of requiring global OSPF convergence on a link failure, we send simple failure notification messages to a small subset of switches located near the failure. This substantially decreases the re-convergence time (by sending small updates over fewer hops) and the control overhead (by involving considerably fewer nodes and eliminating reliance on broadcast). We choose the name *Aspen tree* in reference to a species of tree that survives for years after the failure of redundant roots.

Engineering topologies to support local failure reactions comes with a cost, namely, the tree supports fewer hosts and accommodates less hierarchical aggregation. While a reduction in host support decreases the cost efficiency and scalability of a network in a clear way, the effects of reducing hierarchical aggregation may be more subtle. Addressing and communication schemes that leverage hierarchy to create shared forwarding prefixes are affected by such changes.

In this chapter, we explore the scalability and fault tolerance tradeoffs of building a highly available large-scale network that can react to failures locally. We give an algorithm to determine the set of Aspen trees that can be created, given constraints such as the number of available switches or the requirements for host support. To precisely specify the fault tolerance properties of these trees, we introduce a *Fault Tolerance Vector (FTV)* that quantifies failure reactivity by indicating the quality and locations of added fault tolerance throughout the tree. We then formalize a tree's scalability properties in terms of its FTV. Finally, we present a communication protocol that leverages added fault tolerance in an Aspen tree. This gives intuition about the relative values (or suitability to a particular goal) of different Aspen trees with identical scalability costs and differing FTVs. Finally, we offer recommendations for optimal trees given a set of requirements and goals.

To add fault tolerance to a tree, we introduce redundant links at one or more levels of the tree. This leads to a reduction in the number of hops through which routing updates propagate and thus to a decrease in convergence time. We find that the introduction of these redundant links *at a single level of the tree* results in a multiplicative reduction in the same amount to the maximum number of hosts that can be supported by the tree. That is, we reduce the total number of hosts in the tree by 50% for each level at which we increase from 0 to 1 the number of link failures tolerable without host disconnection. Therefore, adding fault tolerance at every tree level likely comes at too high a cost, as per-level changes multiply quickly. However, our communication protocol is designed to leverage even a small increase in a tree's fault tolerance. In fact, solutions in which only a single level (the highest in the tree) has additional links prove ideal in many situations, as they reduce convergence time by 50%, with the lowest possible cost in terms of host count.

3.1 Failures in Traditional Fat Trees

In a traditional fat tree, a single link failure can be devastating. It can cause all packets destined to a set of hosts to be dropped while updated routing state propagates to *every node in the topology*. For instance, consider a packet traveling from host x to host

y in the 4-level, 4-port fat tree of Figure 3.1 and suppose that the link between switches f and g fails shortly before the packet reaches f. f no longer has a downward path to y and drops the packet. In fact, with the failure of link $f-g$, the packet would have to travel through h to reach its destination. For this to happen, x's ingress switch a would need to know about the failure and to select a next hop accordingly.

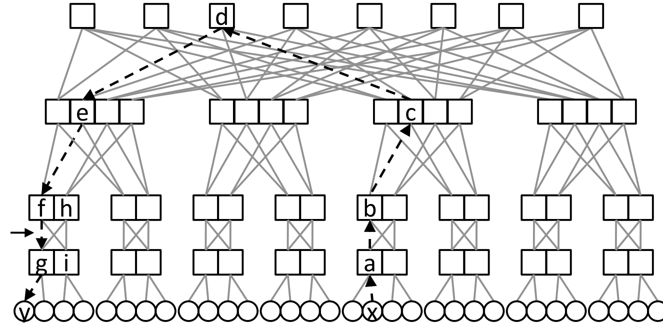


Figure 3.1: Packet Travel in a 4-Level, 4-Port Fat Tree

This means that in the worst case, information about a single link failure needs to propagate to all of the lowest level switches of the tree, passing through every single switch in the process. The overhead of informing so many switches can be significant. The time for this information to propagate grows with the depth of the tree, and the time for recalculating routing state and updating forwarding tables can be substantial.

There are alternative routing techniques that avoid packet loss. For instance, a bounce routing-based technique might send the packet from f to i. Switch i can then bounce the packet back up to h, which still has a path to g. However, bounce routing based on local information introduces additional software complexity to support the calculation and activation of extra, non-shortest path entries and to avoid forwarding loops. Additionally, bounce routing can cause deadlock when combined with flow control protocols [20, 37].

It is possible to construct a protocol that sends the packet back along its path to the nearest switch that can re-route around the failed link, similar to the technique employed by data-driven connectivity (DDC) [50]. In DDC, a packet sent along the path in Figure 3.1 would travel from f back up to the top of the tree and then down three levels to a before it could be re-routed towards h.

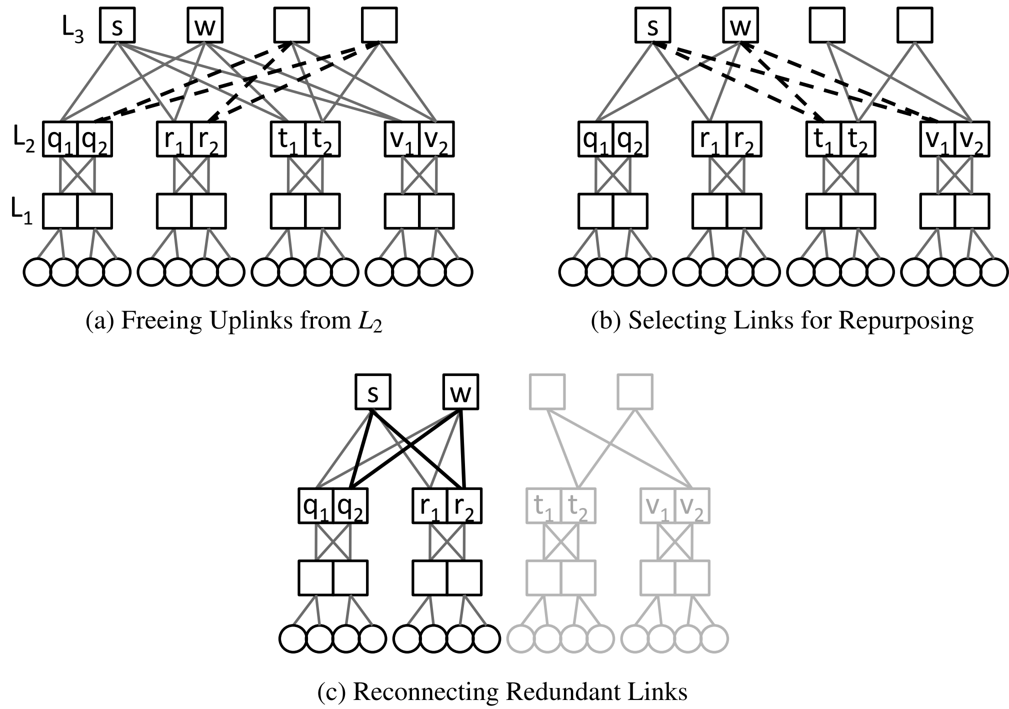
Our approach is to offer an alternative to bouncing packets in either direction. We modify the fat tree by introducing redundancy at a particular level; this allows switches to handle a failure locally without requiring global convergence of topology information. These additional redundant links come at a cost in the topology's scale. We coin the resulting modified fat trees Aspen trees.

Before describing Aspen trees in detail, we define several key terms. An n-level, k-port Aspen tree consists of switches at levels 1 through n (written as $L_1 ... L_n$) and hosts at level 0 (L_0). Each switch has k ports, half of which connect to switches in the level above and half of which connect to switches below. Switches at L_n have k downward-facing ports. We group switches at each level L_i into *pods*. A pod includes the maximal set of L_i switches that all connect to the same set of L_{i-1} pods below, and an L_1 pod consists of a single L_1 switch. In a traditional fat tree, there are S switches at levels L_1 through L_{n-1} and $\frac{S}{2}$ switches at L_n; we retain this property in Aspen trees. For now, we do not consider multi-homed hosts, given the associated addressing complications.

3.2 Designing Aspen Trees

In this section, we describe our method for generating trees with varying fault tolerance properties. Intuitively, our approach is to begin with a traditional fat tree, and then to disconnect links at a given level and "repurpose" them as redundant links for added fault tolerance at the same level. By increasing the number of links between a subset of switches at adjacent levels, we necessarily disconnect another subset of switches at those levels. These newly disconnected switches and their descendants are deleted, ultimately resulting in a decrease in the number of hosts supported by the topology.

Figure 3.2 depicts a sample of this process pictorially. In Figure 3.2a, L_3 switch s connects to four L_2 pods, namely $q = \{q_1, q_2\}$, $r = \{r_1, r_2\}$, $t = \{t_1, t_2\}$ and $v = \{v_1, v_2\}$. To increase fault tolerance between L_3 and L_2, we decide to provide redundant connections from s to pods q and r. We first need to free some upward facing ports from q and r, and we chose the uplinks from q_2 and r_2 as candidates for deletion because they connect to L_3 switches other than s. Once we have deleted these links from q_2 and r_2, we select links to repurpose. Since we wish to increase fault tolerance between s and



(a) Freeing Uplinks from L_2

(b) Selecting Links for Repurposing

(c) Reconnecting Redundant Links

Figure 3.2: Modifying a 3-Level, 4-Port Fat Tree to Have 1-Fault Tolerance at L_3

pods *q* and *r*, we must do so at the expense of pods *t* and *v*, by removing links from *s* to pods *t* and *v* as shown by the dotted lines in Figure 3.2b. For symmetry, we also include switch *w* with *s*. The repurposed links are then connected to the open upward facing ports of q_2 and r_2, leaving the right half of the tree disconnected and ready for deletion, as shown in Figure 3.2c. At this point, *s* is connected to each L_2 pod via two distinct switches and can reach either pod despite the failure of on such link. We describe this tree as 1-*fault-tolerant* at L_3. In general, we use L_i fault tolerance to refer to L_i-to-L_{i-1} links.

For a tree with a given depth and switch size, there may be multiple options for the amount of fault tolerance to add at each level, and fault tolerance can be added to any subset of levels. Additionally, decisions made at one level may affect the available options for other levels. In the following sections, we present an algorithm that makes a coherent set of these per-level decisions throughout an Aspen tree.

3.2.1 Assumptions

In order to limit our attention to a tractable set of options, we introduce a few restrictions on the types of trees we wish to generate. First, we consider only trees in which switches at each level are divided into pods of uniform size. That is, all pods at L_i must be of equal size, though this size may differ from that of the pods at $L_{f:f \neq i}$. Within a single level, all switches have equal fault tolerance to pods in the level below, but as with pod division, the fault tolerance of switches at L_i need not equal that of switches at L_f.

3.2.2 Aspen Tree Generation

We begin at the top level of the tree, L_n, and group the switches into a single pod. We then select a value for the fault tolerance of the connections to the level below, L_{n-1}. Next, we move to L_{n-1}, and divide the L_{n-1} switches into pods (based on the selected L_n values) and choose a value for fault tolerance of the connections to L_{n-2} switches. We repeat this process for each level moving down the tree, terminating when we reach L_1. At each level, we select values according to a set of constraints that ensure that all of the per-level choices work together to form a coherent topology.

Variables and Constraints

Before presenting the technical details of our algorithm, we first introduce several helpful variables and relationships between them. Recall that an Aspen tree has n levels of switches, and that all switches have exactly k ports. In order for the uplinks from L_i to properly match all downlinks from L_{i+1}, to allow for full bisection bandwidth, the number of switches at all levels of the tree except L_n must be the same. We denote this number of switches per level with S. Each L_n switch has twice as many downlinks (k) as the uplinks of an L_{n-1} switch and so for uplinks to match downlinks at these levels, there are $\frac{S}{2}$ L_n switches.

At each level, our algorithm first groups switches into pods and then selects a fault tolerance value to connect to pods below. We represent these choices with four variables: p_i, m_i, r_i and c_i. The first two variables encode pod divisions; p_i indicates the

number of pods at L_i, and m_i represents the number of members per L_i pod. Combining this with our values for the number of switches at each level, we have the constraint:

$$p_i m_i = S, 1 \leq i < n \qquad p_n m_n = \frac{S}{2} \tag{3.1}$$

The other two variables, r_i and c_i, relate to per-level fault tolerance. r_i expresses the responsibility of a switch and is a count of the number of L_{i-1} pods to which each L_i switch connects. c_i denotes the number of connections from an L_i switch s to each of the L_{i-1} pods that s neighbors. Since we require (Section 3.2.1) that switches' fault tolerance properties are uniform within a level, a switch's downward links are spread evenly among all L_{i-1} pods that it neighbors. Combining this with the number of downlinks at each level, we have:

$$r_i c_i = \frac{k}{2}, 1 < i < n \qquad r_n c_n = k \tag{3.2}$$

Each constraint listed thus far relates to only a single level of the tree. Our final equation connects values at adjacent levels. Every pod q below L_n must have a neighboring pod above, otherwise q and its descendants would be disconnected from the graph. This means that the set of pods at $L_{i:i \geq 2}$ must "cover" (or rather, be responsible for) all pods at L_{i-1}:

$$p_i r_i = p_{i-1}, 1 < i \leq n \tag{3.3}$$

Aspen Tree Generation Algorithm

We now use Equations 3.1 through 3.3 to formalize our algorithm, which is presented in pseudo code in Listing 3.1. The algorithm calculates values for p_i, m_i, r_i, c_i and S (lines 1-5), using a level iterator and a record of the number of downlinks at each level (lines 6-7).

We begin with the requirement that each L_n switch connects at least once to each L_{n-1} pod below. This effectively groups all L_n switches into a single pod, so $p_n = 1$ (line 8). We defer calculation of m_n until the value of S is determined.

We consider each level in turn from the top of the tree downwards (lines 9, 14). At each level, we select appropriate values for fault tolerance variables c_i and r_i (lines 10-11) with respect to constraint Equation 3.2. Alternatively, we could accept as an

Listing 3.1: Aspen Tree Generation Algorithm

```
    input  : k, n
    output: p, m, r, c, S
 1  int p[1...n] = 0
 2  int m[1...n] = 0
 3  int r[2...n] = 0
 4  int c[2...n] = 0
 5  int S
 6  int i = n
 7  int downlinks = k
 8  p[n] = 1
 9  while i ≥ 2 do
10      choose c[i] s.t. c[i] is a factor of downlinks
11      r[i] = downlinks ÷ c[i]
12      p[i − 1] = p[i]r[i]
13      downlinks = k/2
14      i = i − 1
15  S = p[1]
16  m[n] = S ÷ 2
17  for i = 1 to n − 1 do
18      m[i] = S ÷ p[i]
19      if m[i] ∉ ℤ
20          report error and exit
21  if m[n] ∉ ℤ
22      report error and exit
```

input, desired per-level fault tolerance values. In this case, we would set each c_i value by adding 1 to the desired fault tolerance for L_i. Based on the value of r_i, we use Equation 3.3 to determine the number of pods in the level below (line 12). Finally, we move to the next level, updating the number of downlinks accordingly (lines 13-14).

The last iteration of the loop calculates the number of pods at L_1. Since each L_1 switch is in its own pod, we know that $S = p_1$ (line 15). We use the value of S with Equation 3.1 to calculate m_i values (lines 16-18). If at any point, we encounter a non-integer value for m_i, we have generated an invalid tree and we exit (lines 19-22).

Note that instead of making decisions for the values of r_i and c_i at each level, we can choose to enumerate all possibilities. Rather than creating a single tree, this generates an exhaustive listing of all possible Aspen trees given k and n.

Generation Example

We illustrate our algorithm for creating modified topologies with a simple example, in which $k = 6$ and $n = 4$, using enumeration to generate all possible topologies, as shown in Table 3.1. We need values for p_i, m_i, r_i and c_i for each level of the tree but L_1; these make up the 13 columns of the table. We omit columns for r_1, c_1 and m_1 as L_1 switches do not connect to switches below, and $m_1 = 1$.

Table 3.1: Topology Enumeration with $k = 6$ and $n = 4$

<table>
<tr><th>p_4</th><th>m_4</th><th>r_4</th><th>c_4</th><th>p_3</th><th>m_3</th><th>r_3</th><th>c_3</th><th>p_2</th><th>m_2</th><th>r_2</th><th>c_2</th><th>p_1</th></tr>
<tr><td rowspan="16">1</td><td rowspan="16">$\frac{S}{2}$</td><td rowspan="4">6</td><td rowspan="4">1</td><td rowspan="4">6</td><td rowspan="4">$\frac{S}{6}$</td><td rowspan="2">3</td><td rowspan="2">1</td><td rowspan="2">18</td><td rowspan="2">$\frac{S}{18}$</td><td>3</td><td>1</td><td>54</td></tr>
<tr><td>1</td><td>3</td><td>18</td></tr>
<tr><td rowspan="2">1</td><td rowspan="2">3</td><td rowspan="2">6</td><td rowspan="2">$\frac{S}{6}$</td><td>3</td><td>1</td><td>18</td></tr>
<tr><td>1</td><td>3</td><td>6</td></tr>
<tr><td rowspan="4">3</td><td rowspan="4">2</td><td rowspan="4">3</td><td rowspan="4">$\frac{S}{3}$</td><td rowspan="2">3</td><td rowspan="2">1</td><td rowspan="2">9</td><td rowspan="2">$\frac{S}{9}$</td><td>3</td><td>1</td><td>27</td></tr>
<tr><td>1</td><td>3</td><td>9</td></tr>
<tr><td rowspan="2">1</td><td rowspan="2">3</td><td rowspan="2">3</td><td rowspan="2">$\frac{S}{3}$</td><td>3</td><td>1</td><td>9</td></tr>
<tr><td>1</td><td>3</td><td>3</td></tr>
<tr><td rowspan="4">2</td><td rowspan="4">3</td><td rowspan="4">2</td><td rowspan="4">$\frac{S}{2}$</td><td rowspan="2">3</td><td rowspan="2">1</td><td rowspan="2">6</td><td rowspan="2">$\frac{S}{6}$</td><td>3</td><td>1</td><td>18</td></tr>
<tr><td>1</td><td>3</td><td>6</td></tr>
<tr><td rowspan="2">1</td><td rowspan="2">3</td><td rowspan="2">2</td><td rowspan="2">$\frac{S}{2}$</td><td>3</td><td>1</td><td>6</td></tr>
<tr><td>1</td><td>3</td><td>2</td></tr>
<tr><td rowspan="4">1</td><td rowspan="4">6</td><td rowspan="4">1</td><td rowspan="4">S</td><td rowspan="2">3</td><td rowspan="2">1</td><td rowspan="2">3</td><td rowspan="2">$\frac{S}{3}$</td><td>3</td><td>1</td><td>9</td></tr>
<tr><td>1</td><td>3</td><td>3</td></tr>
<tr><td rowspan="2">1</td><td rowspan="2">3</td><td rowspan="2">1</td><td rowspan="2">S</td><td>3</td><td>1</td><td>3</td></tr>
<tr><td>1</td><td>3</td><td>1</td></tr>
</table>

Reading from left to right, we begin with the fact that, regardless of any other values, $p_4 = 1$ and $m_4 = \frac{S}{2}$. We now can make a choice for the fault tolerance between L_4 and L_3. Since each L_4 switch has $k = 6$ downlinks to spread evenly among lower-level pods, we have four possibilities for (r_4, c_4). Each corresponds to a different L_4 fault tolerance. For instance setting $r_4 = 1$ and $c_4 = 6$ generates a topology in which each L_4 switch connects 6 times to a single L_3 pod below, whereas setting $r_4 = 3$ and $c_4 = 2$ connects each L_4 switch twice to each of 3 L_3 pods below. Based on the fact that all L_3 pods must be covered by L_4 pods ($p_4 r_4 = p_3$, Equation 3.3), the p_3 column can be filled in. We then use Equation 3.1 to populate the m_3 column.

Since each L_3 switch has $\frac{k}{2}$ downlinks to be spread evenly over L_2 pods, we have that $r_3 c_3 = \frac{k}{2}$. We use this to split the table into possible values for r_3 and c_3, and continue by using Equations 3.3 and 3.1 to populate the p_2 and m_2 columns, respectively. The process for generating values for r_2 and c_2 is identical for that of r_3 and c_3, and these values give us entries for the remaining column, p_1.

Now that we have determined values for p_1 for each possible topology, we can use the fact that $p_1 = S$ to replace any entries that depend on S with numerical values. Finally, we remove rows of the chart that include non-integer values for any variables. Table 3.2 gives our final chart of possibilities for 4-level trees with 6-port switches.

Table 3.2: Replacing *S* with Numerical Values
(Shaded rows have been cut from table)

p_4	m_4	r_4	c_4	p_3	m_3	r_3	c_3	p_2	m_2	r_2	c_2	p_1
1	27	6	1	6	9	3	1	18	3	3	1	54
1	9	6	1	6	3	3	1	18	1	1	3	18
1	9	6	1	6	3	1	3	6	3	3	1	18
1	3	6	1	6	1	1	3	6	1	1	3	6
1	13.5	3	2	3	9	3	1	9	3	3	1	27
1	4.5	3	2	3	3	3	1	9	1	1	3	9
1	4.5	3	2	3	3	1	3	3	3	3	1	9
1	1.5	3	2	3	1	1	3	3	1	1	3	3
1	9	2	3	2	9	3	1	6	3	3	1	18
1	3	2	3	2	3	3	1	6	1	1	3	6
1	3	2	3	2	3	1	3	2	3	3	1	6
1	1	2	3	2	1	1	3	2	1	1	3	2
1	4.5	1	6	1	9	3	1	3	3	3	1	9
1	1.5	1	6	1	3	3	1	3	1	1	3	3
1	1.5	1	6	1	3	1	3	1	3	3	1	3
1	.5	1	6	1	1	1	3	1	1	1	3	1

3.3 Aspen Tree Properties

An Aspen tree generated by the algorithm of Section 3.2 is defined by the set of per-level values selected for p_i, m_i, r_i and c_i; these values determine the per-level

fault tolerance, the number of switches needed and hosts supported, and the amount of hierarchical aggregation from one level to the next.

3.3.1 Fault Tolerance

The fault tolerance at each level of an Aspen tree is determined by the number of connections c_i that each switch s has to pods below. If all but one of the connections between s and a pod q fail, s can still reach q and can route packets to q's descendants. Thus the fault tolerance at L_i is $c_i - 1$.

To express the fault tolerance of a tree as a whole, we introduce the *Fault Tolerance Vector (FTV)*. The FTV lists, from the top of the tree down, individual fault tolerance values for each level, i.e. $<c_n - 1, ... c_2 - 1>$. For instance, an FTV of $<3,0,1,0>$ indicates a five level tree, with 4 links between every L_5 switch and each neighboring L_4 pod, 2 links between an L_3 switch and each neighboring L_2 pod, and only a single link between an L_4 (L_2) switch and neighboring L_3 (L_1) pods. The FTV for a traditional fat tree is $<0,...,0>$.

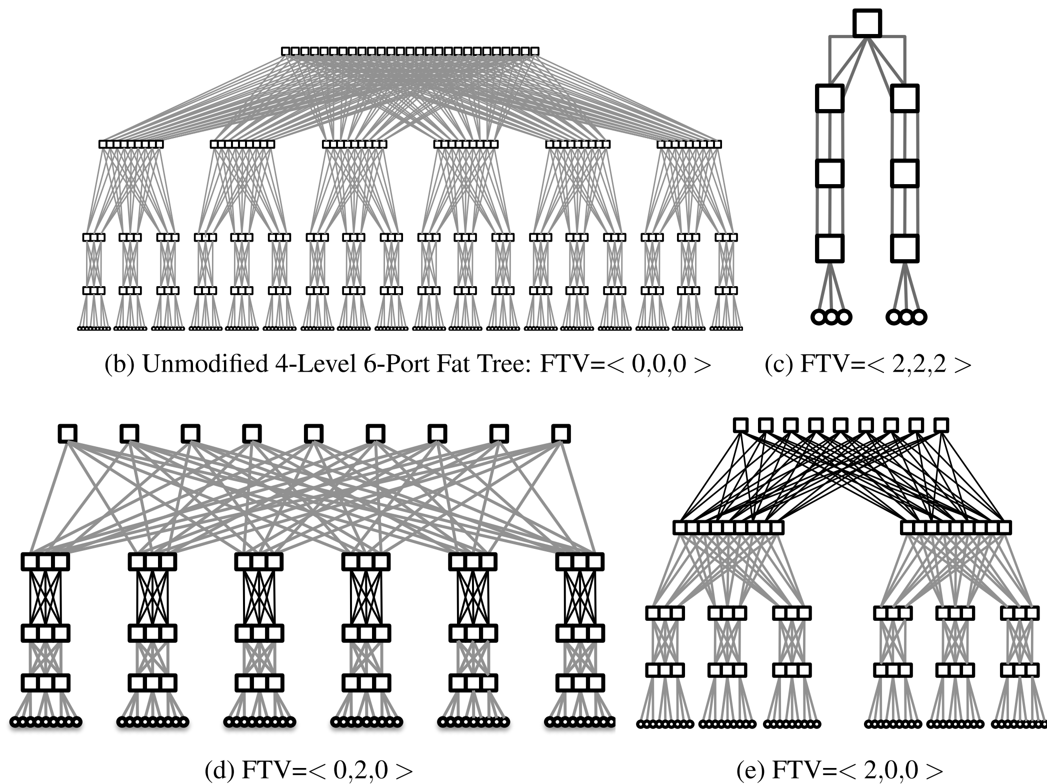
Figure 3.3 presents four sample 4-level Aspen trees of 6-port switches, each with different FTVs. Figure 3.3a lists all possible $k = 6$, $n = 4$ Aspen trees, omitting trees that have a non-integer value for m_i at any level (Listing 3.1). At one end of the spectrum, we have the unmodified fat tree of Figure 3.3b. In this tree, each switch connects via only a single link to each pod below. On the other hand, in the tree of Figure 3.3c, each switch connects three times to each pod below, giving this tree an FTV of $<2,2,2>$. Figures 3.3d and 3.3e show more of a middle ground, each adding duplicate connections at exactly one level of the tree.

3.3.2 Number of Switches Needed

In order to discuss the number of switches and hosts in an Aspen tree, we need a compact way to express the variable S. Recall that our algorithm begins with a value for p_n, chooses a value for r_n, and uses this to generate a value for p_{n-1}, iterating down the tree towards L_1. The driving factor that moves the algorithm from one level to the

Fault Tolerance		S	Switches	Hosts	Hierarchical Aggregation			
FTV	**DCC**				L_4	L_3	L_2	Overall
<**0,0,0**>	1	54	**189**	**162**	**3**	**3**	**3**	**27**
<0,0,2>	3	18	63	54	3	3	1	9
<**0,2,0**>	3	18	**63**	**54**	**3**	**1**	**3**	**9**
<0,2,2>	9	6	21	18	3	1	1	3
<**2,0,0**>	3	18	**63**	**54**	**1**	**3**	**3**	**9**
<2,0,2>	9	6	21	18	1	3	1	3
<2,2,0>	9	6	21	18	1	1	3	3
<**2,2,2**>	27	2	**7**	**6**	**1**	**1**	**1**	**1**

(a) All Possible 4-Level, 6-Port Aspen Trees
(Bold rows correspond to topologies pictured.)



(b) Unmodified 4-Level 6-Port Fat Tree: FTV=< 0,0,0 >

(c) FTV=< 2,2,2 >

(d) FTV=< 0,2,0 >

(e) FTV=< 2,0,0 >

Figure 3.3: Examples of 4-Level, 6-Port Aspen Trees

next is Equation 3.3. “Unrolling” this chain of equations from L_1 up, we have:

$$\begin{aligned} p_1 &= p_2 r_2 \\ p_2 = p_3 r_3 \ \rightarrow p_1 &= (p_3 r_3) r_2 \\ &... \\ p_{n-1} = p_n r_n \ \rightarrow p_1 &= (p_n r_n) r_{n-1} ... r_3 r_2 \\ p_n = 1 \ \rightarrow p_1 &= r_n r_{n-1} ... r_3 r_2 \\ \forall i : 1 \leq i < n, \, p_i &= \prod_{j=i+1}^{n} r_j \end{aligned}$$

We use Equation 3.2 and the fact that S is equal to the number of pods at L_1 to express S as a function of each level's c_i value:

$$S = p_1 = \prod_{j=2}^{n} r_j = r_n \times \prod_{j=2}^{n-1} r_j = \frac{k}{c_n} \times \prod_{j=2}^{n-1} \frac{k}{2c_j} = \frac{k^{n-1}}{2^{n-2}} \times \prod_{j=2}^{n} \frac{1}{c_j}$$

To simplify the equation for S, we introduce the *Duplicate Connection Count (DCC)*, which when applied to an FTV, adds one to each entry (to convert per-level fault tolerance values into corresponding c_i values) and multiplies the resulting vector's elements into a single value. The DCC expresses the fault tolerance of a tree in terms of the number of link-disjoint paths from each L_n switch to each L_1 descendant. For instance, the DCC of an Aspen tree with FTV <1,2,3> is $2 \times 3 \times 4 = 24$. We rewrite the equation for S as $S = \frac{k^{n-1}}{2^{n-2} DCC}$. Figure 3.3a shows the DCCs and corresponding values of S for each tree, where S is equal to 54 divided by the tree's DCC.

This compact representation for S makes it simple to calculate the total number of switches in a tree. Levels L_1 through L_{n-1} each have S switches and L_n has $\frac{S}{2}$ switches. This means that there are $(n - \frac{1}{2})S$ switches altogether in an Aspen tree. Figure 3.3a gives the number of switches in each example tree, using $n - \frac{1}{2} = 3.5$.

3.3.3 Number of Hosts Supported

The most apparent cost of adding fault tolerance to an Aspen tree is the resulting reduction in the number of hosts supported. In fact, each time the fault tolerance of a single level is increased by an additive factor of x with respect to that of a minimal

fat tree, the number of hosts in the tree is decreased by a *multiplicative* factor of x. To see this, note that the maximum number of hosts in the tree is simply the number of L_1 switches multiplied by the number of downward facing ports per L_1 switch. That is,

$$hosts = \frac{k}{2} \times S = \frac{k^n}{2^{n-1}} \times \prod_{j=2}^{n} \frac{1}{c_j} = \frac{k^n}{2^{n-1}DCC} \tag{3.4}$$

As Equation 3.4 shows, changing an individual level's value for c_i from the default of 1 to $x > 1$ results in a multiplicative reduction of $\frac{1}{x}$ to the number of hosts supported. This tradeoff is shown for all 4-level, 6-port Aspen trees in Figure 3.3a and also in the corresponding examples of Figures 3.3b through 3.3c. The traditional fat tree of Figure 3.3b has no fault tolerance and a corresponding DCC of 1. Therefore it supports the maximal number of hosts, in this case, 162. On the other hand, the tree in Figure 3.3c has a fault tolerance of 2 between every pair of levels. Each level contributes a factor of 3 to the tree's DCC, reducing the number of hosts supported by a factor of 27 from that of a traditional fat tree. Increasing the fault tolerance at any single level of the tree affects the host count in an identical way. For instance, Figures 3.3d and 3.3e have differing FTVs, as fault tolerance has been added at a different level in each tree. However, the two trees have identical DCCs and thus support the same number of hosts.

3.3.4 Hierarchical Aggregation

Another property of interest is hierarchical aggregation, that is, how many pods at L_i are folded into each L_{i+1} pod. While hierarchical aggregation is generally less of a concern than the number of hosts supported, it may play a role in determining the efficiency of certain communication schemes. For hierarchical topologies, a labeling scheme such as those in [56, 73] can be used to enable compact forwarding state. In this type of labeling scheme, descendant switches below a given L_i pod share the same label prefix, and therefore it is desirable to group as many L_{i-1} switches together as possible under a single L_i switch. The hierarchical aggregation at L_i of an Aspen tree expresses the number of L_{i-1} pods to which each L_i switch connects, and can be written as $\frac{m_i}{m_{i-1}}$.

As with host count, there is a direct tradeoff between fault tolerance and hierarchical aggregation. This is because the number of downlinks available at each switch does not change as the fault tolerance of a tree is varied. So if the c_i value for a switch s

is to be increased, the extra links must come from other downward neighbors of s. This necessarily reduces the number of pods to which s connects below.

It is difficult to provide an equation that directly relates fault tolerance and hierarchical aggregation at a single level, because hierarchical aggregation is not a single-level concept. To increase the hierarchical aggregation at L_i ($\frac{m_i}{m_{i-1}}$) we must either increase m_i or decrease m_{i-1}. However, this in turn reduces either L_{i+1} or L_{i-1} hierarchical aggregation. Because of this, we consider the hierarchical aggregation across the entire tree. While this does not provide a complete picture, it does give intuition about the trade-off between fault tolerance and hierarchical aggregation. We measure an Aspen tree's overall hierarchical aggregation as the product of its per-level hierarchical aggregation values:

$$\frac{m_n}{m_{n-1}} \times \frac{m_{n-1}}{m_{n-2}} \times \ldots \times \frac{m_3}{m_2} \times \frac{m_2}{m_1} = \frac{m_n}{m_1} = \frac{S}{2}$$

Therefore, overall hierarchical aggregation has an identical dependency on an Aspen tree's FTV to that of host count; an additive increase to a level's c_i value results in a multiplicative reduction in hierarchical aggregation by the same factor. Figure 3.3b has the maximal possible hierarchical aggregation at each level (in this case, 3) while Figure 3.3c has no hierarchical aggregation at all. The additional fault tolerance at a single level of each of Figures 3.3e and 3.3d costs these trees a corresponding factor of 3 in overall aggregation. The values related to hierarchical aggregation for all possible $k = 6$, $n = 4$ Aspen trees are given in Figure 3.3a.

3.4 Leveraging Fault Tolerance: Routing Around Failures

Recall from the example of Section 3.1 that a minimal fat tree has no choice but to drop a packet arriving at a switch incident on a failed link. In fact, in Figure 3.1, a packet sent from host x to host y would be doomed to be lost the instant that x's ingress switch a selected b as the packet's next hop. Extra fault tolerance links allow for this "dooming" decision to happen later in the packet's path; this reduces the chances that a packet will be dropped due to a failure that occurs while the packet is in flight. Moreover,

keeping the set of switches that need to react close to a failure limits both convergence time and overhead of the reaction.

Figure 3.4 shows a $k = 4$, $n = 4$ Aspen tree, modified from the 4-level, 4-port fat tree of Figure 3.1 to have an FTV of <0,1,0>, with additional fault tolerance links between L_3 and L_2. As described in Section 3.3, this comes at the cost of half of the hosts in the tree. The added fault tolerance links between L_3 and L_2 give a packet sent from x to y an alternate path through h, as indicated by the darkened arrows. If switch e knows about the failed link between f and g, it can simply route packets towards h rather than g.

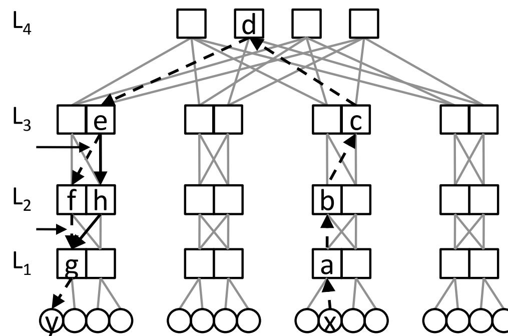


Figure 3.4: 4-Level, 4-Port Aspen Tree with FTV= <0,1,0>

In this example the switch that needs to know about the failure and make alternate routing decisions is relatively far along the packet's path. In contrast, in the traditional fat tree of Figure 3.1, knowledge of the failure needs to propagate all the way back to the sender's ingress switch. In other words, the addition of fault tolerance links reduces the set of switches that react to a failure to the ancestors of a switch incident on the failure, rather than the entire tree. Based on this property, we suggest a protocol for reacting to failures in Aspen trees with added fault tolerance links (and non-zero FTV entries).

3.4.1 Communication Protocol Overview

A key reason for the slow convergence of broadcast-based protocols (e.g. OSPF and IS-IS) in the data center is the need to disseminate topology information to every possible sender after a single link failure. Each switch performs expensive calculations

that grow with the size of the topology, and routing updates propagate through a number of hops proportional to the depth of the tree.

We leverage the existence of added fault tolerance links to drastically reduce this expense, by considering an insight similar to that of failure-carrying packets [45]: the tree consists a relatively stable set of deployed physical links, and a subset of these links are up and available at any given time. Our approach is to allow OSPF to converge for the full physical topology, and to use separate out-of-band notifications to alert nearby ancestor switches of transient link failures and recoveries. These ancestors can select alternate paths to avoid failures, even for packets that are in flight as a failure occurs. The number of hops across which notifications propagate is smaller, as notifications move upwards to nearby ancestors rather than up and back down the entire tree. More importantly, these notifications are much simpler to compute and to process than the corresponding calculations required for global converge of OSPF. Finally, the number of switches that react to the failure decreases significantly, reducing the overall control overhead of re-convergence.

3.4.2 Propagating Failure Notifications

To determine the ancestors that receive a failure notification, we consider the effect of a link failure along an in-flight packet's intended path. Shortest path routing will send packets up and back down the tree, so we consider both the upward and the downward path segments.

If a link along the upward segment of a packet's path fails, the path simply changes on the fly. This is because each of a switch's upward-facing ports leads to a potentially different subset of L_n switches. In Section 3.2, we introduced the requirement that all L_n switches connect at least once to all L_{n-1} pods, so all L_n switches ultimately reach all hosts. As such, a packet can travel upward towards any L_n switch and therefore its upward path can change on the fly in response to link failure. The switch at the bottom of the failed link can simply select an alternate upward-facing output port. Therefore, no failure notifications are necessary to support re-routing of in-flight packets on the upward segments of their paths.

The case in which a link fails along the downward segment of a packet's intended path is somewhat more complicated. If a failure occurs below a downward-moving packet's current location, and if there is added fault tolerance between the packet's location and the failure, then there is an opportunity to re-route around the failure. Consider a failure that occurs between L_i and L_{i-1} along a packet's intended downward path. Fault tolerance properties below L_i are not relevant, as the packet needs to be diverted *at or before* reaching L_i in order to avoid the failure. However, if there is added fault tolerance at or above L_i, nearby switches can route around the failure, according to the following cases:

1. $c_i>1$: The failed link is at a level with added fault tolerance.
2. $c_i=1$, $c_{i+1}>1$: The closest added fault tolerance is immediately above the failure.
3. $c_i=1$, $c_f>1$, for some $f>i+1$: The nearest level with additional links is more than one hop above.

Case 1: This case corresponds to the failure of link $e-f$ in Figure 3.4. When the packet reaches switch e, e realizes that the intended link $e-f$ is unavailable and instead uses its second connection to $f's$ pod, through h. By definition of a pod, h has downward reachability to the same set of descendants as f and therefore can reach the packet's intended destination. Since e is incident on the failed link, it does not need to propagate any notifications.

Case 2: Case (2) corresponds to the failure of link $f-g$ in Figure 3.4. In this case, if the packet travels all the way to f it will be dropped. But if switch e learns of the failure of $f-g$ before the packet's arrival, it chooses the alternate path through f's pod member h. To allow for this, when f notices the failure of link $f-g$, it should notify any parent (e.g. e) that has a second connection to f's pod (e.g. via h).

Case 3: Finally, Figure 3.5 shows an example of case (3), in which L_2 link $f-g$ fails and the closest added fault tolerance is at L_4. Here, the closest switch to f that can route around the failure is d. Upon a packet's arrival, d can select the path $d-i-h-g$, ultimately reaching the packet's destination. While the fault tolerance is located further from the failure in this case than in case (2), the goal is the same: f notifies any ancestor (e.g. d) with a downward path to another member of f's pod.

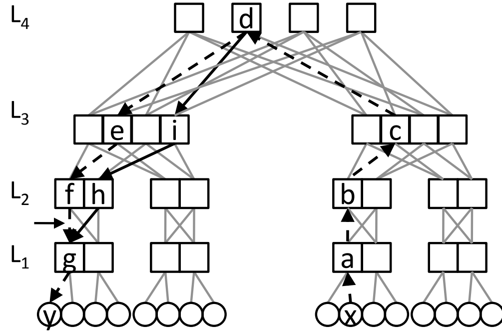


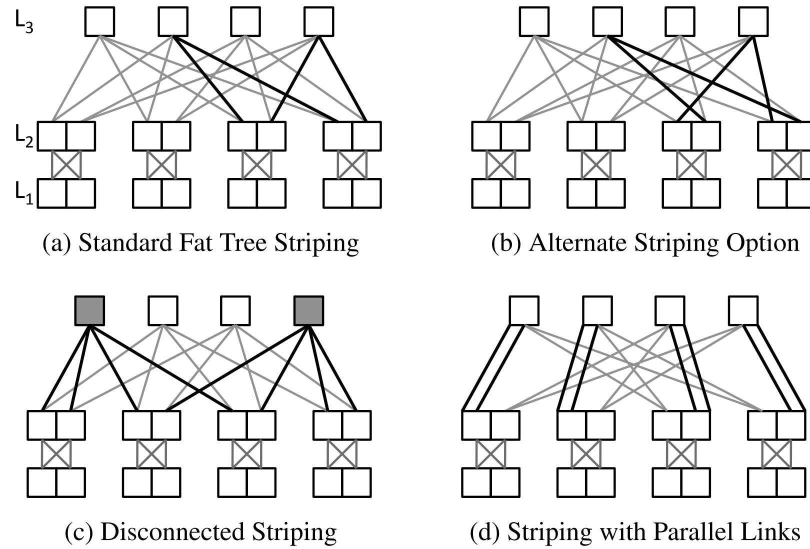
Figure 3.5: 4-Level, 4-Port Aspen Tree with FTV= <1,0,0>

To generalize, when a link from L_i switch s to L_{i-1} neighbor t fails, s first determines whether it has non-zero fault tolerance. If so, it subsequently route all packets intended for t to an alternate member of t's pod. Otherwise, s passes a notification of the failure (indicating the hosts that it no longer reaches) upwards. If an ancestor that receives this notification has alternate paths to these hosts (via an alternate member of s's pod), it adjusts its local state accordingly. Otherwise it forwards the notification upwards. Overall, the complexity of incorporating these notifications is minimal.

3.5 Wiring the Tree: Striping

In Section 3.2, we discussed ways to generate Aspen trees in terms of switch count and placement, and the number of connections between switches at adjacent levels. Here, we consider the organization of connections between switches, a process we refer to as *striping*. We have deferred this discussion until now because of the topic's dependence on the techniques described in Section 3.4 for routing around failures.

Striping refers to the distribution of connections between an L_i pod and neighboring L_{i-1} pods. For instance, consider the striping pattern between L_3 and L_2 in the 3-level tree of Figure 3.6a. The leftmost switch in each L_2 pod connects to the leftmost two L_3 switches, whereas the rightmost switch in each L_2 pod connects to the rightmost two L_3 switches. On the other hand, Figure 3.6b shows a different connection pattern for the switches in the rightmost two L_2 pods, as indicated with the darkened lines.



(a) Standard Fat Tree Striping (b) Alternate Striping Option

(c) Disconnected Striping (d) Striping with Parallel Links

Figure 3.6: Striping Examples for a 3-Level, 4-Port Tree
(Hosts are omitted for clarity.)

Striping can affect connectivity, over-subscription ratios, and the effectiveness of additional fault tolerance links in hierarchical topologies. Some striping schemes even disconnect switches at one level from pods at the level below. In fact, we made a striping assumption in Section 3.2 to avoid exactly this scenario, by introducing the constraint that each L_n switch connects to each L_{n-1} pod at least once. The striping scheme in Figure 3.6c violates this constraint, as the two shaded L_3 switches do not connect to all L_2 pods. Striping patterns can include parallel links, as in Figure 3.6d. Each L_3 switch connects twice to one neighboring L_2 pod, via parallel connections to a single pod member.

Introducing additional fault tolerance into an Aspen tree increases the number of links between switches and pods at adjacent levels, thus increasing the set of possibilities for distributing these connections. Since the techniques of Section 3.4 rely on the existence of ancestors common to a switch s incident on a failed link and alternate members of s's pod, a correct striping policy must yield such common ancestors. Specifically, this is necessary for routing around failures in cases (2) and (3), in which the fault tolerance at the level of the failure is zero, with non-zero fault tolerance higher up in the tree.

In case (2) (Figure 3.4), the additional fault tolerance is immediately above the failed link. Here, the packet can be successfully re-routed by the common L_3 parent shared by f and h (i.e. e). Had e simply had duplicate parallel connections to f, it would not been able to route around this failure. In general, there will be a common parent whenever the striping is not *entirely* comprised of parallel links between a switch and each neighboring pod below. That is, it is possible to route around a single failure between L_i and L_{i-1} if the L_{i+1} fault tolerance is x, including up to $x-1$ parallel links to an L_i pod member and at least one link to alternate member.

Case (3) (Figure 3.5) is more complicated. We again require that switches f and h have at least one ancestor in common, but this ancestor is further above f and h in the tree. For re-routing to work correctly, the following striping policy must hold: *For every level L_i with minimal connectivity to L_{i-1}, if $L_{f:f>i}$ is the closest fault-tolerant level above L_i, each L_i switch s shares at least one L_f ancestor a with another member of s's pod, t.*

3.6 Evaluation

We explore the tradeoffs between convergence time and scalability in Aspen trees, and consider trees that provide a significant reduction in convergence time at a reasonable scalability cost.

3.6.1 Convergence versus Scalability

An Aspen tree with added fault tolerance, and therefore an FTV with non-zero entries, has the ability to react to failures locally. This eliminates the need for global re-convergence of broadcast-based routing protocols on failure, and instead relies on simple failure notifications to a small set of switches close to a failure. These messages require less processing time and travel shorter distances in the tree to fewer nodes, significantly reducing convergence time and control overhead.

If the fault tolerance at L_f is non-zero, then switches at L_f can route around failures that occur at or below L_f, provided a switch incident on an L_i failure notifies its L_f ancestors to use alternate routes. So, the convergence time for a fault between L_i

and L_{i-1} is simply the set of network delays and the cost of processing time for each switch along an $(f-i)$-hop path. Adding extra links at the closest possible level L_f above expected failures at L_i minimizes this convergence time.

The cost of adding this fault tolerance is in the overall scalability of the tree, both in terms of host support and hierarchical aggregation. For each level with an FTV entry $x > 0$, the maximum possible number of hosts is reduced by a multiplicative factor of $\frac{1}{x+1}$. The tree's inherent hierarchical aggregation changes identically.

We begin with the small example of $k = 6$ and $n = 4$ in order to explain the evaluation process. For each possible 4-level, 6-port Aspen tree, we consider the FTV and correspondingly, the distance that updates would have to travel in response to a failure at each level. For instance if there is non-zero fault tolerance between L_i and L_{i-1} then the distance for failures at L_i is 0 whereas the distance for failures at L_{i-2} is 2. If there is no area of non-zero fault tolerance above a level, we assume that updated routing information travels up the tree and back down to L_1, as in a traditionally-defined fat tree. We omit trees for which any of the variables introduced in Section 3.2 are not integers. We average this propagation distance across all levels of the tree[1] to give a metric for expressing overall convergence time.

We consider the scalability cost of adding fault tolerance, by counting the number of hosts missing in each Aspen tree as compared to a traditional fat tree with the same switch size and number of levels. We elected to consider hosts removed, rather than hosts remaining, so that the compared measurements (convergence time and hosts removed) are both minimal in the ideal case and can be more intuitively depicted graphically. Figure 3.7 shows this convergence/scalability tradeoff; for each possible FTV option, the figure displays the average convergence time (in hop count) across all levels, alongside the number of hosts missing with respect to a traditional fat tree. We normalize the values shown to percentages of the worst case. We omit graphs for hierarchical aggregation as the relationship to fault tolerance is identical to that of host count.

Thus, we have a spectrum of Aspen trees. At one end of this spectrum is the tree with no added fault tolerance (FTV = <0,0,0>) but with no hosts removed. At the other end we have trees with high fault tolerance (all failure reactions are local) but

[1] We do not include first-hop failures, as our techniques can not help in such situations.

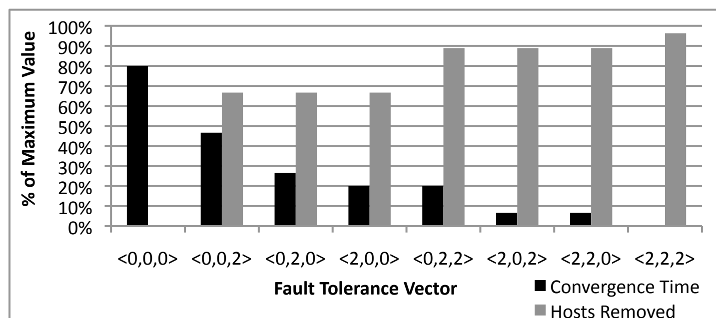
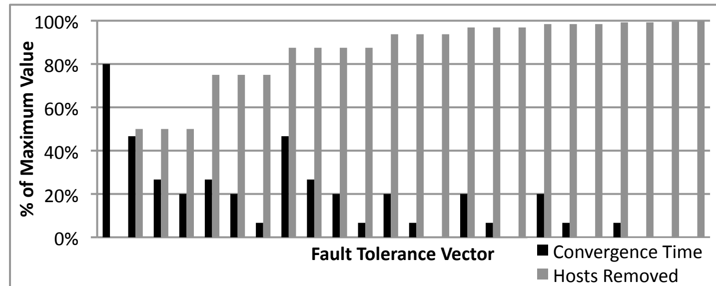


Figure 3.7: Host Removal and Convergence Time vs. Fault Tolerance in 4-Level, 6-Port Aspen Trees (Max Hops=5, Max Hosts = 162)

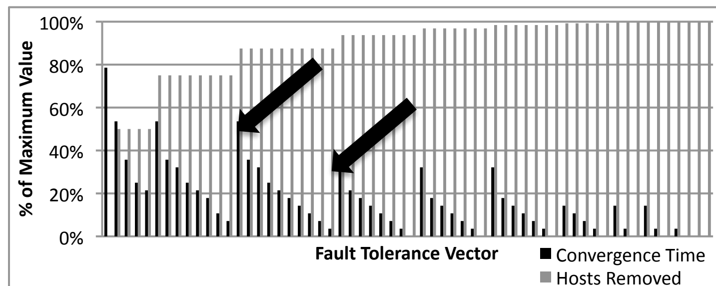
with over 95% of the hosts removed. In the middle we find interesting cases: in these, not every failure can be handled locally, but those failures that are not handled locally can be masked within a small and limited number of hops. The convergence times for these middle-ground trees are significantly reduced from that of a traditional fat tree, but substantially fewer hosts are removed than for the tree with all local failure reactions.

An interesting observation is that there are often several ways to generate the same host count, but with different convergence times. This is shown in the second, third and fourth entries of Figure 3.7, in which the host counts are all $\frac{1}{3}$ of that for a minimal fat tree, but the average update propagation distance varies from 1 to 2.3 hops. A network designer constrained by the number of hosts to support should select a tree that yields the smallest convergence time for the required number of hosts. Similarly, there are cases in which the convergence times are identical but the host count varies, e.g. FTVs <2,0,0> and <0,2,2>. Both have average update propagation distances of 1, but the former supports 54 hosts and the latter only 18. If constrained to a particular convergence time, we recommend the tree with the largest number of hosts.

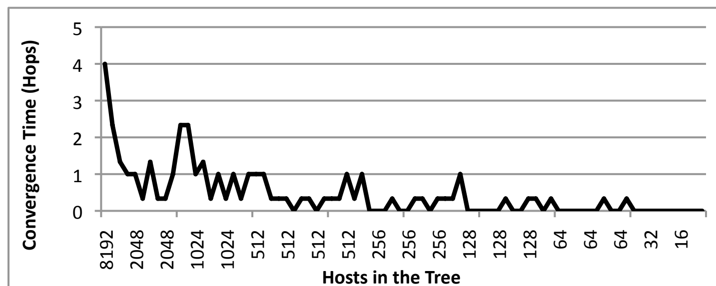
We now examine more realistically sized Aspen trees. In practice, we expect trees with $3 \leq n \leq 7$ levels and $16 \leq k \leq 128$ ports per switch, in support of tens of thousands of hosts. Figures 3.8a and 3.8b show graphs similar to that of Figure 3.7, for 16-port trees of depths 4 and 5, respectively. Because of the large number of configuration options for these values of k and n, we often find that numerous FTVs all correspond to a single (host count, convergence time) pair. We collapsed all such duplicates into single entries, and because of this, we removed the FTV labels from the resulting graphs.



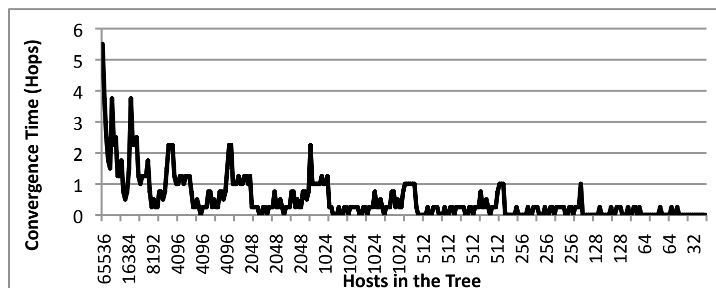
(a) Host Removal and Convergence Time vs. Fault Tolerance in 4-Level, 16-Port Aspen Trees (Max Hops=5, Max Hosts = 8,192)



(b) Host Removal and Convergence Time vs. Fault Tolerance in 5-Level, 16-Port Aspen Trees (Max Hops=7, Max Hosts = 65,536) (Arrows show varying convergence time for single host count value.)



(c) Convergence Time vs. Host Count in 4-Level, 16-Port Trees



(d) Convergence Time vs. Host Count in 5-Level,16-Port Trees

Figure 3.8: Convergence vs. Scalability for 4 and 5-Level, 16-Port Aspen Trees

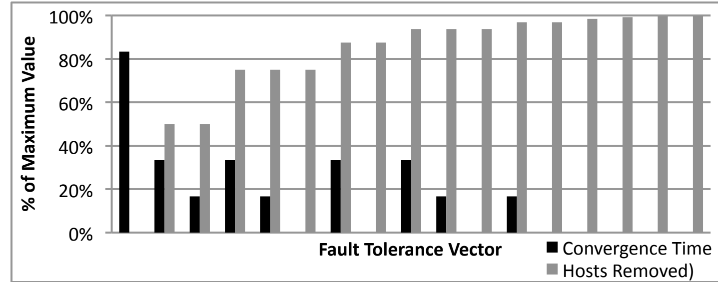
Figures 3.8a and 3.8b show the same trends as does Figure 3.7, but since there are more options for generating trees, the results are perhaps more apparent. As we move from left to right in the bar graphs, we remove more hosts. However, the host removal bars in the graph resemble step functions; each individual number of hosts removed corresponds to several different values for average convergence time. We mark one such step in Figure 3.8b with arrows. In this case, if we are constrained by the number of hosts to support, we would select the rightmost entry in the corresponding step, i.e. that with the minimum convergence time.

Figures 3.8c and 3.8d directly compare convergence time and host count in order to provide more intuition about the relationship between the two. Note that for these figures, the x-axis is in terms of hosts present in the tree and that the figures show numerical values rather than percentages. These graphs show the same trends as those of Figures 3.8a and 3.8b; convergence time decreases with the number of hosts. This relationship is non-linear and there are many local minima and maxima along each graph. A local maximum represents a case in which the host count is similar to that for nearby points, but convergence time is high. Local maxima therefore correspond to less desirable trees. A similar argument shows that local minima represent (relatively) better trees.

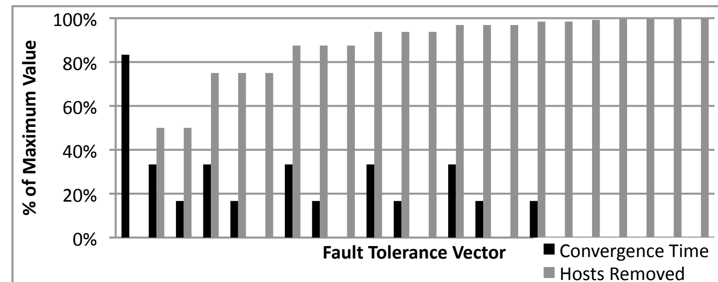
Figure 3.9 shows trees with larger switches ($k = 32$ and 64) but with smaller values for the depth of the tree ($n = 3$) so as to keep our results in line with the topology sizes we expect to see in practice. For these graphs, we again collapsed duplicates and thus had to omit the FTV labels, but since the small number of levels limits the number of possible trees, there are fewer entries than in the graphs of Figure 3.8. These results again show that with only modest reductions to host count, the reaction time of a tree can be significantly reduced.

3.6.2 Recommended Aspen Trees

We showed in Section 3.4 that the most useful and efficient fault tolerance is both above failures above failures and as close to failures as possible. We formalize this in terms of the FTV. The most fault-tolerant tree has an FTV with all maximal (and non-zero) entries. However, an optimal FTV may come at too high of a scalability cost.



(a) Host Removal and Convergence Time vs. Fault Tolerance in 3-Level, 32-Port Aspen Trees (Max Hops=3, Max Hosts = 8,192)



(b) Host Removal and Convergence Time vs. Fault Tolerance in 3-Level, 64-Port Aspen Trees (Max Hops=3, Max Hosts = 65,536)

Figure 3.9: Convergence vs. Scalability for 3-Level, 32 and 64-Port Aspen Trees

To enable usable and efficient fault tolerance, in FTVs with non maximal entries it is best to cluster non-zero values to the left while simultaneously minimizing the lengths of series of contiguous zeros. For instance, if we can put only two non-zero entries in an FTV of length 6, the ideal placement would be <1,0,0,1,0,0>. There are at most two contiguous zeros, so updates propagate a maximum of two hops, and each 0 has a corresponding 1 to its left, so no failure leads to global re-convergence.

One Aspen tree in particular bears special mention. Given our goal of keeping fault tolerance at upper tree levels (and towards the left of an FTV), the biggest value-add with minimal scalability cost is the addition of extra links at the single level of the tree that can accommodate all failures, i.e. the top level. A tree with only L_n fault tolerance has an FTV of <1,0,0,...> and a DCC of 2, and therefore supports half as many hosts as does a minimal fat tree. The average convergence propagation distance is cut in half for this tree from that of a traditional fat tree, and more importantly, all updates only travel upward rather than moving upward and then fanning out to all switches in the tree.

There are pathological tree options in which added fault tolerance can not help, and therefore is clearly not worth its cost in scalability. In these trees we have bottleneck pods, i.e. pods with only a single switch, at high levels in the tree. If a failure occurs immediately below a bottleneck pod, no amount of fault tolerance higher in the tree can help as there are not alternate pod members to route around the failure. We do not expect to see such trees in practice.

3.7 Related Work

There are two ways to handle failures in a network. We can structure the network so that failures only minimally impact service or we can work around them when they occur. Historically, the approach has been to work around failures, either by re-routing packets on the fly, or by using pre-computed backup paths.

3.7.1 Alternative Routing Techniques

Bounce routing techniques work around a failure by temporarily sending packets away from a destination in order to avoid a failed link. For instance, consider a fat tree in which switch s is connected to pods p and q below. Suppose that s needs to send a packet through pod p but that its link to p has failed. s can instead bounce the packet through pod q and back up to one of s's alternate pod members, so long as the appropriate connections and striping patterns are in place. This small 3-hop detour easily works around the failed link.

However, such a detour does not follow shortest path-style routing and introduces the need for additional forwarding logic in order to avoid loops and deadlocks, especially when combined with flow control algorithms [20, 37]. On the other hand, engineering the topology to enable local failure reaction avoids the software complexity and robustness difficulties of bounce routing techniques, but at a cost in scalability.

Failure carrying packets (FCP) [45] eliminate the convergence process after a failure by allowing data packets to carry failure information. FCPs leverage the fact that an intradomain ISP network has a set of relatively stable links, in terms of presence, if not availability. Therefore, if all routers in the topology know the full physical network

topology, they simply need to learn the set of links not currently available for a given packet. FCP provide guaranteed eventual delivery of packets if the graph does not become disconnected, which solves the problem of temporary disconnection. However, the implementation and deployment cost of introducing a new data plane may hinder the adoption of FCP in the data center, and the paths ultimately taken by packets can be long.

Data-driven connectivity (DDC) [50] addresses connectivity issues separately from the more far reaching distributed computations of the control plane (e.g. load balancing, shortest path calculation) with a scheme somewhat similar to bounce routing. We share a similar goal to DDC's "ideal connectivity" in which packets are not dropped unless the destination is physically unreachable, however we choose orthogonal approaches. In the authors' evaluation of various topologies, they note that fat trees lack "resilient nodes" that provide multiple output ports to a destination. In fact, by modifying fat trees, we effectively increase the average resilience across all nodes.

Multi-path TCP (MPTCP) [60] breaks individual flows into *subflows*, each of which may be sent via a different path based on current congestion conditions in the network. A path that includes a failed link will appear to be congested since a portion of it offers no bandwidth, and MPTCP will move any corresponding subflows to another path. A downside of MPTCP is its reliance on host modifications.

The idea behind this work derives from fast failure recovery [44] techniques in WANs. Our approach is to engineer data center topologies so as to enable FFR for link failures.

A difficulty with routing techniques that work around failures is that they may (temporarily) result in long paths, up to 50% longer in the case of DDC. Because we base our approach on a topology with fixed path lengths, we avoid this issue.

3.7.2 Backup Paths

Another way to improve the fault tolerance of a network is to establish backup paths for use when a primary path (or link along the path) for a flow fails. Many works consider this topic in the context of either ad hoc networks or resource allocation for performance guarantees. Generally, such works fall into two camps. Some advocate as-

signing backup paths at the start of a flow [31, 32, 43, 68, 75], so that the flow continues to function after the failure of its primary path and even $N-1$ of its N backup paths. This comes at the cost of potentially wasting resources that are reserved for backup paths but are rarely or never used, as well as the time cost of determining backup paths on flow entry. Also, for flows with strict performance requirements, it is difficult to pre-compute appropriate backup paths in the face of dynamic traffic.

On the other hand, some approaches [10, 11, 74] establish a backup path on the fly at the time of a failure. The downsides of this are the possibility of contention for new paths upon failure (especially if the failure affects multiple flows that all try to establish new paths at once), the time to calculate new paths upon failure, and the fact that recovery is not guaranteed for any given flow. However, this type of solution does not have the drawback of potentially wasting valuable bandwidth that may never be needed, nor the time cost of setting backup paths initially.

The authors of [10] and [11] consider dynamically recovering from faults by recalculating new routes on the fly. They study this process along several metrics, varying the portion of the path that is recalculated, the timing of recalculation, and the possibility of retrying recalculation. Their findings show that when one physical link failure can affect multiple flows, local reaction is faster. This finding supports our belief that it is ideal to keep the switches that react to a failure as close as possible to the failure itself. A concern with local re-routing in general is the use of longer paths; the regular structure of our Aspen trees renders this a non-issue.

The authors of [74] present a hybrid method, calculating backup paths prior to failure, but admitting flows once a primary path is found without waiting for backup path calculation to complete. Backup paths are not complete paths, but rather "patches" that avoid failed links along portions of a path. While this approach differs from ours in its use of source routing, it is similar in that it enables local failure reaction.

3.7.3 High Performance Computing Topologies

Our topologies derive from the initial presentations of fat trees as non-blocking architectures for communication in supercomputers [19, 47]. The traditionally defined fat tree of Figure 3.1 comes from DeHon's Butterfly Fat-Tree [22]. A number of works

have extended traditional fat tree topologies by essentially raising c_i from 1 to 2 uniformly at all levels of the tree. Upfal's multi-butterfly networks [71] and Leighton et al's routing algorithms for these topologies [46] show examples of these subsets of our topologies, as do Goldberg et al's splitter networks [26]. These works consider path existence for message scheduling but none examine the applicability of the topologies in terms of running real protocols (e.g. IP) over modern switch hardware in today's data centers.

3.8 Summary

In this chapter, we introduce a new class of data center topologies called Aspen trees. Aspen trees are based on fat trees, but are a more general class of multi-rooted tree topologies in which a network designer can tune the tradeoffs between scalability and fault tolerance to meet the requirements for a particular situation. The additional fault tolerance in an Aspen tree comes from redundant links added to a subset of levels in the tree. These additional links provide alternate paths in the case of one or more link failures. We present a protocol to generate an Aspen tree, given scalability and fault tolerance requirements, and we explore the fault tolerance and scalability properties of several example topologies. Finally, we offer a particular type of Aspen tree that gives excellent fault tolerance with minimal scalability cost.

3.9 Acknowledgment

Chapter 3, in part, contains material submitted for publication as "Scalability vs. Fault Tolerance in Apsen Trees." Walraed-Sullivan, Meg; Vahdat, Amin; Marzullo, Keith. The dissertation author was the primary investigator and author of this paper.

Chapter 4

ALIAS: Scalable, Decentralized Label Assignment for Data Centers

In this chapter, we present the design and implementation of ALIAS, a scalable, automatic and decentralized protocol for labeling switches and hosts in a hierarchically structured data center network. The labels assigned by ALIAS encode both the locations of switches and hosts within the network as well as the path multiplicity inherent in a hierarchical topology. As such, these labels form a basis for scalable routing and forwarding within the data center while simultaneously reducing the management overhead on the network administrator. ALIAS provides the following features:

- *Automatic, decentralized host labeling*: Switches learn their positions in the topology and automatically group themselves into *hypernodes* of high connectivity. Each hypernode (HN) is uniquely numbered among all hypernodes, serving as the basis for hierarchical label assignment for hosts. This proceeds via pair-wise message exchange between immediate neighbors, with no reliance on centralized components, manual configuration, topology blueprints or flooding-based routing protocols.

- *Scalable route discovery*: Each switch discovers routes to remote hosts. For distant switches, it is sufficient to learn the route to the correct hypernode. We distribute this reachability information using pair-wise (broadcast-free) exchanges along a hierarchical control path established during the host labeling process.

- *Fault tolerance and rapid convergence*: ALIAS converges to a correct host labeling with global reachability (assuming appropriate underlying connectivity) after arbitrary flux in the topology. It limits the effects of topology changes to a small surrounding portion of the network.

- *Unmodified host and switch compatibility*: While ALIAS would be simplified with host support or switch hardware modification, a goal of the work is to lower the barrier to adoption. As such, ALIAS runs entirely in switch software and accesses the underlying hardware through standard APIs such as OpenFlow [1].

We evaluate ALIAS through model checking, simulations and practical experiments. We show that ALIAS successfully provides scalable, decentralized data center addressing and communication while simultaneously reducing the management burden.

4.1 ALIAS

The goal of ALIAS is to automatically assign globally unique, topologically meaningful host *labels* that the network can internally employ for efficient forwarding. We aim to deliver one of the key benefits of IP addressing—hierarchical address assignments such that hosts with the same prefix share the same path through the network and a single forwarding table entry suffices to reach all such hosts—without requiring manual address assignment and subnet configuration. A requirement in achieving this goal is that ALIAS be entirely decentralized and broadcast-free. At a high level, ALIAS switches automatically locate clusters of good switch connectivity within network topologies and assign a shared, non-conflicting prefix to all hosts below such pockets. The resulting hierarchically aggregatable labels lead to compact switch forwarding entries.[1] Labels are simply a reflection of current topology; ALIAS updates and reassigns labels to affected hosts based on topology dynamics.

[1] We use the terms “label” and “address” interchangeably.

4.1.1 Environment

ALIAS overlays a logical hierarchy on its input topology. Within this hierarchy, switches are partitioned into *levels*; each switch belongs to exactly one level. Switches connect predominantly to switches in the levels directly above or below them, though pairs of switches at the same level (peers) may connect to each other via *peer links*.

One high-level dichotomy in multi-computer interconnects is that of direct versus indirect topologies [66]. In a direct topology, a host can connect to any switch in the network. With indirect topologies, only a subset of the switches connect directly to hosts; communication between hosts connected to different switches is facilitated by one or more intermediate switch levels. We focus on indirect topologies because such topologies appear more amenable to automatic configuration and because they make up the vast majority of topologies currently deployed in the data center [4, 13, 18, 28, 47, 56, 57]. Figure 4.1 gives an example of an indirect 3-level topology, on which ALIAS has overlaid a logical hierarchy. In the figure, S_x and H_x denote the unique IDs of switches and hosts, respectively.

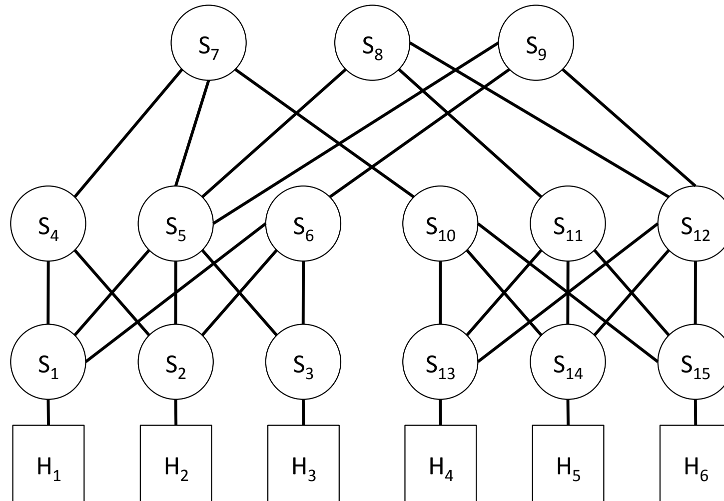


Figure 4.1: Sample Multi-Rooted Tree Topology

A host with multiple network interfaces may connect to multiple switches, and will have separate ALIAS labels for each interface. ALIAS also assumes that hosts do not play a switching role in the network and that switches are programmable (or run software such as OpenFlow [1]).

4.1.2 Protocol Overview

ALIAS first assigns topologically meaningful *labels* to hosts, and then enables communication over these labels. As with IP subnetting, topologically nearby hosts share a common prefix in their labels. In general, longer shared prefixes correspond to closer hosts. ALIAS groups hosts into related clusters by automatically locating pockets of strong connectivity in the topology—groups of switches separated by one level in the hierarchy with full bipartite connectivity between them. However, even assigning a common prefix to all hosts connected to the same leaf switch can reduce the number of required forwarding table entries by a large factor (e.g., the number of host-facing switch ports multiplied by the typical number of virtual machines on each host).

Hierarchical Label Assignment

ALIAS labels are of the form $(c_{n-1} \ldots c_1.\texttt{H}.\texttt{VM})$, wherein the first $n-1$ fields encode a host's location within an n-level topology, the `H` field identifies the port to which each host connects on its local switch, and the `VM` field provides support for multiple VMs multiplexed onto a single physical machine. ALIAS assigns these hierarchically meaningful labels by locating clusters of high connectivity and assigning to each cluster (and its member switches) a *coordinate*. Coordinates then combine to form host labels; the concatenation of switches' coordinates along a path from the core of the hierarchy to a host make up the c_i fields of a host's label.

Prior to selecting coordinates, switches first discover their levels within the hierarchy, as well as those of their neighbors. Switches i hops from the nearest host are in level L_i, as indicated by the L_1, L_2 and L_3 labels in Figure 4.2. Once a switch establishes its level, it begins to participate in coordinate assignment. ALIAS first assigns unique `H-coordinates` to all hosts connected to the same L_1 switch, creating multiple one-level trees with an L_1 switch at the root and hosts as leaves. Next, ALIAS locates sets of L_2 switches connected via full bipartite graphs to sets of L_1 switches, and groups each such set of L_2 switches into a *hypernode* (HN). The intuition behind hypernodes is that all L_2 switches in an L_2HN can reach the same set of L_1 switches, and therefore these L_2 switches can all share the same prefix. This process continues up the hierarchy, grouping L_i switches into L_iHNs based on bipartite connections to L_{i-1}HNs.

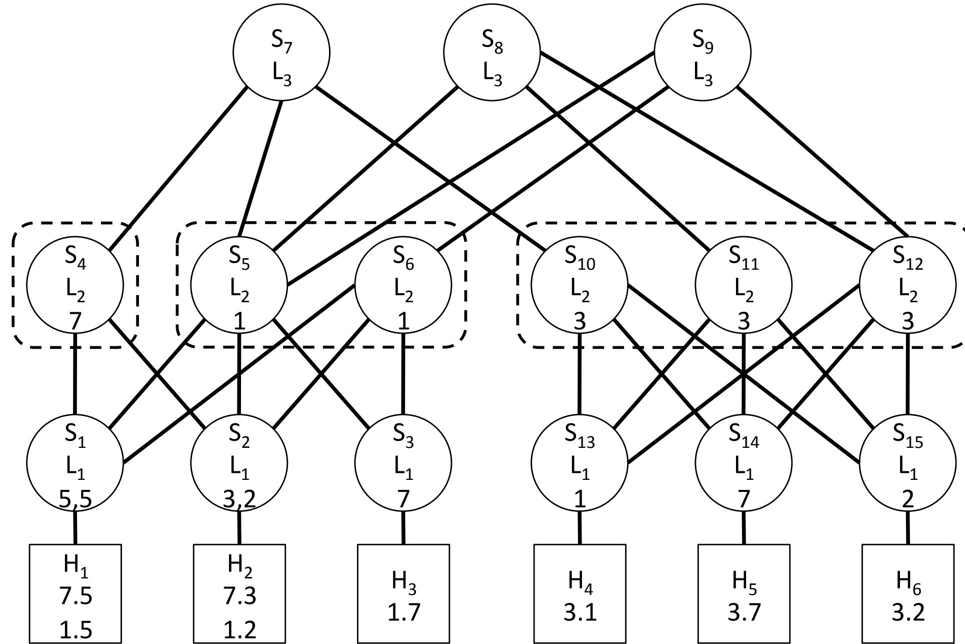


Figure 4.2: ALIAS Applied to a Sample Topology: Level and Label Assignments

Finally, ALIAS assigns unique coordinates to switches, where a coordinate is a number shared by all switches in an HN and unique across all other HNs at the same level. By sharing coordinates among HN members, ALIAS leverages the hierarchy present in the topology and reduces the number of coordinates used overall, thus collapsing forwarding table entries. Switches at the core of the hierarchy do not require coordinates and are not grouped into HNs. L_1 switches select coordinates without being grouped into HNs. Further, we employ an optimization (Section 4.2.2) that assigns multiple coordinates to an L_i switch, one per neighboring L_{i+1}HN.

When the physical topology changes due to a switch, host or link failure, configuration changes, or any other circumstances, ALIAS adjusts all label assignments and forwarding entries as necessary (Sections 4.2.3 and 4.3.3).

Figure 4.2 shows a possible set of coordinate assignments and the resulting host label assignments for the topology of Figure 4.1; only topology-related prefixes are shown for host labels. For this 3-level topology, L_2 switches are grouped in HNs (as shown with dotted lines), and L_1 switches have multiple coordinates corresponding to multiple neighboring L_2HNs. Hosts have multiple labels corresponding to the L_2HNs connected to their ingress L_1 switches.

Communication

ALIAS's labels can be used in a variety of routing and forwarding contexts, such as tunneling, IP-encapsulation or MAC address rewriting [56]. We have implemented one such communication technique (based on MAC address rewriting) and present here an example of this communication.

An ALIAS packet's traversal through the topology is controlled by a combination of forwarding (Section 4.3.2) and addressing logic (Section 4.2.2) Consider the topology shown in Figure 4.2. A packet sent from H_4 to H_2 must flow upward to one of S_7, S_8 or S_9, and then downward towards its destination. First, H_4 sends an ARP request to its first-hop switch, S_{13}, for H_2's label (Section 4.3.3). S_{13} determines this label (with cooperation from nearby switches if necessary) and responds to H_4. H_4 can then forward its packet to S_{13} with the appropriate label for H_2, for example `(1.2.1.0)` if H_2 is connected to port 1 of S_2 and has `VM` coordinate 0. At this point, forwarding logic moves the packet to one of S_7, S_8 or S_9, all of which have a downward path to H_2, The routing protocol (Section 4.3.1) creates the proper forwarding entries at switches between H_4 and the core of the network, so that the packet can move towards an appropriate L_3 switch. Next, the packet is forwarded to one of S_5 or S_6, based on the `(1.x.x.x)` prefix of H_2's label. Finally, based on the second field of H_2's label, the packet moves to S_2 where it can be delivered to its destination.

4.1.3 Multi-Path Support

Multi-rooted trees provide multiple paths between host pairs, and routing and forwarding protocols should discover and utilize these multiple paths for good performance and fault tolerance. ALIAS provides multi-path support for a given destination label via its forwarding component (Section 4.3.2). For example, in Figure 4.2, a packet sent from H_4 to H_2 with destination label `(1.2.1.0)` may traverse one of five different paths.

An interesting aspect of ALIAS is that it enables a second class of multi-path support: hosts may have multiple labels, where each label corresponds to a set of paths to a host. Thus, choosing a label corresponds to selecting a *set of paths* to a host. For

example, in Figure 4.2, H_2 has two labels. Label `(1.2.1.0)` encodes 5 paths from H_4 to H_2, and label `(7.3.1.0)` encodes a single H_4-to-H_2 path. These two classes of multi-path support help limit the effects of topology changes and failures. In practice, common data center fabric topologies will result in hosts with few labels, where each label encodes many paths. Policy for choosing a label for a given destination is a separable issue; we present some potential methods in Section 4.3.3.

4.2 Protocol

ALIAS is comprised of two components, Level Assignment and Coordinate Assignment. These components operate continuously, acting whenever topology conditions change. For example, a change to a switch's level may trigger changes to that switch's and its neighbors' coordinates. ALIAS also involves a Communication Component for routing, forwarding, and label resolution and invalidation; in Section 4.3 we present one of the many possible communication components that might use the labels assigned by ALIAS.

ALIAS operates based on the periodic exchange of *Topology View Messages* (TVMs) between switches. In an n-level topology, individual computations rely on information from no more than $n-1$ hops away.

Listing 4.1 gives an overview of the general state stored at each switch, as well as that related to level assignment. A switch knows its own unique ID and the IDs of its neighbors (lines 1-2). It also records an indication of whether each neighbor is a host or switch (line 3). Switches also know their own levels and those of their neighbors (lines 4-5) as well as the types of links (regular or peer) that connect them to each neighbor (line 6). The values in lines 4-6 of the listing are set by the level assignment protocol (Section 4.2.1).

4.2.1 Level Assignment

ALIAS level assignment enables each switch to determine its own level as well as those of its neighbors, and to detect and mark peer links for special consideration by other components. ALIAS defines an L_i switch to be a switch with a minimum of i hops

Listing 4.1: ALIAS local state

```
1 UID myId
2 UIDSet nbrs
3 Map(UID→NodeType) types

4 Level level
5 Map(UID→Level) levels
6 Map(UID→LinkType) link_types
```

to the nearest host. For convenience, in an n-level topology, L_n switches may be referred to as *cores*. *Regular links* connect L_1 switches to hosts, and L_i switches to switches at $L_{i\pm1}$, while *peer links* connect switches of the same level.

Level assignment is bootstrapped by L_1 switch identification as follows: In addition to sending TVMs, each switch also periodically sends IP pings to all neighbors that it does not know to be switches. Hosts reply to pings but do not send TVMs, enabling switches to detect neighboring hosts. This allows L_1 identification to proceed without host modification. If hosts provided self-identification, then the protocol becomes much simpler. Recent trends toward virtualization in the data center with a trusted hypervisor may take on this functionality.

When a switch receives a ping reply from a host, it immediately knows that it is at L_1 and that the sending neighbor is a host, and updates its state accordingly (lines 3-4, Listing 4.1). If a ping reply causes the switch to change its current level, it may need to mark some of its links to neighbors as peer links (line 6). For instance, if the switch previously believed itself to be at L_2, it must have done so because of a neighboring L_1 switch and its connection to that neighbor is now a peer link.

Based on L_1 identification, level assignment operates via a *wave* of information from the lowest level of the hierarchy upwards; A switch that receives a TVM from an L_1 switch labels itself as L_2 if it has not already labeled itself as L_1, and this process continues up the hierarchy. More generally, each switch labels itself as L_i, where $i-1$ is the minimum level of all of its neighbors.

On receipt of a TVM, a switch s determines whether the source's level is smaller than that recorded for any of its others neighbors, and if so, adjusts its own level assignment (line 4, Listing 4.1). It also updates its state for its neighbor's level and type if necessary (lines 3,5). If s's level or that of its neighbor has changed, it detects and

records any changes to the link types to any of its neighbors (line 6). For instance, if an L_3 switch moves to L_2, links to L_2 neighbors become peer links. Links to L_4 neighbors are no longer legal but are not disabled, as the L_4 neighbors will adjust their level assignments upon receipt of the next TVM from the modified switch. When a switch detects disconnection from a neighbor, it proceeds in a similar fashion, making any necessary level changes and updating link types accordingly.

The presence of unexpected or numerous peer links may indicate a *miswiring*, or erroneous cabling, with respect to the intended topology. If ALIAS suspects a miswiring, it raises an alert (e.g., by notifying the administrator) but continues to operate. In this way, miswirings do not bring the system to a halt, but are also not ignored.

ALIAS's level assignment can assign levels to all switches as long as at least one host is present. Once a switch learns its level, it participates in coordinate assignment.

4.2.2 Label Assignment

An ALIAS switch's *label* is the concatenation of $n-1$ coordinates, $c_{n-1}c_{n-2}\ldots c_2c_1$, each corresponding to one switch along a path from a core switch to the labeled switch. A host's label is then the concatenation of an ingress L_1 switch's label and its own `H` and `VM` coordinates. As there may be multiple paths from the core switches of the topology to a switch (host), switches (hosts) may have multiple labels.

Coordinate Aggregation

Since highly connected data center networks tend to have numerous paths to each host, per-path labeling can lead to overwhelming numbers of host labels. ALIAS creates compact forwarding tables by dynamically identifying sets of L_i switches that are strongly connected to sets of L_{i-1} switches below. It then assigns to these L_i *hypernodes* unique L_i coordinates. By sharing one coordinate among the members of an L_iHN, ALIAS allows hosts below this HN to share a common label prefix, thus reducing forwarding table entries.

An L_iHN is defined as a maximal set of L_i switches that all connect to an identical set of L_{i-1}HNs, via any constituent members of the L_{i-1}HNs. Each L_i switch is a member of exactly one L_iHN. L_2HN grouping are based on L_1 switches rather than

HNs. In Figure 4.3, L_2 switches S_5 and S_6 connect to same set of L_1 switches, namely $\{S_1,S_2,S_3\}$, and are grouped together into an L_2HN, whereas S_4 connects to $\{S_1,S_2\}$, and therefore forms its own L_2HN. Similarly, S_7 and S_8 connect to both L_2HNs (though via different constituent members) and form one L_3HN while S_9 forms a second L_3HN, as it connects only to one L_2HN below.

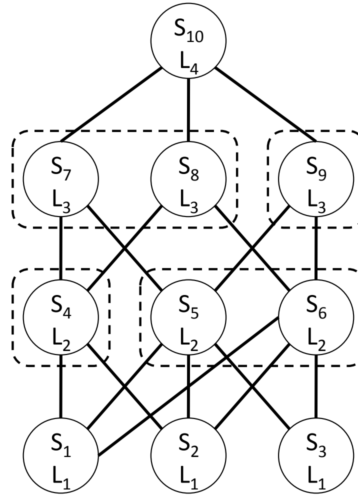


Figure 4.3: ALIAS Hypernodes

Since L_iHNs are defined based on connectivity to identical sets of L_{i-1}HNs, the members of an L_iHN are interchangeable with respect to downward forwarding. This is the key intuition that allows HN members to share a coordinate, ultimately leading to smaller forwarding tables.

ALIAS employs an optimization with respect to HN grouping for coordinate assignment. Consider switch S_1 of Figure 4.3, and suppose the L_2HNs $\{S_4\}$ and $\{S_5,S_6\}$ have coordinates x and y, respectively. Then S_1 has labels of the form $...xc_1$ and $...yc_1$, where c_1 is S_1's coordinate. Since S_1 is connected to both L_2HNs, it needs to ensure that c_1 is unique from the coordinates of all other L_1 switches neighboring $\{S_4\}$ and $\{S_5,S_6\}$ (in this example, all other L_1 switches).

It is helpful to limit the sets of switches competing for coordinates, to decrease the probability of collisions (two HNs selecting the same coordinate) and to allow for a smaller coordinate domain. We accomplish this as follows: S_1 has two coordinates, one corresponding to each of its label prefixes, giving it labels of the form $...xc_1$ and $...yc_2$.

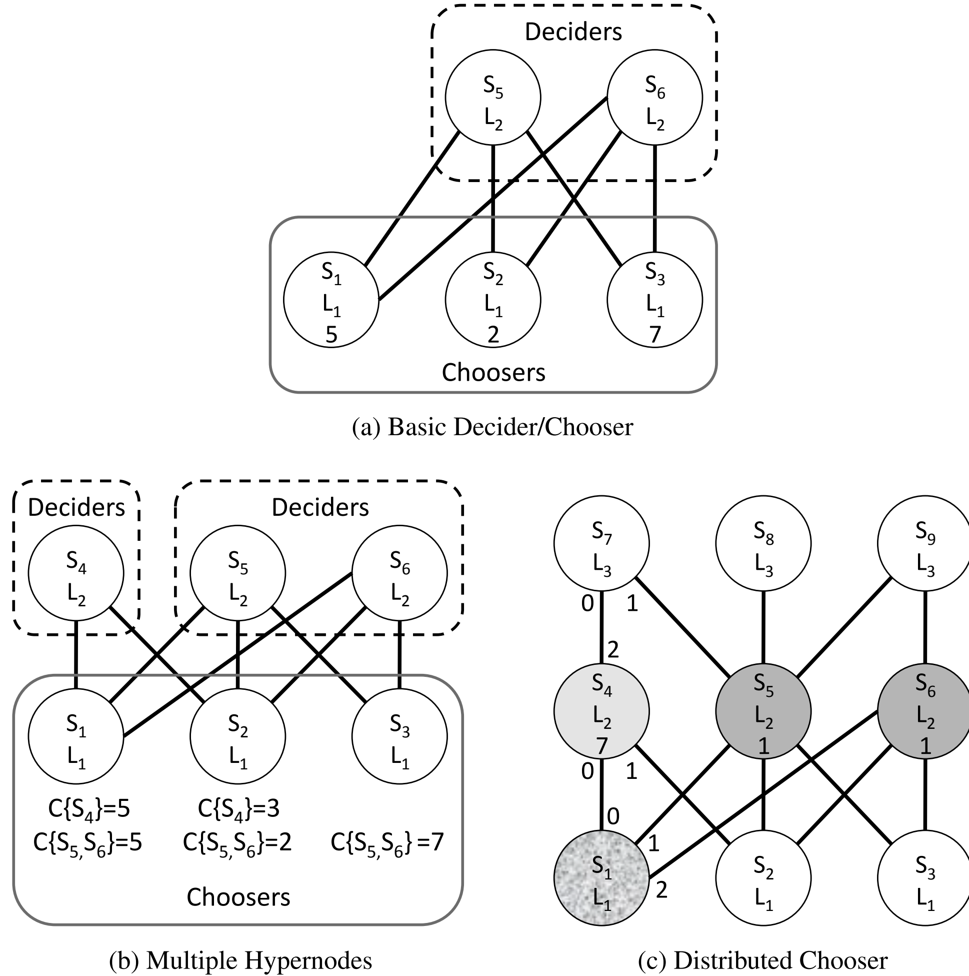
In this way S_1 competes only with S_2 for labels corresponding to HN $\{S_4\}$. In general, ALIAS assigns to each switch a coordinate *per upper neighboring HN*. This reduces coordinate contention without increasing the coordinate domain size.

Decider/Chooser Abstraction

The goal of coordinate assignment in ALIAS is to select coordinates for each switch such that these coordinates can be combined into forwarding prefixes. By assigning per-HN rather than per-switch coordinates, ALIAS leverages a topology's inherent hierarchy and allows nearby hosts to share forwarding prefixes. In order for an L_i switch to differentiate between two lower-level HNs, for forwarding purposes, these two HNs must have different coordinates. Thus, the problem of coordinate assignment in ALIAS is to enable L_iHNs to cooperatively select coordinates that do not conflict with those of other L_iHNs that have overlapping L_{i+1} neighbors. HN members are typically not directly connected to one another, so this task requires indirect coordination.

To explain ALIAS's coordinate assignment protocol, we begin with a simple *Decider/Chooser Abstraction* (DCA), and refine the abstraction to solve the more complicated problem of coordinate assignment. The basic DCA includes a set of choosers that select random values from a given space, and a set of deciders that ensure uniqueness among the choosers' selections. A requirement of DCA is that any two choosers that connect to the same decider select distinct values. Choosers make choices and send these requests to all connected deciders. Upon receipt of a request from a chooser, a decider determines whether it has already stored the value for another chooser. If not, it stores the value for the requester and sends an acknowledgment. If it has already stored the requested value for another chooser, the decider compiles a list of hints of already selected values and sends this list with its rejection to the chooser. A chooser reselects its value if it receives a rejection from any decider, and considers its choice *stable* once it receives acknowledgments from all connected deciders.

We employ DCA within a single L_iHN and its L_{i-1} neighbors to assign coordinates to the L_{i-1} switches, as in Figure 4.4a. The members of L_2HN $\{S_5,S_6\}$ act as deciders for L_1 choosers, S_1, S_2 and S_3, ensuring that the three choosers select unique L_1-coordinates.



(a) Basic Decider/Chooser

(b) Multiple Hypernodes

(c) Distributed Chooser

Figure 4.4: Decider/Chooser Abstraction in ALIAS

Recall that as an optimization, ALIAS assigns to each switch multiple coordinates, one per neighboring higher level HN. We extend the basic DCA to have switches keep track of the HN membership of upward neighbors, and to store coordinates (and an indication of whether a choice is stable) on a per-HN basis. This is shown in Figure 4.4b, where each L_1 switch stores information for all neighboring L_2HNs. The figure includes two instances of DCA, that from Figure 4.4a and that in which S_4 is a decider for choosers S_1 and S_2.

Finally, we refine DCA to support coordinate sharing within an HN. Since each member of an HN may connect to a different set of higher level switches (deciders), it is necessary that all HN members cooperate to form a distributed chooser. HN mem-

bers cooperate with the help of a deterministically selected *representative* L_1 *switch* (for example, the L_1 switch with the lowest MAC address of those connected to the HN). L_1 switches determine whether they represent a particular HN as a part of HN grouping calculations.

The members of an L_iHN, and the HN's representative L_1 switch collaborate to select a shared coordinate for all HN members as follows: The representative L_1 switch performs all calculations and makes all decisions for the chooser, and uses the HN's L_i switches as *virtual channels* to the deciders. L_iHN members gather and combine hints from deciders, passing them down to the representative L_1 switch for calculations. The basic chooser protocol introduced above is extended to support reliable communication over the virtual channels between the representative L_1 switch and the HN's L_i switches. Additionally, for the distributed version of DCA, deciders maintain state about the HN membership of their L_i neighbors in order to avoid falsely detecting conflicts; a decider may be connected to a single chooser via multiple virtual channels (L_i switches) and should not perceive identical requests across such channels as conflicts.

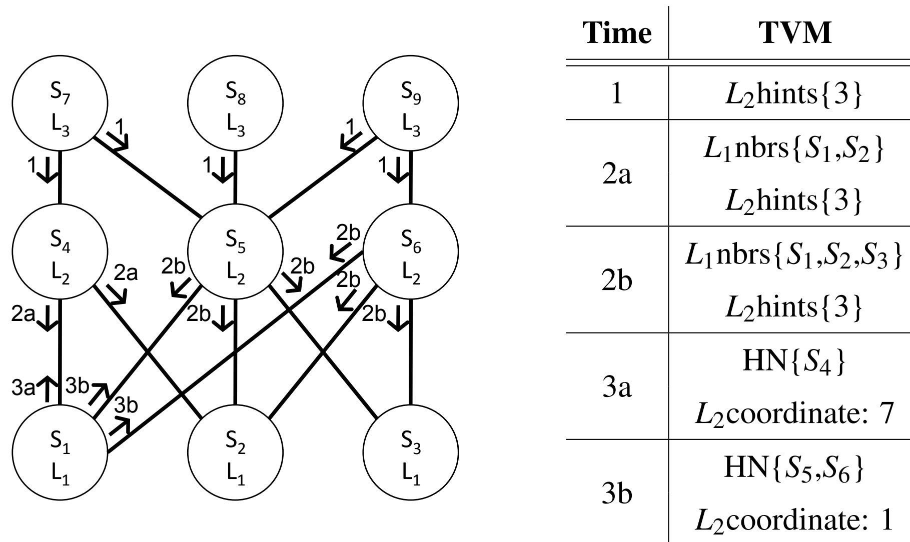
Figure 4.4c shows two distributed choosers in our example topology. Choosers $\{S_1,S_4\}$ and $\{S_1,S_5,S_6\}$ are shaded in light and dark grey, respectively. Note that S_7 is a decider for both choosers while S_8 and S_9 are deciders only for the second chooser. S_2 and S_3 play no part in L_2-coordinate selection for this topology. (The figure's numbered links will be discussed in Section 4.3.2.)

Our implementation does not separate each level's coordinate assignment into its own instance of the extended DCA protocol; rather, all information pertaining to both level and coordinate assignment is contained in a single TVM. For instance, in a 5-level topology, a TVM from an L_3 switch to an L_2 switch might contain hints for L_2 coordinates, L_3HN grouping information, and L_4 information on its way down to a representative L_1 switch. Full details of the Decider/Chooser Abstraction, a proof of correctness, and a protocol derivation for its refinements are presented in Chapter 5.

Label assignment converges when all switches at L_2 through L_{n-1} have grouped themselves into hypernodes, and all L_1 through L_{n-1} switches have selected coordinates.

Example Assignments

Figure 4.5 depicts the TVMs sent to assign coordinates to the L_2 switches in Figure 4.4's topology. For clarity, we show TVMs only for a subset of the switches. In TVM 1, all core switches disallow the selection of L_2-coordinate 3, due to its use in another HN (not shown). L_2 switches incorporate this restriction into their outgoing TVMs, including their sets of connected L_1 switches (TVMs 2a and 2b). S_1 is the representative L_1 switch for both HNs , as it has the lowest ID. S_1 selects coordinates for the HNs and informs neighboring L_2 switches of their HNs and coordinates (TVMs 3a and 3b.)



Time	TVM
1	L_2hints{3}
2a	L_1nbrs{S_1,S_2} L_2hints{3}
2b	L_1nbrs{S_1,S_2,S_3} L_2hints{3}
3a	HN{S_4} L_2coordinate: 7
3b	HN{S_5,S_6} L_2coordinate: 1

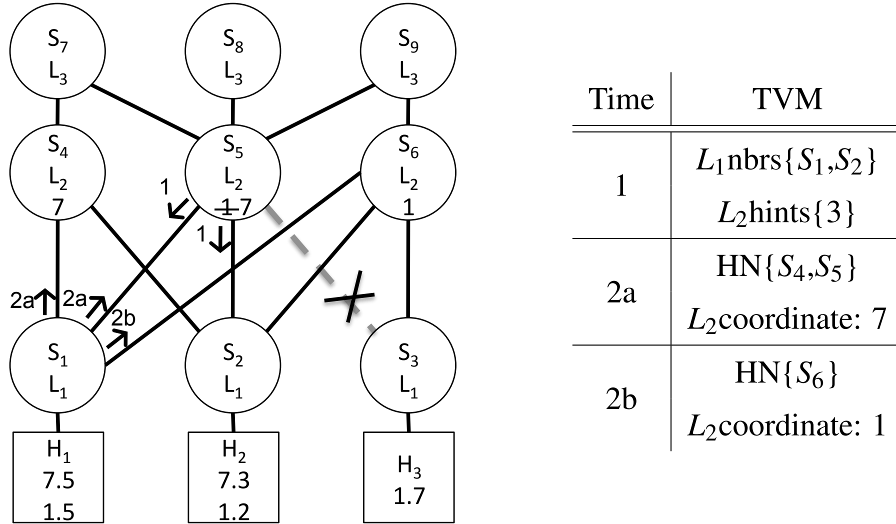
Figure 4.5: Label Assignment: L_2-Coordinates

4.2.3 Relabeling

Since ALIAS labels encode paths to hosts, topology changes may affect switch coordinates and hence host labels. For instance, when the set of L_1 switches reachable by a particular L_2 switch changes, the L_2 switch may have to select a new L_2-coordinate. This process is coined *relabeling*.

Consider the example shown in Figure 4.6 where the highlighted link between S_5 and S_3 fails. At this point, affected switches must adjust their coordinates. With TVM 1, S_5 informs its L_1 neighbors of its new connection status. Since S_1 knows the L_1

neighbors of each of its neighboring L_2 switches, it knows that it remains the representative L_1 switch for both HNs. S_1 informs S_4 and S_5 of the HN membership changes in TVM 2a, and informs S_6 of S_4's departure in TVM 2b. Since S_5 simply left one HN and joined another, existing HN, host labels are not affected.



Time	TVM
1	L_1nbrs$\{S_1,S_2\}$ L_2hints$\{3\}$
2a	HN$\{S_4,S_5\}$ L_2coordinate: 7
2b	HN$\{S_6\}$ L_2coordinate: 1

Figure 4.6: Relabeling Example

In some cases, topology fluctuations may cause host labels to change. In fact, the effects of relabeling (whether caused by link addition or deletion) are determined solely by changes to the HN membership of the upper level switch incident on the affected link. Table 4.1 shows the effects of relabeling after a change to a link between an L_2 switch s_2 and an L_1 switch s_1, in a 3-level topology. Case 1 corresponds with the example of Figure 4.6; s_2 moves from one HN to another. In this case, no labels are created nor destroyed. In case 2, one HN splits into two and all L_1 switches neighboring s_2 add a new label to their sets of labels. In case 3, two HNs merge into a single HN, and with the exception of s_1, all L_1 switches neighboring s_2 lose one of their labels. Finally, case 4 represents a situation in which an HN simply changes its coordinate, causing all neighboring L_1 switches to replace the corresponding label.

Changes due to relabeling are completely encapsulated in the forwarding information propagated by ALIAS, as described in Section 4.3.2. Additionally, in Section 4.3.3 we present an optimization that limits the effects of relabeling on ongoing sessions between pairs of hosts.

Table 4.1: Relabeling Cases

Case	Cause		Effects	
	Previous HN	New HN	s_1	Remaining L_1 switches
1	Intact	Existing	None	None
2	Intact	New	+ 1 label	+ 1 label
3	Removed	Existing	None	- 1 label
4	Removed	New	Swap label	Swap label

4.2.4 M-Graphs

There are some rare situations in which ALIAS provides connectivity between switches from the point of view of the communication component, but not from that of coordinate assignment. The presence of an *M-graph* in a topology can lead to this problem, as can the use of peer links. We consider M-graphs below and discuss peer links in Section 4.3.4.

ALIAS relies on shared core switch parents to enforce the restriction that pairs of L_{n-1}HNs do not select identical coordinates. There are topologies, though, in which two L_{n-1}HNs do not share a core and could therefore select identical coordinates. Such an M-graph is shown in Figure 4.7. In the example, there are 3 L_2HNs, $\{S_4\}$, $\{S_5,S_6\}$ and $\{S_7\}$. It is possible that S_4 and S_7 select the same L_2-coordinate, e.g., 3, as they do not share a neighboring core. Since HN $\{S_5,S_6\}$ shares a parent with each of the other HNs, its coordinate is unique from those of $\{S_4\}$ and $\{S_7\}$. L_1 switches S_1 and S_3 are free to choose the same L_1-coordinates, 1 in this example. As a result, two hosts H_1 and H_3 are legally assigned identical ALIAS labels, `(3.1.4.0)`, if both H_1 and H_3 are connected to their L_1 switches on the same numbered port (in this case, 4), and have `VM` coordinate 0.

H_2 can now see two non unique ALIAS labels, which introduces a routing ambiguity. If H_2 attempts to forward a packet to H_1, it will use the label `(3.1.4.0)`. When S_2 receives the packet, S_2 can send this packet either to S_5 or S_6, since it thinks it is connected to an L_2HN with coordinate 3 via both. The packet could be transmitted to

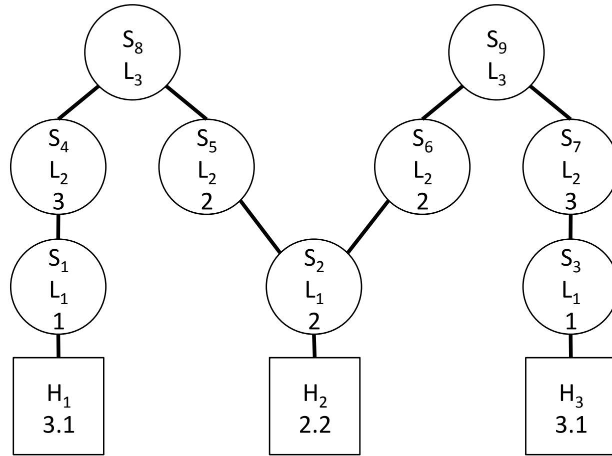


Figure 4.7: Example M-Graph

the unintended destination H_3 via S_6, S_9, S_7, S_3. When the packet reaches S_3, S_3 is in a position to verify whether the packet's IP address matches H_3's ALIAS label, by referencing a flow table entry that holds IP address-to-ALIAS label mappings. (Note that such flow table entries are already present for the communication component, as shown in Section 4.3.3.) A packet destined to H_1's IP address would not match such a flow entry and would be punted to switch software.[2]

Because we expect M-graphs to occur infrequently in well-connected data center environments, our implementation favors a simple "detect and resolve" technique. In our example, S_3 receives the mis-routed packet and knows that it is part of an M-graph. At this point S_3 sends a directive to S_7 to choose a new L_2-coordinate. This will result in different ALIAS labels for H_1 and H_3. Once the relabeling decision propagates via routing updates, S_2 correctly routes H_1's packets via S_5. The convergence time of this relabeling equals the convergence period for our routing protocol, or 3 TVM periods.[3]

In our simulations we encounter M-graphs only for input topologies with extremely poor connectivity, or when we artificially reduce the size of the coordinate domain to cause collisions. If M-graphs are not tolerable for a particular network, they can be prevented in two ways, each with an additional application of the DCA abstraction.

[2]If the L_1 switches' coordinates did not overlap, detection would occur at S_7.

[3]It is possible that two HNs involved in an M-graph simultaneously detect and recover from a collision, causing an extra relabeling. However, we optimize for the common case, as this potential cost is small and unlikely to occur.

For the first method, the set of deciders for a pair of HNs is augmented to include not only shared parents but also lower-level switches that can reach both HNs. For example, in Figure 4.7, S_2 would be a decider for (and would ensure L_2-coordinate uniqueness among) all three L_2HNs. A second method for preventing M-graphs is to assign coordinates to core switches. In this case, core switches group themselves into hypernodes and select shared coordinates, using representative L_1 switches to facilitate cooperation. Lower level switches act as deciders for these core-HNs. Both of these M-graph prevention techniques increase convergence time, as there may be up to n hops between a core-HN and its deciders in an n-level hierarchy. Given this cost and because of the low probability of M-graphs in practice, our implementation uses the detect-and-resolve solution.

4.3 Communication

Here, we present an example of one of the many communication components that could operate over ALIAS labels.

4.3.1 Routing

An ALIAS label specifies a 'downward' path from a core to the identified host. Each core switch is able to reach all hosts with a label that begins with the coordinate of any L_{n-1}HN directly connected to it. Similarly, each switch in an L_iHN can reach any host with a label that contains one of the HN's coordinates in the ith position. Thus, routing packets downward is simply based on an L_i switch matching the destination label's $(i-1)^{th}$ coordinate to that of one or more of its L_{i-1} neighbors.

To leverage this simple downward routing, ingress switches must be able to move data packets to cores capable of reaching a destination. This reduces to a matter of sending a data packet towards a core that reaches the L_{n-1}HN corresponding to the first coordinate in the destination label. L_{n-1} switches learn which cores reach other L_{n-1}HNs directly from neighboring cores and pass this information downward via TVMs. Switches at level L_i in turn learn about the set of L_{n-1}HNs reachable via each neighboring L_{i+1} switch and inform L_{i-1} neighboring switches.

4.3.2 Forwarding

Switch forwarding entries map a packet's input port and coordinates to the appropriate output port. The coordinate fields in a forwarding entry can hold a number, requiring an exact match, or a 'don't care' (DC) that matches all values for that coordinate. An L_i switch forwards a packet with a destination label matching any of its own label prefixes downward to the appropriate L_{i-1}HN. If none of its prefixes match, it uses the label's L_{n-1}-coordinate to send the packet towards a core that reaches the packet's destination.

Figure 4.8 presents a subset of the forwarding tables entries of switches S_7, S_4 and S_1 of Figure 4.4c, assuming the L_1-coordinate assignments of Figure 4.4b and that S_1 has a single host on port 3. Entries for exception cases are omitted for clarity.

Core (S_7)

InPort	L_2	L_1	H	OutPort
0	1	DC	DC	1
1	7	DC	DC	0

Level L_2 (S_4)

InPort	L_2	L_1	H	OutPort
1/2	7	5	DC	0
0/2	7	3	DC	1
0/1	1	DC	DC	2

Level L_1 (S_1)

InPort	L_2	L_1	H	OutPort
0	7	5	3	3
1/2	1	5	3	3
3	7	DC	DC	0
3	1	DC	DC	1/2

Figure 4.8: Example of Forwarding Table Entries

All forwarding entries are directional, in that a packet can be headed 'downwards' to a lower level switch or 'upwards' to a higher level switch. Directionality is determined by the packet's input port. ALIAS restricts the direction of packet forwarding to ensure loop-free forwarding. The key restriction is that a packet coming into a switch from a higher level switch can only be forwarded downwards, and that

a packet moving laterally cannot be forwarded upwards. We refer to this property as up*/across*/down* forwarding, an extension of the up*/down* forwarding introduced in Autonet [65].

4.3.3 End-to-End Communication

Recall that ALIAS labels can serve as a basis for a variety of communication techniques. Here we present an implementation based on MAC address rewriting.

When two hosts wish to communicate, the first step is generally ARP resolution to map a destination host's IP address to a MAC address. In ALIAS, we instead resolve IP addresses to ALIAS labels. This ALIAS label is then written into the destination Ethernet address. All switch forwarding proceeds based on this destination label. Unlike standard Layer 2 forwarding, the destination MAC address is not rewritten hop-by-hop through the network.

Figure 4.9 depicts the flow of information used to establish end-to-end communication between two hosts. When an L_1 switch discovers a connected host h, it assigns to h a set of ALIAS labels. L_1 switches maintain a mapping between the IP address, MAC address and ALIAS labels of each connected host. Additionally, they send a mapping of IP address-to-ALIAS labels of connected hosts upwards to all reachable cores. This eliminates the need for a broadcast-based ARP mechanism. Arrows 1-3 in Figure 4.9 show this mapping as it moves from L_1 to the cores.

To support unmodified hosts, L_1 switches intercept ARP queries (arrow 4) and reply if possible (arrow 9). Otherwise, they send a proxy ARP query to all cores above them in the hierarchy via intermediate switches (arrows 5,6). Cores with the requested mappings reply (arrows 7,8). The querying L_1 switch then replies to the host with an ALIAS label (arrow 9) and incorporates the new information into its local map, taking care to ensure proper handling of responses from multiple cores. As a result, the host will use this label as the address in the packet's Ethernet header. Prior to delivering a data packet, the egress switch rewrites the ALIAS label with the actual MAC address of the destination host, using locally available state information encoded in the hardware forwarding table.

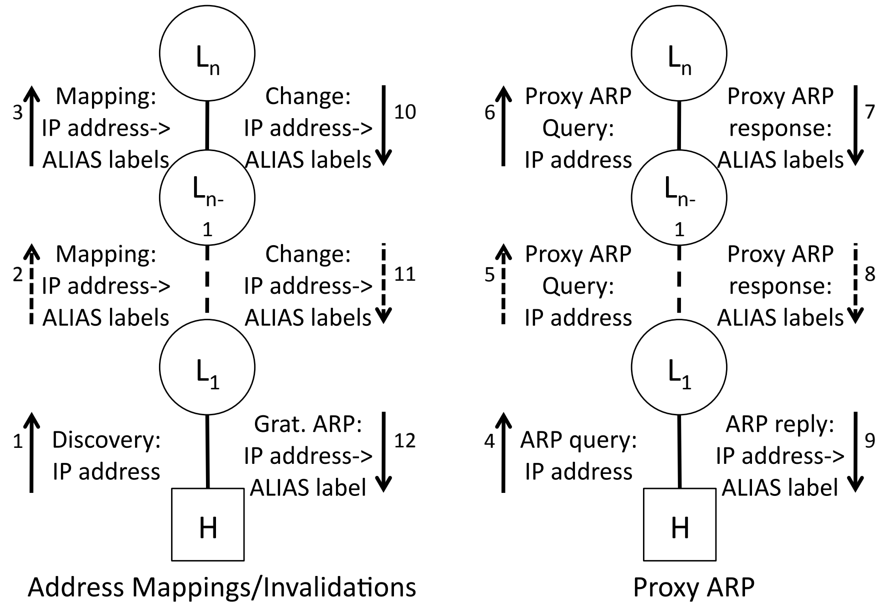


Figure 4.9: End-to-End Communication

Hosts can have multiple ALIAS labels corresponding to multiple sets of paths from cores. However, during ARP resolution, a host expects only one MAC address to be associated with a particular IP address. To address this, the querying host's neighboring L_1 switch chooses one of the ALIAS labels of the destination host. This choice could be made in a number of ways; switches could select randomly or could base their decisions on local views of dynamically changing congestion. In our implementation, we include a measure of each label's value when passing labels from L_1 switches to cores. We base a host label's value on connectivity between the host h and the core of the network as well as on the number of other hosts that can reach h using this label. An L_1 switch uses these combined values to select a label out of the set returned by a core.

Link additions and failures can result in relabeling. While the routing protocol adapts to changes, existing flows to previously valid ALIAS labels will be affected due to ARP caching in unmodified end hosts. Here, we describe our approach to minimize disruption to existing flows in the face of shifts in the topology. We note however that any network environment is subject to some period of convergence following a failure. Our goal is to ensure that ALIAS convergence time at least matches the behavior of currently deployed networks.

Upon a link addition or failure, ALIAS performs appropriate relabeling of switches and hosts (Section 4.2.3) and propagates the new topology view to all switches as part of standard TVM exchanges. Recall that cores store a mapping of IP addresses-to-ALIAS labels for hosts. Cores compare received mappings to existing state to determine newly invalid mappings. Cores also maintain a cache of recently queried ARP mappings. Using this cache, core switches inform recent L_1 requesters that an ARP mapping has changed (arrows 10-11), and L_1 switches in turn send gratuitous ARP replies to hosts (arrow 12).

Additionally, ingress L_1 switches can preemptively rewrite stale ALIAS labels to maintain connectivity between pairs of hosts during the window of vulnerability when a gratuitous ARP has been sent but not yet received. In the worst case, failure of certain cores may necessitate an ARP cache timeout at hosts before communication can resume.

Recall that ALIAS enables two classes of multi-path support. The first class is tied to the selection of a particular label (and thus a corresponding set of paths) from a host's label set, whereas the second represents a choice within this set of paths. For this second class of multi-path, ALIAS supports standard multi-path forwarding techniques such as ECMP [35]. Essentially, forwarding entries on the upward path can contain multiple next hops toward the potentially multiple core switches capable of reaching the appropriate top-level coordinate in the destination host label.

4.3.4 Peer Links

ALIAS considers *peer links* between switches at the same level of the hierarchy as special cases for forwarding. There are two considerations to keep in mind when introducing peer links into ALIAS: maintaining loop-free forwarding guarantees and retaining ALIAS's scalability properties. We consider each in turn below.

To motivate our method for accommodating peer links, we first consider the reasons for which a peer link might exist in a network. A peer link might be added

1. to create direct connectivity between two otherwise disconnected HNs or cores,
2. to create a "shortcut" between two HNs (e.g., HNs with frequent interaction) or
3. unintentionally.

ALIAS supports *intentional* peer links with up*/across*/down* forwarding. In other words, a packet may travel upwards and then "jump" directly from one HN to another, or traverse a set of cores, before moving downwards to its destination.

Switches advertise hosts reachable via peer links in their outgoing TVMs. While the up* and down* components of the forwarding path are limited in length by the overall depth of the hierarchy, the across* component can be arbitrarily long. To avoid the introduction of forwarding loops, peer link advertisements include a hop count.

The number of peer link traversals allowed during the across* portion of forwarding represents a tradeoff between routing flexibility and ALIAS convergence time. This is due to the fact that links used for communication must also be considered for coordinate assignment, as explored in Section 4.2.4 for M-graphs. Consider the example of Figure 4.10. In the figure, dotted lines indicate long chains of links, perhaps involving switches not shown. Since host H_k can reach both other hosts, H_i and H_j, switches S_i and S_j need to have unique coordinates. However, they do not share a common parent, and therefore, must cooperate across the long chain of peer links between them to ensure coordinate uniqueness. In fact, if a packet is allowed to cross p peer links during the across* segment of its path, switches as far as $2p$ peer links apart must not share coordinates. This increases convergence time for large values of p. Because of this ALIAS allows a network designer to tune the number of peer links allowed per across* segment to limit convergence time while still providing the necessary routing flexibility. Since core switches do not have coordinates, this restriction on the length of the across* component is not necessary at the core level; cores use a standard hop count to avoid forwarding loops.

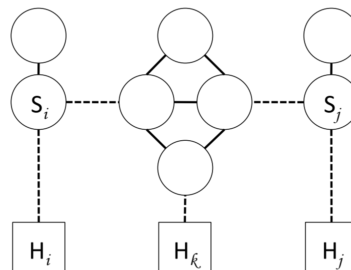


Figure 4.10: Peer Link Tradeoff

It is important that peer links are used judiciously, given the particular style of forwarding chosen. For instance, supporting shortest path forwarding may require disabling "shortcut"style peer links when they represent a small percentage of the connections between two HNs. This is to avoid a situation in which all traffic is directed across a peer link (as it provides the shortest path) and the link is overwhelmed.

4.3.5 Switch Modifications

We engineer ALIAS labels to be encoded into 48 bits to be compatible with existing destination MAC addresses in protocol headers. Our task of assigning globally unique hierarchical labels would be simplified if there were no possibility of collisions in coordinates, for instance if we allowed each coordinate to be 48-bits in length. If we adopted longer ALIAS labels, we would require modified switch hardware that would support an encapsulation header containing the forwarding address. Forwarding tables would need to support matching on pre-selected and variable numbers of bits in encapsulation headers. Many commercial switches already include such functionality in support of emerging Layer 2 protocols such as TRILL [69] and SEATTLE [42].

Our goal of operating with unmodified hosts does require some support from network switching elements. ALIAS L_1 switches intercept all ARP packets from hosts. This does not require any hardware modifications, since packets that do not match a flow table entry can always be sent to the switch software and ARP packets need not necessarily be processed at line rate. We further introduce IP address-to-ALIAS label mappings at cores, and IP address, actual MAC address and ALIAS label mappings at L_1. We also maintain a cache of recent ARP queries at cores. All such functionality can be realized in switch software without hardware modifications.

4.4 Implementation

ALIAS switches maintain the state necessary for level and coordinate assignment as well as local forwarding tables. Switches react to two types of events: timer firings and message receipt. When a switch receives a TVM it updates the necessary local state and forwarding table entries. The next time its `TVMsend` timer fires, it compiles

a TVM for each switch neighbor as well as a `ping` for all host neighbors. Neighbors of unknown types receive both. Outgoing TVMs include all information related to level and coordinate assignment, and forwarding state, and may include label mappings as they are passed upwards towards cores. The particular TVM created for any neighbor varies both with levels of the sender and the receiver as well as with the identity of the receiver. For instance, in a 3-level topology, an L_2 switch sends the set of its neighboring L_1 switches downward for HN grouping by the representative L_1 switch. On the other hand, it need not send this information to cores.

Figure 4.11 shows the basic architecture of ALIAS. We have produced two different implementations of ALIAS, which we describe below.

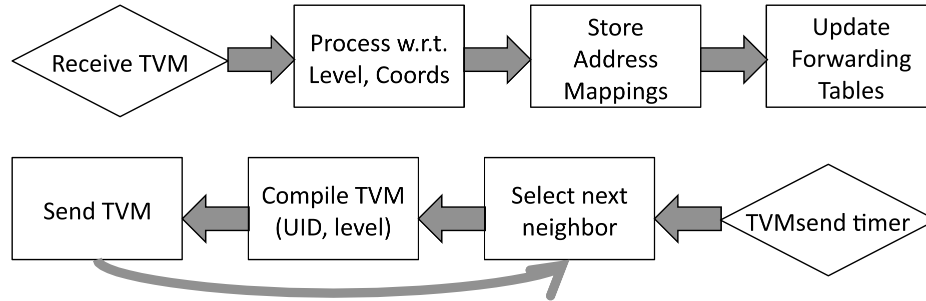


Figure 4.11: ALIAS Architecture

4.4.1 Mace Implementation

We first implemented ALIAS in Mace [21, 40]. Mace is a language for distributed system development that we chose for two reasons; the Mace toolkit includes a model checker [39] that can be used to verify correctness, and Mace code compiles into standard C++ code for deployment of the exact code that was model checked.

We verified the correctness of ALIAS by model checking our Mace implementation. This included all protocols discussed in this chapter: level assignment, coordinate and label assignment, routing and forwarding, and proxy ARP support with invalidations on relabeling. For a range of topologies with intermittent switch, host, and network failures, we verified (via liveness properties) the convergence of level and coordinate assignment and routing state as well as the correct operation of label resolution and invalidation. Further, we verified that all pairs of hosts that are connected by the physical

topology are eventually able to communicate infinitely often (though connectivity may temporarily be lost due to switch or link failure).

4.4.2 NetFPGA Testbed Implementation

Using our Mace code as a specification, we integrated ALIAS into an OpenFlow [1] testbed, consisting of 20 4-port NetFPGA PCI-card switches [51] hosted in 1U dual-core 3.2 GHz Intel Xeon machines with 3GB of RAM. 16 end hosts connect to the 20 4-port switches wired as a 3-level fat tree. All machines run Linux 2.6.18-92.1.18.el5 and switches run OpenFlow v0.8.9r2.

Although OpenFlow is based around a centralized controller model, we wished to remain completely decentralized. To accomplish this, we implemented ALIAS directly in the OpenFlow switch, relying only on OpenFlow's ability to insert new forwarding rules into a switch's tables. We also modified the OpenFlow configuration to use a separate controller per switch. These modifications to the OpenFlow software consist of approximately 1,200 lines of C code.

4.5 Evaluation

We set out to answer the following questions with our experimental evaluation of ALIAS:

- How scalable is ALIAS in terms of storage requirements and control overhead?
- How effective are hypernodes in compacting forwarding tables?
- How quickly does ALIAS converge on startup and after faults? How many switches relabel after a topology change and how quickly does the new information propagate?

Our experiments run on our NetFPGA testbed, which we augment with miswirings and peer links as necessary. For measurements on topologies larger than our testbed, we rely on simulations.

4.5.1 Storage Requirements

We first consider the storage requirements of ALIAS. This includes all state used to compute switches' levels, coordinates and forwarding tables. For a given number of hosts, we determined the number of L_1, L_2 and L_3 switches present in a 3-level, 128-port fat tree-based topology. We then calculated analytically the storage overhead required at each type of switch as a function of the input topology size, as shown in Figure 4.12. L_1 switches store the most state, as they may be representative switches for higher level HNs, and therefore must store state on behalf of these HNs.

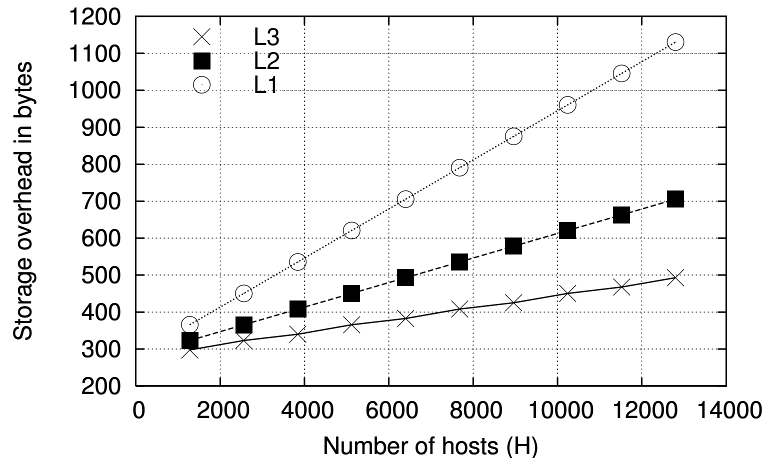


Figure 4.12: Storage Overhead for 3-Level, 128-Port Tree

We also empirically measured the storage requirements of ALIAS on our testbed. L_1, L_2 and L_3 switches require 122, 52 and 22 bytes of storage, respectively, for our 16-switch topology; these results would grow linearly with the number of hosts. Overall, the total required state is well within the range of what is available in commodity switches today. Note that this state need not be accessed on the data path; it can reside in DRAM accessed by the local embedded processor.

4.5.2 Control Overhead

We next consider the control message overhead of ALIAS. Table 4.2 shows the contents of TVMs, both for immediate neighbors and for communication with representative L_1 switches. The table gives the expected size of each field, (where S and C are

the sizes of a switchID and coordinate), as well as the measured sizes for our testbed implementation (where $S = 48$, $C = 8$ bits). Since our testbed has 3 levels, TVMs from L_2 switches to their L_1 neighbors are combined with those to representative L_1 switches (and likewise for upward TVMs); our measured results reflect these combinations. Messages sent downwards to L_1 switches come from all members of an L_iHN and contain per-parent information for each HN member; therefore, these messages are the largest.

Table 4.2: Level/Coordinate Assignment Overhead

<table>
<tr><th>Sender and Receiver</th><th>Field</th><th>Expected Size</th><th>Measured Size</th></tr>
<tr><td>All-to-All</td><td>level</td><td>$log(n)$</td><td>2 bits</td></tr>
<tr><td rowspan="2">To Downward Neighbor</td><td>hints</td><td>$kC/2$</td><td rowspan="2">L_3 to L_2: 6B</td></tr>
<tr><td>dwnwrd_HNs</td><td>$kS/2$</td></tr>
<tr><td rowspan="2">To rep. L_1 switch</td><td>per-parent hints</td><td>$k^2C/4$</td><td rowspan="2">L_2 to L_1: 28B</td></tr>
<tr><td>per-parent dwnwrd_HNs</td><td>$k^2S/4$</td></tr>
<tr><td rowspan="3">To Upward Neighbor</td><td>coord</td><td>C</td><td rowspan="3">L_2 to L_3: 5B</td></tr>
<tr><td>HN</td><td>$kS/2$</td></tr>
<tr><td>rep. L_1</td><td>S</td></tr>
<tr><td rowspan="2">From Rep. L_1 switch</td><td>per-parent coords</td><td>$kC/2$</td><td rowspan="2">L_1 to L_2: 7B</td></tr>
<tr><td>HN assignment</td><td>$kS/2$</td></tr>
</table>

The TVM period must be at least as large as the time it takes a switch to process k incoming TVMs, one per port. On our NetFPGA testbed, the worst case processing time for a set of TVMs was $57\mu s$ plus an additional $291\mu s$ for updating forwarding table entries in OpenFlow in a small configuration. Given this, $100ms$ is a reasonable setting for a TVM cycle at scale. L_1 switches send $\frac{k}{2}$ TVMs per cycle while all other switches send k TVMs. The largest TVM is dominated by $\frac{k^2S}{4}$, giving a control overhead of $\frac{k^3S}{400}\frac{b}{ms}$. For a network with 64-port switches, this is $31.5Mbps$ or 0.3% of a $10Gbps$ link, an acceptable cost for a routing/location protocol that scales to hundreds of thousands hosts at 4 levels. This brings out a tradeoff between convergence time and control overhead; a smaller TVM cycle time is certainly possible, but would correspond to a larger amount of control data sent per second. It is also important to note that this control overhead is a function only of k and TVM cycle time; it does not increase with link speed.

4.5.3 Compact Forwarding Tables

Next, we asses the effectiveness of hypernodes in compacting forwarding tables. We use our simulator to generate fully provisioned fat tree topologies with k-port switches. We then remove a percentage of the links at each level of the hierarchy to model less than fully-connected networks. We use the smallest possible coordinate domain that can accommodate the worst-case number of HNs for each topology, and allow data packets to cross as many peer links as needed, within the constraints of up*/across*/down* forwarding.

Once the input topology has been generated, we use our simulator to calculate all switches' levels and HNs, and we select random coordinates for switches based on common upper-level neighbors. Finally, we populate forwarding tables with the labels corresponding to the selected coordinates and analyze the forwarding table sizes of switches.

Table 4.3 gives the parameters used to create each input topology along with the total number of servers supported and the average number of number of forwarding table entries per switch. The table lists values for optimized forwarding tables (in which redundant entries are removed and entries for peer links appear only when providing otherwise unavailable connectivity) and unoptimized tables (that include redundant entries for use with techniques such as ECMP). As the tables shows, even in graphs supporting millions of servers, the number of forwarding entries is dramatically reduced from the entry-per-host requirement of Layer 2 techniques.

As the provisioning of the tree reduces, the number of forwarding entries initially increases. This corresponds to cases in which the tree has become somewhat fragmented from its initial fat tree specification, leading to more HNs and thus more coordinates across the graph. However, as even more links are deleted, forwarding table sizes begin to decrease; for extremely fragmented trees, mutual connectivity between pairs of switches drops, and a switch need not store forwarding entries for unreachable destinations.

Table 4.3: Forwarding Entries Per Switch

Topology Info				Forwarding Entries	
Levels	Ports	% Fully Provisioned	Total Servers	Optimized	Unoptimized
3	16	100	1024	22	112
		80		62	96
		50		48	58
		20		28	31
	32	100	8,192	45	429
		80		262	386
		50		173	217
		20		86	95
	64	100	65,536	90	1677
		80		1028	1530
		50		653	842
		20		291	320
4	16	100	8,192	23	119
		80		197	246
		50		273	307
		20		280	304
	32	100	131,072	46	457
		80		1278	1499
		50		2079	2248
		20		2415	2552
5	16	100	65,536	23	123
		80		492	550
		50		886	931
		20		1108	1147

4.5.4 Convergence Time

We measured ALIAS's convergence time on our testbed for both an initial startup period as well as across transient failures. We consider a switch to have converged when it has stabilized all applicable coordinates and HN membership information.

As shown in Figure 4.13, ALIAS takes a maximum of 10 TVM cycles to converge when all switches and hosts are initially booted, even though they are not booted simultaneously. L_3 switches converge most quickly since they simply facilitate L_2-coordinate uniqueness. L_1 switches converge more slowly; the last L_1 switch to converge might see the following chain of events: (1) L_2 switch s_{2a} sends its coordinate to L_3 switch s_3, (2) s_3 passes a hint about this coordinate to L_2 switch s_{2b} that (3) forwards the hint to its representative L_1 switch, which replies (4) with an assignment for s_{2b}'s coordinate.

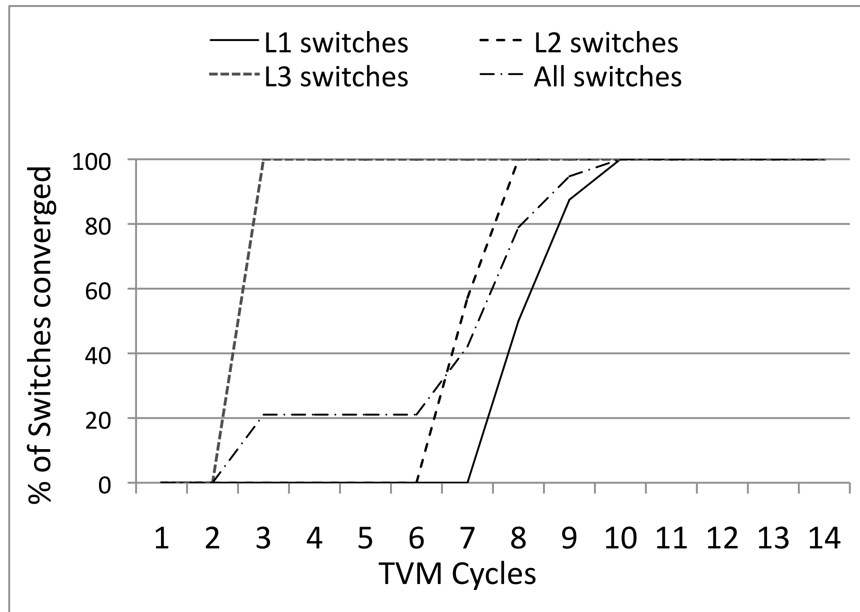


Figure 4.13: CDF of Startup Convergence Times

These 4 TVM cycles combine with 5 cycles to propagate level information up and down the 3-level hierarchy, for a total of 9 cycles. The small variation in our results is due to our asynchronous deployment setting.

In our implementation, a TVM cycle is 400μs, leading to an initial convergence time of 4ms for our small topology. Our cycle time accounts for 57μs for TVM pro-

cessing and 291μs for flow table updates in OpenFlow. In general, the TVM period may be set to anything larger than the time required for a switch to process one incoming TVM per port. In practice we would expect significantly longer cycle times in order to minimize control overhead.

We also considered the behavior of ALIAS in response to failures. As discussed in Section 4.2.3, relabeling is triggered by additions or deletions of links, and its effects depend on the HN membership of the upper level switch on the affected link. Figure 4.14 shows an example of each of the cases from Table 4.1 along with measured convergence time on our testbed. The examples in the figure are for link addition; we verified the parallel cases for link deletion by reversing the experiments. We measured the time for all HN membership and coordinate information to stabilize at each affected switch. Our results confirm the locality of relabeling effects; only immediate neighbors of the affected L_2 switch react, and few require more than the 2 TVM cycles used to recompute HN membership.

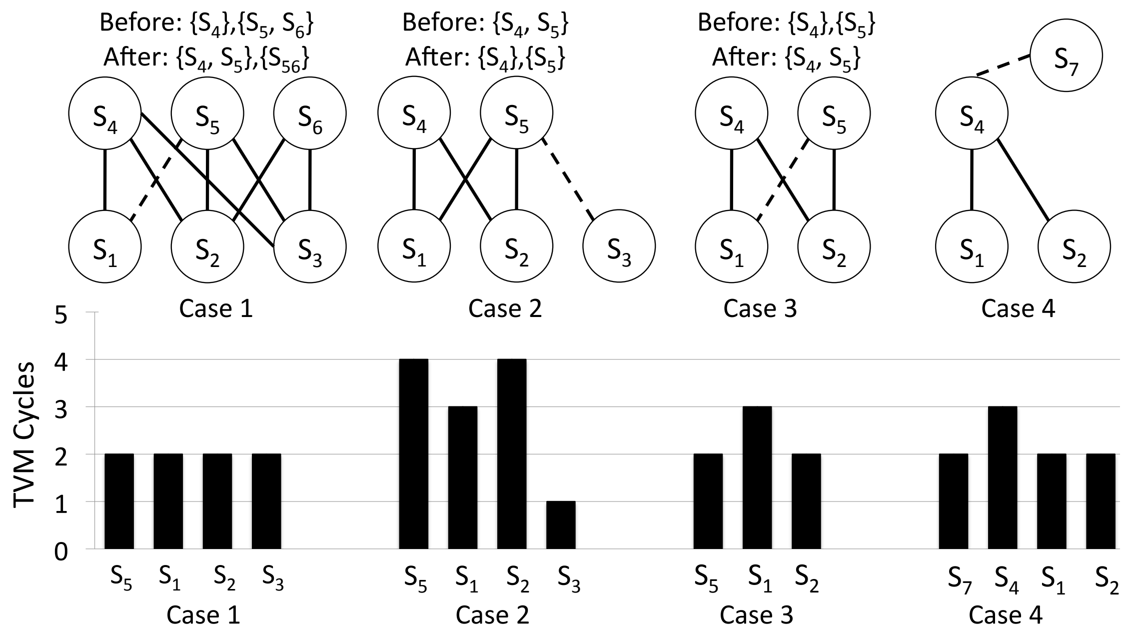


Figure 4.14: Relabeling Convergence Times

(Dashed lines are new links.)

4.6 Related Work

ALIAS provides automatic, decentralized, scalable assignment of hierarchical host labels. To the best of our knowledge, this is the first system to address all three of our goals simultaneously.

Our work can trace its lineage back to the original work on spanning trees [58] designed to bridge multiple physical Layer 2 networks. While clearly ground-breaking, spanning trees suffer from scalability challenges and do not support hierarchical labeling. SmartBridge [61] provides shortest path routing among Layer 2 hosts but is still broadcast-based and does not support hierarchical host labels. More recently, Rbridges [59] and TRILL [69] suggest running a full-blown routing protocol among Layer 2 switches along with an additional Layer 2 header to protect against forwarding loops.

SEATTLE [42] improves upon aspects of Rbridge's scalability by distributing the knowledge of host-to-egress switch mapping among a distributed directory service implemented as a one-hop DHT. In general, however, all of these earlier protocols target arbitrary topologies with broadcast-based routing and flat host labels. ALIAS benefits from the underlying assumption that we target hierarchical topologies.

VL2 [28] proposed scaling Layer 2 to mega data centers using end-host modification, and addressed load balancing to improve agility in data centers. However VL2 uses an underlying IP network fabric, which requires subnet and DHCP server configuration, and does not address the requirement for automation.

Most related to ALIAS are PortLand [56] and DAC [16]. PortLand employs a Location Discovery Protocol for host numbering but differs from ALIAS in that it relies on a central fabric manager, assumes a 3-level fat tree topology, and does not support arbitrary miswirings and failures. Also, LDP makes decisions (e.g. edge switch labeling and pod groupings) based on particular interconnection patterns in fat trees. This limits the approach under heterogeneous conditions (e.g. a network fabric that is not yet fully deployed) and during transitory periods (e.g., when the system first boots). Contrastingly, ALIAS makes decisions solely based on current network conditions. DAC supports arbitrary topologies but is fully centralized and requires that an administrator manually input configuration information both initially and prior to planned changes.

Landmark [70] also automatically configures hierarchy onto a physical topology and relabels as a result of topology changes for ad hoc wireless networks. However, Landmark's hierarchy levels are defined such that even small topology changes (e.g. a router losing a single neighbor) trigger relabeling. Also, routers maintain forwarding state for distant nodes while ALIAS aggregates such state with hypernodes.

4.7 Summary

In this chapter, we present ALIAS, a protocol that provides scalable, automatic and decentralized label assignment in the data center. This addresses the difficulties associated with labeling protocols that rely on centralized coordination, error-prone manual configuration or excessively large forwarding state. We then offer a communication protocol that efficiently leverages the labels assigned by ALIAS. We show the correctness of our labeling and communication protocols through model checking and we evaluate ALIAS via a realistic deployment on our netFPGA testbed. Our evaluation shows that ALIAS operates with low message overhead and quick convergence time, and that ALIAS switches have significantly less forwarding state than that of other decentralized and automatic addressing protocols.

4.8 Acknowledgment

Chapter 4, in part, contains material as it appears in the Proceedings of the ACM Symposium on Cloud Computing (SOCC) 2011. Walraed-Sullivan, Meg; Niranjan Mysore, Radhika; Tewari, Malveeka; Zhang, Ying; Marzullo, Keith; Vahdat, Amin. The dissertation author was the primary investigator and author of this paper.

Chapter 5

A Randomized Algorithm for Label Assignment in Dynamic Networks

In this chapter, we consider the formalization of ALIAS, so as to reason more carefully about the protocol's correctness and performance. We specify the problem solved by coordinate assignment in ALIAS as a a more general class of problems, that of label assignment to network elements.

The assignment of labels to network elements is a well-understood problem. Often, labels can be assigned statically, as with MAC addresses in traditional Layer 2 networks, or by a central authority as in DHCP in Layer 3 networks. When a dynamic, decentralized solution is required, one can employ a Consensus-based state machine approach [63]. However, dynamic assignment becomes more complex when the rules for labels depend on connectivity and when connectivity (and, hence, the labels) can change over time. As we will show in Section 5.2.1, using a state machine approach becomes difficult in this case.

As we came to this problem while designing ALIAS (Chapter 4), we have a number of related requirements. Practical constraints are important. We require a decentralized solution because a centralized approach has its own challenges, such as exhibiting a single point of failure. Additionally, at the scale of the data center, establishing communication between a centralized component and all network elements necessitates either flooding or a separate out-of-band control network, an undesirable requirement. As well as being decentralized, our solution needs to scale to hundreds of thousands

of nodes, and to be robust in the face of miswirings. It needs to have a low message overhead and convergence time, to be robust under transient startup conditions, and to retain high availability and quick stabilization after failures. Finally, a simple solution is ideal, since it is important that it can be designed and implemented correctly. This chapter describes a simple randomized approach that meets our practical goals.

We formally specify the problem of label assignment in and provide a new algorithm, the Decider/Chooser Protocol (DCP), as a solution to this problem. We then discuss the correctness and performance of DCP and provide a probabilistic analysis of its convergence time. Next, we extend DCP to solve the issue of automatic labeling in data center networks and offer another application of DCP, handoff in wireless networks. Finally, we provide a full derivation of label assignment in ALIAS from the basic DCP protocol.

5.1 ALIAS Details

In this section, we present a brief review of ALIAS in order to help the reader to understand the concepts to follow.

In ALIAS, switches are organized into a multi-rooted tree, with end hosts connected to leaf switches, as shown in Figure 5.1. The ALIAS protocol includes three components: **Level Assignment**, **Label Assignment** and **Communication**. First, switches run a distributed protocol to determine their levels, L_1 through L_n, within the tree. They then select *labels* that will form the basis for communication. To select labels, switches first choose *coordinates*, which are values from a given domain. These coordinates are then concatenated along paths from the roots of the tree to switches in order to form switch labels. There may be multiple paths from the top level of the tree to any given switch, so switches in ALIAS can have multiple labels.[1] A host label is formed by concatenating a host h's neighboring L_1 switch s_1's labels to the number of the port on which h connects to s_1. Finally, once labels have been established, switches communicate with other switches and hosts using these labels as a basis for the ALIAS routing and forwarding protocols.

[1] In Section 5.6, we show how ALIAS reduces the number of labels per host.

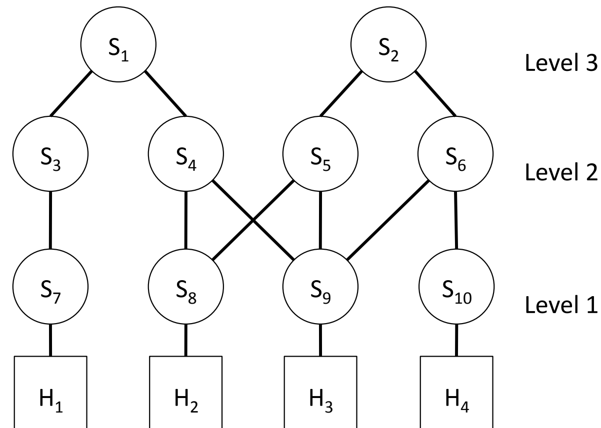


Figure 5.1: ALIAS Topology

In this chapter, we consider the problem of assigning coordinates to switches in ALIAS. In Section 5.2, we describe the requirements of coordinates and labels in order for ALIAS communication to function properly. We specify the Label Selection Problem and show how coordinate selection in ALIAS maps to this problem.

5.2 The Label Selection Problem

In the Label Selection Problem (LSP), we consider topologies made up of *chooser* processes connected to *decider* processes, as shown in Figure 5.2. These chooser and decider processes correspond to nodes at adjacent levels of a multi-rooted tree in ALIAS. All processes have globally unique identifiers, such as MAC addresses, chosen from a large address space. Desired is an assignment of labels from a small label space to choosers such that any two choosers that are connected to the same decider have distinct labels; this is the key requirement that allows ALIAS communication to operate over assigned labels.

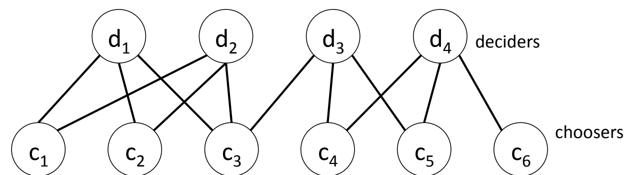


Figure 5.2: Label Selection Problem Topology

More formally, each chooser c has a set $c.deciders$ of deciders associated with it. We denote c's current choice of label with $c.me$, and $c.me = \perp$ indicates that c has not chosen a label.

A chooser c is connected to each decider in $c.deciders$ with a fair lossy link. Such links can drop messages, but if two processes p and q are connected by a fair lossy link and p sends m infinitely often to q, then q will receive m infinitely often.

Both decider and chooser processes can *crash* in a failstop manner (thus going from *up* to *down*) and can *recover* (thus going from *down* to *up*) at any time. We assume that a process writes its state to stable storage before sending a set of messages. When a process recovers, it is restored to the state that it was in before sending the last set of messages: duplicate messages may be sent upon recovery. So, we treat recovered processes as perhaps slow processes, and assume that duplicate messages can occur.

Figure 5.3 illustrates sets of choosers and the deciders they share, based on the topology shown in Figure 5.2. For instance, chooser c_3 shares deciders d_1 and d_2 with choosers c_1 and c_2 and shares decider d_3 with choosers c_4 and c_5. Because of this, c_3 may not select the same label as any of choosers c_1, c_2, c_4 and c_5. However, c_3 and c_6 are free to select the same label. In fact, the highlighted sub-graphs in Figure 5.3 correspond to the maximal bipartite graphs embedded in the topology.

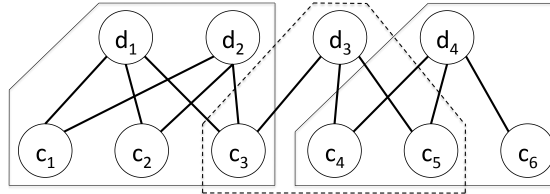


Figure 5.3: Choosers and Shared Deciders

We more formally specify LSP with the following two properties:

Progress: For each chooser c, once c remains up, eventually $c.me \neq \perp$.

Distinctness: For each distinct pair of choosers c_1 and c_2, once c_1 and c_2 remain up and there is some decider that remains up and remains in $c_1.deciders \cap c_2.deciders$, eventually always $c_1.me \neq c_2.me$.

As specified, a chooser does not know when its choice satisfies **Distinctness**. Indeed, it is impossible for a chooser to know this without further constraining the problem. Consider the example in Figure 5.4, where nodes c_1 through c_3 are choosers and d_1 through d_4 are deciders. A valid set of choices is $c_1.me = c_3.me = 0$ and $c_2.me = 1$. If a link between c_3 and d_1 appears—perhaps it is newly added—then, this set of choices is no longer valid: c_1 and c_3 now share decider d_1 and so $c_1.me$ should differ from $c_3.me$. This could also occur were a new decider d_5 to appear that connects to both c_1 and c_3.

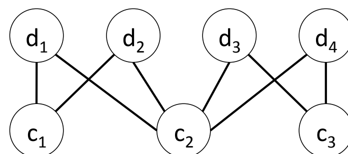


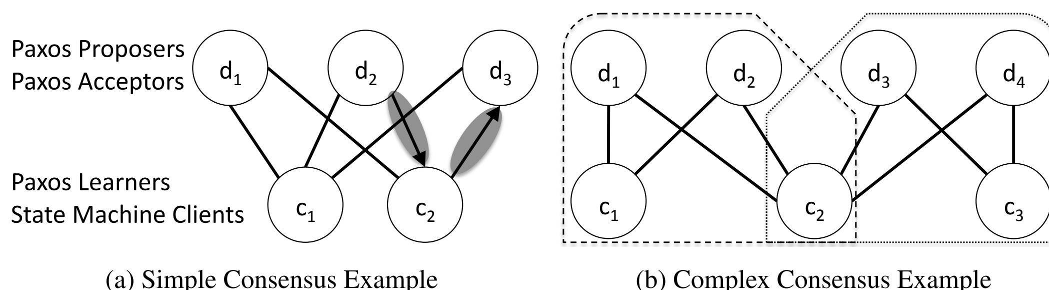
Figure 5.4: Stability Example

Thus, if an application based on LSP requires a chooser to know that its label will not change, then one would need to ensure, for example, that new connections between deciders and choosers cannot be created.

5.2.1 The Label Selection Problem with Consensus

One might be tempted to implement LSP with Consensus, because Consensus can be used to solve the arbitration problem in **Distinctness**. In this section, we discuss the difficulty of solving LSP with Consensus, beginning with a simple example. Assume the choosers and deciders are connected with a complete bipartite graph. One can implement a Paxos-based state machine in which the choosers implement both the clients of the state machine and the learners of Paxos, and the deciders implement the proposer and acceptors of Paxos, as illustrated in Figure 5.5a. A proposer and an acceptor (e.g. nodes d_2 and d_3 in the figure) can communicate by relaying via a chooser, selected randomly for each message to ensure liveness in the face of crashed choosers. One can implement the state machine so that the client (chooser) that submits the first command is given label 0, the second client is given label 1, etc. Or, one can have each client c choose a random $c.me$ and send it to the state machine; if $c.me$ has been previously requested, then c chooses a label that it has not yet learned has been assigned and tries

again. As long as no more than a minority of the deciders remain down (any number of choosers can remain down), this protocol implements the **Progress** and **Distinctness** properties of LSP.



(a) Simple Consensus Example

(b) Complex Consensus Example

Figure 5.5: Example Consensus Scenarios

If not all choosers have the same set of deciders, then using Consensus becomes messy. The Paxos state machine approach given above can be used by flooding all communication, thereby virtually connecting all processes. This has the drawback of possibly sending excessive messages; the path between any two processes can be as long as the total number of processes. It also unnecessarily restricts the choices of choosers not sharing a decider: all choosers' values will be unique even if they don't share deciders.

Another approach, and one that would not add such unnecessary restrictions to the choices, is to use multiple state machines. Any two choosers that share a decider use a common state machine to agree on unique labels. For example, consider the scenario shown in Figure 5.5b. A valid set of choices is $c_1.me = c_3.me = 0$ and $c_2.me = 1$. One could have two Paxos state machines, one with c_1, c_2, d_1, d_2 and one with c_2, c_3, d_3, d_4. In this approach, client c_2 chooses $c_2.me$ at random and sends it to both state machines. If $c_2.me$ has been previously assigned by either state machine, then it chooses another label and tries again.

This approach has its own set of problems. In this example, if any decider crashes then the solution is not live, because each instance of Paxos can tolerate only a minority of failures; with only two deciders, no permanent crashes can be tolerated. In addition, determining the set of state machines to run is not simple. The set can change as links and switches fail and recover, which adds further complexity.

5.3 The Decider/Chooser Protocol

In Section 5.2.1, we showed that using Consensus presents considerable difficulties in the face of dynamic network environments and changing sets of deciders and choosers. Instead, we develop here the Decider/Chooser Protocol (DCP), which is a randomized protocol that solves LSP with dynamic sets of deciders and choosers. The input to DCP is a bipartite graph between a set of *choosers* and a set of *deciders*, and the output is an assignment of labels to choosers such that all choosers have non-$\perp$ labels and no two choosers sharing a decider have the same label.

DCP proceeds as follows: A chooser c repeatedly chooses a label *me* from some range of labels and sends it to $c.deciders$, its set of neighboring deciders. If a decider d has not currently assigned *me* to another chooser, then it assigns *me* to c. To accomplish this, d maintains a table $d.chosen$ of labels that it has accepted from choosers. If *me* is not in $d.chosen$ for some other chooser c', then d sets $d.chosen[c]$ to *me* and sends a reply to c indicating that *me* was accepted. Otherwise, d sets $d.chosen[c]$ to $\perp$ (indicating that d has not assigned a value for c) and sends a reply to c indicating that its choice was rejected. d includes the set of labels assigned to other choosers in this reply as hints so c can avoid them when choosing another label.

To guard against difficulties caused by message duplication and reordering, each chooser attaches a monotonically increasing sequence number with each choice that it sends to a decider. A deciders d keeps records in $d.last_seq[c]$ of the largest sequence number seen from each chooser c and ignores messages from c with sequence numbers less than $d.last_seq[c]$. This allows us to consider channels between choosers and deciders as fair lossy FIFO channels: if p sends m_1 to q and then sends m_2 to q, q may receive m_1, m_2, both, or neither of these messages, but once it receives m_2 it will never receive m_1.

Listing 5.1 gives the decider's state and its two Actions F and G. Action G was described in the previous paragraph; Action F executes when decider d first learns that it is connected to a new chooser c. When this happens, d updates its set $d.choosers$ of known choosers and initializes $d.chosen[c]$ and $d.last_seq[c]$. Note that d never removes a chooser from these tables.

Listing 5.1: Decider Algorithm

```
1  set⟨Chooser⟩ choosers = ...
2  Choice[choosers] chosen = all[⊥]
3  int[choosers] last_seq = all[0]

   // when connected to new chooser c
4  F: when new chooser c
5      choosers ← choosers ∪ {c}
6      chosen[c] ← ⊥
7      last_seq[c] ← 0

   // respond to a message from chooser c
8  G: when receive ⟨s, x⟩ from c
9      if s≥last_seq[c]
10         last_seq[c] ← s
11         if ∃ c' ∈ (choosers \ {c}): chosen[c'] == x
12             chosen[c] ← ⊥
13         else
14             chosen[c] ← x
15     hints ← {chosen[c']_{∀ c' ∈ (choosers \ {c})}} \ {⊥}
16     send ⟨s, chosen[c], hints⟩ to c
```

Listings 5.2 and 5.3 together give the chooser's implementation, which includes its state, communication predicates and routines, and its four Actions *A* through *D*. We separate the chooser's description into two listings for readability; Listing 5.2 shows the routines, predicates and state used to implement FIFO channels whereas Listing 5.3 includes the chooser's actions and related state.

A chooser *c* stores the set of deciders that it knows exists ($c.deciders$), the sequence number of its current choice ($c.seq$), the value of its current choice ($c.me$), hints of choices to avoid according to each decider *d* ($c.hints[d]$), and the most recent sequence number acknowledged by each decider *d* ($c.last_ack[d]$).

The code makes use of a watchdog timer. The timer provides a variable *timeout* that is true iff the timer is unarmed. The operation TO_arm ensures that the timer is armed (so *timeout* is false). If TO_arm is not subsequently executed, then *timeout* eventually becomes true.

A chooser *c* has the following routines for communication with deciders:

SendTo(s,x,D): Send choice *x* with sequence number *s* to all deciders in *D*.

ResendTo(D): Resend the last message sent to all deciders in *D*.

ReceiveAck(s,d): Receive an acknowledgment from d on sequence number s.

A chooser c also has three macros to represent some of the re-used code related to channel activities:

HasReceivedAck(d): true iff c has received an acknowledgment from d for its latest choice.

CurrentChoice(s): true iff sequence number s acknowledges c's most recent choice.

OldChoice(s): true iff sequence number s acknowledges an obsolete choice.

These predicates and routines appear along with the associated state in Listing 5.2. The chooser's actions and related state are shown in Listing 5.3.

Listing 5.2: Chooser Channel Predicates and Routines (Unbounded Channels)

```
1  int[deciders] last_ack = all[0]
   // ⟺ c has an ack from d for its latest choice
2  boolean HasReceivedAck (d):
3      last_ack[d] == seq
   // ⟺ s acknowledges c's most recent choice
4  boolean CurrentChoice (s):
5      s == seq
   // ⟺ s acknowledges an obsolete choice
6  boolean OldChoice (s):
7      s < seq
8  SendTo (s,x,D):
9      foreach d ∈ D do
10         send ⟨s,x⟩ to d
11 ResendTo (D):
12     foreach d ∈ D do
13         send ⟨me,seq⟩ to d
14 ReceiveAck (s,d):
15     last_ack[d] ← s
```

When a chooser needs to select a new value (Action A), it selects one at random, avoiding potentially unavailable values, and sends this to neighboring deciders. It then arms the watchdog timer. When the timer fires (Action B), if the chooser's value has not yet been denied, it resends this selection on any channels necessary. When a chooser receives an acknowledgment from a decider (Action C), it stores the decider's hints if

they are up-to-date, and records the sequence number for the acknowledgment. If the message is a rejection, the chooser sets *c.me* back to undecided so that, via Action *A*, it will try again. Finally, when a new decider connects to a chooser and the chooser has already sent a proposal to other deciders, it sends its choice to the new decider (Action *D*). Note that a chooser crashing or recovering has no specific effect in the protocol: a decider only releases the label it has assigned to a chooser *c* when *c* asks for a new label. A decider *d* recovering can cause *c* to send *d* its latest choice via Action *D*.

Listing 5.3: Chooser Algorithm: Actions and State (Unbounded Channels)

```
1  set⟨Decider⟩ deciders = ...
2  int seq = 0
3  Choice me = ⊥
4  (set⟨Choice⟩)[deciders] hints = all[∅]

   // when needs to make a choice
5  A: when me == ⊥
6      choices ← domain(Choice) \ {⊥} \ {hints[d]∀ d ∈ deciders}
7      me ← choose from choices
8      seq ++
9      SendTo(seq,me,deciders)
10     TO_arm

   // retransmit last msg sent to deciders yet to acknowledge
11 B: when timeout ∧ (me ≠ ⊥)
12     ResendTo({d ∈ deciders: ¬HasReceivedAck(d)})
13     TO_arm

   // receive response from d
14 C: when receive ⟨s, chosen, hint⟩ from d
15     ReceiveAck(s,d)
16     if ¬OldChoice(s)
17         hints[d] ← hint
18     if CurrentChoice(s) ∧ (chosen == ⊥)
19         me ← ⊥

   // learn of decider d and round is active
20 D: when detect new decider d ∧ (me ≠ ⊥)
21     SendTo(seq,me,{d})
```

This algorithm is not guaranteed to terminate because any pair of choosers can conflict with one another. For example, let choosers c_1 and c_2 both choose the yet-unassigned label x and send it to deciders d_1 and d_2. Decider d_1 may receive c_1's message first and d_2 may receive c_2's message first. Thus, d_1 will reject c_2 and d_2 will reject c_1. This kind of conflict can continue for an unbounded time. However, as long as the domain from which a chooser c selects is large enough, there is a significant probabil-

ity with each choice that c chooses a label x that is different than any label currently accepted by any decider, and that is different than any label that any other chooser has currently chosen or will choose before c's message with x is received by all deciders. Once this occurs, c's value will be accepted by all deciders. This, in turn, increases the chances that another chooser will have its value chosen. Thus, as the running time tends to infinity, the probability of **Distinctness** holding tends to 1, as we show in Section 5.4.1.

5.3.1 Bounding the Channels

This protocol can be modified so that each chooser c limits the number of messages in flight to any given decider. Doing so limits the number of conflicting assignments that might occur in the future from some state: this is useful in computing the expected number of choosers that terminate in a given round (see Section 5.4.1).

We extend both the basic chooser code as well as its channel code to accommodate channel bounding. In fact, this extension requires only moderate changes to the protocol, as we are able to leverage the variable *seq* that is used to ensure that out-of-date messages are ignored. We add some simple book-keeping to the chooser's channel and some extra logic to the chooser's Action C. We consider the changes to the channel code first.

A chooser c stores the most recent sequence number acknowledged by each decider d ($c.last_ack[d]$). c also now stores, for each decider d, a set of unacknowledged sequence numbers ($c.sent[d]$), a tuple of the most recent choice and corresponding sequence number sent to d ($c.last_sent[d]$), and the sequence number of the most recent choice it would have sent to d if it were not limited by available channel space ($c.last_choice[d]$). The three predicates used for unbounded channels, *HasReceivedAck*, *CurrentChoice* and *OldChoice*, are modified to make comparisons based on values stored for a particular decider d. That is, they compare a sequence number s to the sequence number of the most recent choice with respect to a decider d ($c.last_choice[d]$) rather than to a global sequence number *seq*.

Choosers also have three new channel predicates:

CanSendTo(d): true iff there is space in the channel from *c* to *d*.

SentLatest(d): true iff *c* has sent its latest choice to *d*.

RecentAck(s,d): true iff the sequence number *s* acknowledges *c*'s most recent message to *d*.

Finally, the *SendTo*, *ResendTo* and *ReceiveAck* routines are updated to include book-keeping and verification, and to send new messages only when there is room in the channel:

SendTo(s,x,D): Send choice *x* with sequence number *s* to all deciders in *D*, keeping a copy for retransmission and bounding the channel.

ResendTo(D): Resend the last message sent (if applicable) to all deciders in *D*.

ReceiveAck(s,d): Receive an acknowledgment from *d* on sequence number *s*, update channel book-keeping variables.

Note that with channel bounding, a chooser maintains the sequence number of the most recent message sent to a decider *d* ($c.last_sent[d]$) as well as that of the most recent choice of $c.me$ with respect to *d* ($c.last_choice[d]$). A chooser may be temporarily unable to send its current choice to *d* if the channel between the two is full. This accounts for the subtle difference between the **RecentAck** and **CurrentChoice** predicates. Listing 5.4 shows the code for the channel-related predicates and routines when channels are bounded.

The chooser's actions change only slightly to accommodate channel-bounding. Actions *A* and *B* rely on the now channel-bounding routines *SendTo* and *ResendTo* for sending messages to deciders. This change is encapsulated in the channel code (described above). The chooser's Action *C* does change; a chooser stores a decider's hints only if the decider is responding to the most recent message *sent to that decider*. Additionally, if an acknowledgment is out-of-date and may have opened space in the channel, the chooser resends its current selection. Listing 5.5 shows the updated Action *C*. Modified code is shown in black, while unchanged code is grey.

Listing 5.4: Chooser Channel Predicates and Routines (Bounded Channels)

```
1  int[deciders] last_ack = all[0]
2  (set⟨int⟩)[deciders] sent = all[∅]
3  ⟨int,Choice⟩[deciders] last_sent = all[⟨0,⊥⟩]
4  int[deciders] last_choice = all[0]
5  int max_in_chan = a non-zero constant

   // ⟺ c has an ack from d for its latest choice
6  boolean HasReceivedAck (d):
7      last_ack[d] == last_choice[d]
   // ⟺ s acknowledges c's most recent choice for d
8  boolean CurrentChoice (s,d):
9      s == last_choice[d]
   // ⟺ s acknowledges an obsolete choice for d
10 boolean OldChoice (s,d):
11     s < last_choice[d]
   // ⟺ there is room in the channel to send to d
12 boolean CanSendTo (d):
13     | sent[d] | < max_in_channel
   // ⟺ c has sent its most recent choice to d
14 boolean SentLatest (d):
15     last_sent[d][0] == last_choice[d]
   // ⟺ s acknowledges c's most recent message to d
16 boolean RecentAck (s,d):
17     s == last_sent[d][0]

18 SendTo (s,x,D):
19     foreach d ∈ D do
20         if CanSendTo(d)
21             send ⟨s,x⟩ to d
22             sent[d] ← sent[d] ∪ {s}
23             last_sent[d] ← (s,x)
24         last_choice[d] ← s
25 ResendTo (D):
26     foreach d ∈ D do
27         if | sent[d] |>0
28             send ⟨last_sent[d]⟩ to d
29 ReceiveAck (s,d):
30     sent[d] ← sent[d] \ {i: i≤s}
31     last_ack[d] ← s
```

Listing 5.5: Chooser Algorithm: Actions and State (Bounded Channels)

```
1  set⟨Decider⟩ deciders = ...
2  int seq = 0
3  Choice me = ⊥
4  (set⟨Choice⟩)[deciders] hints = all[∅]

   // when needs to make a choice
5  A: when me == ⊥
6      choices ← domain(Choice) \ {⊥} \ {hints[d]∀ d ∈ deciders}
7      me ← choose from choices
8      seq ++
9      SendTo(seq,me,deciders)
10     TO_arm

   // retransmit last msg sent to deciders yet to acknowledge
11 B: when timeout ∧ (me ≠ ⊥)
12     ResendTo({d ∈ deciders: ¬HasReceivedAck(d)})
13     TO_arm

   // receive response from d
14 C: when receive ⟨s, chosen, hint⟩ from d
15     ReceiveAck(s,d)
16     if RecentAck(s,d)
17         hints[d] ← hint
18     if CurrentChoice(s,d) ∧ (chosen == ⊥)
19         me ← ⊥
20     if OldChoice(s,d) ∧ (me ≠ ⊥)
21         SendTo(last_choice[d],me,{d})

   // learn of decider d and round is active
22 D: when detect new decider d ∧ (me ≠ ⊥)
23     SendTo(seq,me,{d})
```

5.4 Analysis of the Decider/Chooser Protocol

In this section, we consider the correctness of DCP, first via proof and then by using model checking software.

5.4.1 Proof of Correctness of DCP

We prove here that DCP implements LSP. We assume that each channel contains no more than *max_in_channel* messages (Listing 5.4).

Our proof of correctness uses the following *Eventual Delivery* lemma:

Lemma 1 (Eventual Delivery)**.** *If chooser c sends a message* $[seq, me]$ *to d, and both c and d remain uncrashed and connected to each other, then eventually d receives a message* $[seq', me']$ *from c with* $seq' \geq seq$*, and eventually c receives an acknowledgment from d for a message with a sequence number* $seq'' \geq seq$*.*

Lemma 1 Proof. When c sends $[seq, me]$ to d, it will keep sending messages with some sequence number $seq' \geq seq$ to d via Actions A or B until it receives an acknowledgment (via Actions G, C) for $seq'' \geq seq$. □

Progress Proof. Initially $c.me$ is $\perp$. This variable is set to a non-$\perp$ value only by Action A, and Action A is continuously enabled starting with the initial state. Hence, if c does not remain crashed, $c.me$ will be set to some non-$\perp$ value. □

Distinctness Proof. A chooser that remains up will execute Action A one or more times. If it executes Action A a final time, we say that the chooser c's choice $c.me$ *stands*: from that point on, $c.me$ does not change. If c's value stands and c remains up, then $c.me \neq \perp$ since, otherwise, Action A is enabled.

We first show that two choosers that share a decider cannot both choose the same label and have their choices stand. That is, if two choosers' values $c_1.me$ and $c_2.me$ stand, then $c_1.me \neq c_2.me$. We then show that with high probability, the choosers will choose distinct values that stand.

(a) It is impossible for two choosers c_1 and c_2, both connected to decider d, to both set $c_1.me = c_2.me = x$ with $x \neq \perp$ and have these values stand. This is because c_1 will send $[seq_1, x]$ to d and c_2 will send $[seq_2, x]$ to d for some seq_1 and seq_2. Since both

leave *me* at x, neither sends a message with larger sequence numbers. From Lemma 1, d will eventually deliver both messages, and will reply $\perp$ to at least one of the choosers. Again from Lemma 1 the chooser will receive this acknowledgment and set *me* to $\perp$.

(b) Consider some point in the execution of the protocol. Let D be the set of deciders and C be the set of choosers. Let C^+ be the subset of choosers that will choose again by executing Action A — that is, $C \backslash C^+$ are the choosers whose choices stand.

If a chooser in C^+ chooses a value that some decider d has already given to another process, then it may receive $\perp$ from d. There are up to $|D| \times |C|$ distinct values that have already been given by some decider to some chooser. If multiple choosers in C^+ choose the same value, then some decider d they share may send one of them $\perp$.

If a chooser c in C^+ chooses a value that is in a message m that was sent by another chooser to a decider d but not yet delivered by d, then d may deliver m before receiving c's choice, and thus d will send $\perp$ to c. There are up to

$$|D| \times |C| \times max_in_channel$$

distinct values in channels.

Let $P(q, m, L)$ be the probability that if we take m samples with replacement from a domain of size L, then exactly q of them are distinct. In our case, L corresponds to the label domain, m to the number of choosers still attempting to select values, and q to the number of choosers that choose values that will stand as labels because they are distinct. Let *Choice* be the domain from which choosers choose. Even if all choosers pick distinct values, there are up to

$$|D| \times |C| + |D| \times |C| \times max_in_channel$$

values that, if chosen, will result in a chooser receiving $\perp$. Thus, the probability that the choosers in C^+ all choose values that stand is at least

$$P(|C^+|, |C^+|, |Choice| - |C| \times |D|(1 + max_in_channel))$$

In fact, the probability that some choosers choose values that stand is positive. Thus, with enough choices, C^+ will continue to decrease with high probability, until it becomes empty. $\square$

5.4.2 Model Checking DCP

We implemented DCP in Mace [21, 40], which is a language for distributed system development. The Mace toolkit includes both a model checker [39] that allows one to verify the correctness of the system and a simulator [41] for testing timed behavior. A major benefit of Mace is that Mace code compiles into standard C++ code, which allows one to deploy code that has been model checked.

A few differences between our implementation [72] and the listings of Section 5.3 bear special mention. A Mace service contains variables, messages, and code segments called *transitions*, which are executed in reaction to four types of events: timer expiration, message receipt, error indication, and downcalls from applications using the service. Mace cannot constantly test the guards for the actions shown in our listings; instead, we determine when each guard may become true and evaluate each guard at all necessary points (executing the corresponding action if necessary). A decider's Action G executes upon receipt of a message from any chooser, whereas Action F executes only upon receipt of a message from a chooser that has not yet been encountered. The case for the chooser is more complicated. Action A needs to execute whenever $c.me = \bot$. This can occur initially upon startup of the chooser, upon recovery from a crash (if the value was not set prior to the crash), and as the result of a rejection message in Action C. So, the guard for Action A is evaluated at these three times. The guard for Action B is evaluated when the watchdog timer fires and also upon reset. Action C executes directly as a result of a message receipt from a decider. The guard for Action D is evaluated whenever a chooser receives a message from a decider not currently in $c.deciders$.

Both the Mace model checker and the Mace simulator construct a set of behaviors of the program. Mace knows the sources of nondeterminism (in our case, node failures, UDP packet reordering and loss, and random number generation) and so constructs all behaviors over which it checks for violations of any safety or liveness property. The model checker differs from the simulator in how the sets of behaviors are constructed: the model checker does a breadth first construction while the simulator chooses, at random, a value for each nondeterministic event to construct a behavior. Since the tests cannot be run for an infinite time, each behavior is extended to a maximum depth (set with a run-time parameter).

We used the Mace model checker to check the liveness properties **Progress** and **Distinctness**. We considered three types of topologies, all modifications of a 3-level fat tree.[2] We constructed all three topologies by first creating a 3-level fat tree using k-port switches, with $k = 4$, 6, 8, 10 and 12, and extracting the bottom two levels of nodes. The first topology (*fat tree-based*) consists of this bipartite graph embedded in the lowest two levels of a fat tree. For our *random bipartite* topology, we began with the fat tree-based topology and removed all edges in the graph. We then generated edges between each lower-level node and a randomly chosen set of $\frac{k}{2}$ upper-level nodes. Finally, we also created a *complete bipartite* graph between the nodes within the fat tree-based topology. The complete bipartite graph topology imposes the most restrictions on DCP because all choosers share all deciders: no two choosers can have the same label.

For each topology type, we show that the **Progress** and **Distinctness** properties eventually hold. We also verify the channel bounding aspects of the protocol (see Section 5.3.1) using safety properties.

5.5 Performance of the Decider/Chooser Protocol

In this section, we consider the performance of DCP, that is, we explore the time required for an instance of DCP to satisfy **Progress** and **Distinctness**. We begin in Section 5.5.1 by mathematically analyzing DCP and then we simulate its behavior in Section 5.5.2.

5.5.1 Analyzing DCP Performance

Recall that $P(q,m,L)$ expresses the probability that if we take m samples with replacement from a domain of size L, then exactly q of them are distinct. In other words, this value expresses the probability that any given set of choosers will succeed (and therefore exit the competition) during any given round. Therefore, sequences of $P(q,m,L)$ values can form probability distributions for the completion of DCP instances.

[2] We selected a fat tree as a base topology because it arises in the context of ALIAS (Chapter 4).

$P(q,m,L)$ can be computed as follows. Let $S(m)$ be the set of different sets of positive numbers that sum to m. For example,

$$
\begin{aligned}
S(6) = &\{\{1,1,1,1,1,1\},\{1,1,1,1,2\},\{1,1,1,3\},\{1,1,4\},\\
&\{1,5\},\{1,1,2,2\},\{1,2,3\},\{2,4\}\\
&\{2,2,2\},\{3,3\},\{6\}\}
\end{aligned}
$$

We use each element of $S(m)$ to denote a *configuration* of the m choosers. So, $\{1,5\}$ represents a configuration of six choosers in which five choose the same label, and the sixth chooses another label.

Let $C(s)$ be the number of ways the m choosers can be grouped into a configuration s and let $T(s,L)$ be the number of unique ways elements of L can be assigned to configuration s. That is,

$$T(s,L) = |s|! \times \binom{L}{|s|} = \frac{L!}{(L-|s|)!}$$

The probability that m choosers result in configuration s is $C(s) \times T(s,L)/L^m$. For example, let $s = \{1,1,2,2\}$.

$$C(s) = \binom{6}{1} \times \binom{5}{1} \times \frac{\binom{4}{2}}{2!} \times \frac{\binom{2}{2}}{2!} = 45$$

$T(s,10)$ for $s = \{1,1,2,2\}$ is 5,040 and the probability that the choosers are in configuration $\{1,1,2,2\}$ for $L = 10$ is

$$\frac{45 \times 5040}{10^6} = 0.2268$$

Finally, let $S_q(m)$ be the subset of $S(m)$ that contain exactly q values of 1. For example,

$$S_2(6) = \{\{1,1,4\},\{1,1,2,2\}\}$$

Then we have

$$P(q,m,L) = \frac{\sum_{s \in S_q(m)} C(s) * T(s,L)}{L^m}$$

So, $P(2,6,10)$ is

$$
\begin{aligned}
P(2,6,10) &= \frac{C(\{1,1,2,2\}) \times T(\{1,1,2,2\},10) + C(\{1,1,4\}) \times T(\{1,1,4\},10)}{10^6} \\
&= 0.2268 + 0.0108 \\
&= 0.2376
\end{aligned}
$$

That is, just under a quarter of the time, if six choosers choose labels from 0 to 9, exactly two will end up with labels distinct from all the other chosen labels. Over 95% of the time that this happens, two other choosers will choose a third label and the remaining two will choose a fourth label, and under 5% of the time the four remaining choosers will choose the same label.

To give an idea of the probability of choosing distinct values, Figure 5.6 shows a plot for $P(q,32,L)$ for $L = 32$, 64 and 128 (that is, 32 choosers and labels with 5, 6 and 7 bits). With $L = 128$, the most likely value for q is 26, which would leave 6 choosers choosing again. When $L = 32$ (the smallest possible value for L) , the most likely value for q is 12, which leaves 20 choosers choosing again. This shows how decreasing L increases the expected convergence time.

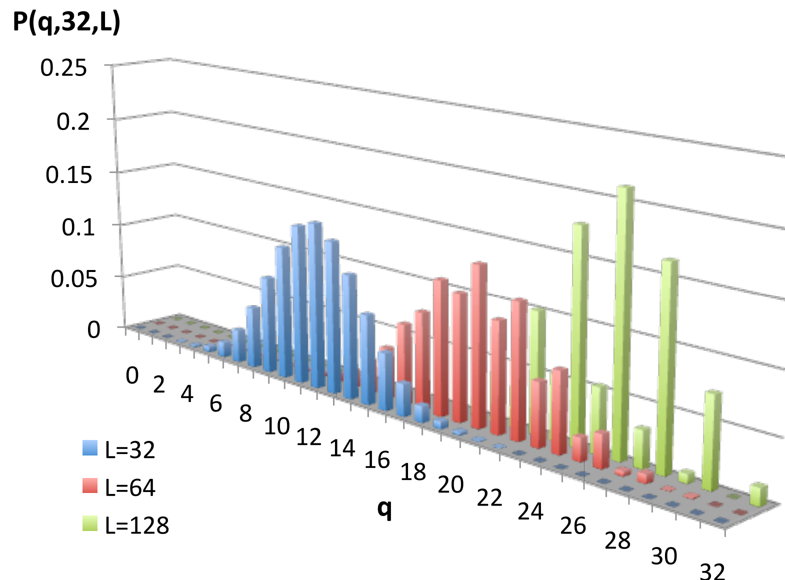


Figure 5.6: $P(q,32,L)$ with $L = 32,64,128$

It would be useful to compute an upper bound on the convergence time of DCP, but it has proven difficult to do so: as more choosers choose values that stand, fewer values remain for other choosers, but the number of choosers competing for values decreases. For the purposes of ALIAS, simulation has been sufficient to show that the expected convergence time is short.

5.5.2 Simulating DCP Performance

After verifying **Progress** and **Distinctness** with the Mace model checker, we used the Mace simulator to determine how quickly DCP converges over the same three types of topologies (fat tree-based, random bipartite and complete bipartite). In addition to the topology type, we varied the number of choosers and deciders[3] ($|C|$) as well as the size of the domain ($|L|$) from which the choices are made. We simulated $|L| = |C|$, which is the smallest domain that allows for a solution with a bipartite graph, $|L| = 1.5|C|$ and $|L| = 2|C|$. For a given number of choosers and deciders, there are 9 possible configurations, corresponding to the three topology types and the three label domain sizes. For each configuration, we simulated 100 different executions (thus giving different values for the nondeterministic events). Table 5.1 shows the results of these simulations. Each column gives the percentage of choosers (averaged over 100 executions) that have converged after a given number of choices.

For the first two types of topologies, most choosers converge within 2 choices, and only a few require 3-5 choices before settling on a value. For the complete bipartite graphs, especially when $|L| = |C|$, it takes longer for all choosers to converge because each chooser must choose a distinct value. Even so, in most cases over 90% of the choosers converge with 2 choices and over 99% converge with 4 choices. But, the time for all to decide under such constraints can sometimes be long. For example, in one particular execution for the complete bipartite topology with $|L| = |C| = 18$, the hint messages to a single chooser were repeatedly dropped, and the chooser chose already-taken labels for 89 cycles before converging.

[3] The number of deciders is equal to the number of choosers.

Table 5.1: Convergence Time of DCP

Configuration			% Choosers converged vs. Number of Choices									
Topo	**|C|**	**|L|**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**35**	**55**	**89**
fat tree-based	**8**	**8**	94.13	99.88	**100**							
		12	96.38	**100**								
		16	95.50	**100**								
	18	**18**	94.33	99.94	**100**							
		27	96.50	99.94	**100**							
		36	97.44	**100**								
	32	**32**	95.38	99.91	**100**							
		48	96.56	**100**								
		64	97.09	99.94	**100**							
	50	**50**	97.36	**100**								
		75	97.74	**100**								
		100	95.54	99.98	**100**							
	72	**72**	96.01	99.89	**100**							
		108	97.43	**100**								
		144	98.01	**100**								
random	**8**	**8**	88.13	98.75	**100**							
		12	92.88	**100**								
		16	93.88	99.88	**100**							
	18	**18**	87.89	98.11	99.83	**100**						
		27	92.33	99.17	**100**							
		36	93.94	99.39	99.94	**100**						
	32	**32**	84.69	98.16	99.88	**100**						
		48	90.16	99.19	99.97	**100**						
		64	92.78	99.53	**100**							
	50	**50**	83.80	97.02	99.62	99.94	**100**					
		75	89.58	98.92	99.98	**100**						
		100	91.78	99.38	99.98	99.98	**100**					
	72	**72**	84.19	97.57	99.71	99.96	**100**					
		108	89.40	98.81	99.89	**100**						
		144	92.40	99.29	99.99	**100**						
complete bipartite	**8**	**8**	50.00	70.50	81.00	84.88	86.00	87.88	88.88	99.63	**100**	
		12	68.50	91.38	97.75	99.50	99.88	**100**				
		16	75.50	96.25	99.25	**100**						
	18	**18**	32.17	43.44	50.44	53.56	55.11	56.22	57.61	90.56	98.72	**100**
		27	59.94	84.44	95.17	98.83	99.56	99.83	99.89	**100**		
		36	68.78	91.33	98.28	99.72	**100**					

5.6 DCP in Data Center Labeling

In this section, we consider the application of DCP in the context of automatic label assignment in large-scale data center networks. ALIAS (Chapter 4) operates over indirect hierarchical topologies [66], in which servers (end hosts) connect to the lowest level of a multi-rooted tree of switches. Such topologies currently underly many data center networks [4, 13, 18, 28, 56]. Switches at each level of the hierarchy but the topmost select *coordinates* and these coordinates combine to form hierarchically meaningful labels; a label corresponds to a path from the root of the tree to a host. In data center networks, a key concern is automatic configuration in the face of a dynamically changing topology, so DCP is well-suited to this environment.

5.6.1 Distributing the Chooser

Recall that the input to DCP is a bipartite graph of choosers connected to deciders; each chooser and decider resides in a single process. Before we discuss DCP as a solution for coordinate assignment in ALIAS, we first present an extension to the basic protocol, in which a logical chooser can be distributed across multiple nodes. These nodes cooperate to select a single shared label. We will use this extension when we apply DCP within ALIAS's multi-rooted trees in Section 5.6.2. A full protocol derivation appears in Section 5.7.

We begin with the set of nodes that wish to cooperate in order to select a shared label, and introduce a new type of process for these nodes: the *chooser relay*. Each node within the cooperating set functions as a relay, providing a connection from the distributed chooser to one or more deciders. A distributed chooser's set of neighboring deciders consists of the union of all deciders with a direct link to one or more of the chooser's relays. We then introduce another type of process, the *chooser representative*. Each distributed chooser has exactly one representative, which performs all of the functionality of the chooser (Actions A through D of Listing 5.5), and communicates with deciders via the chooser's relays. This representative can be co-located with one of the relays or it can be a separate node; the only requirement is that it is able to communicate with all of the chooser's relays.

The structure of a distributed chooser with a separately located representative is shown in Figure 5.7. In the figure, the nodes marked d_1 through d_4 are deciders, and the dotted lines denote the boundaries of the two distributed choosers. Within $Chooser_1$ and $Chooser_2$, rel_1 through rel_5 are relays, and rep_1 and rep_2 are representatives.

For a distributed chooser $\mathscr{C}$, we denote with $Relays(\mathscr{C})$ the set of relays in $\mathscr{C}$ and with $Repr(\mathscr{C})$ the process that represents $\mathscr{C}$. Together, the processes in $Relays(\mathscr{C}) \cup Repr(\mathscr{C})$ make up the distributed chooser $\mathscr{C}$. Similarly, for an individual node r, we use $Relays(r)$ and $Repr(r)$ to denote the relays and representative of the chooser in which r participates. In our example of Figure 5.7, the relays and representatives for the two distributed choosers are as follows: $Relays(Chooser_1) = \{rel_1, rel_2, rel_3\}$, $Repr(Chooser_1) = rep_1$, $Relays(rel_4) = \{rel_4, rel_5\}$ and $Repr(rel_4) = rep_2$.

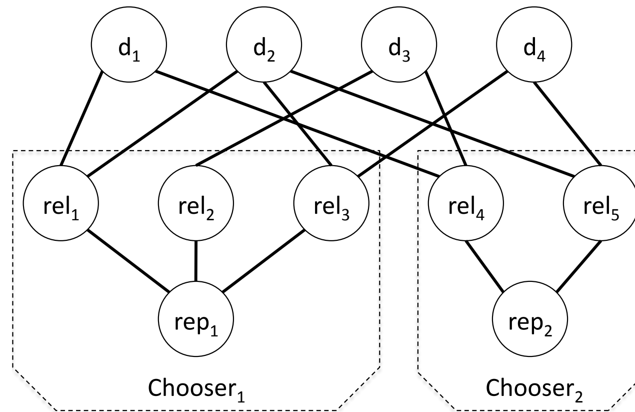


Figure 5.7: Distributed Chooser

There are some issues to address in implementing a distributed chooser. The first is that of communication between the chooser's representative and its relays. We support this communication with two queues, *Send* and *Receive*:

Send: is a queue of messages stored at each relay r, that implements a virtual channel from $Repr(r)$ to the deciders. $Repr(r)$ appends a message m to this queue by sending a message to $Relays(r)$. When a relay receives this message, it adds m to the end of its own copy of $Send$. $Repr(r)$ never takes an action based on the value of $Send$, and so a relay r need not notify $Repr(r)$ when it removes m from $Send$.

Receive: is a queue of messages stored at $Repr(\mathscr{C})$ that accumulates messages sent to a chooser $\mathscr{C}$ from its deciders. A relay r for chooser $\mathscr{C}$ appends messages to this queue by sending them to $Repr(r)$.

These changes only affect the chooser's actions and channel code slightly; the *SendTo* and *ResendTo* channel functions (Listing 5.4) append to the *Send* queue rather than sending messages directly to deciders, and a chooser's Action C (Listing 5.5) is triggered by a non-empty *Receive* queue rather than by direct receipt of a message from a decider.

A second issue has to do with the connections between a distributed chooser and a decider: a chooser may be connected to each of its deciders via various subsets of its relays. Rather than having the representative keep track of which relays are connected to each decider, it can simply send all messages to all of its relays. Each relay then filters

out messages destined to deciders that it does not neighbor. While this increases message load, it does not require that a representative keep track of the possibly changing connections between the relays and deciders. Similarly, since a chooser may connect to a decider d via multiple relays, it has the option of selecting only a single such relay for each message sent to d, or it may use any subset of the relays connected to d. This again represents a tradeoff between message load and complexity.

A third issue has to do with data representation at both the chooser and the decider. Since a representative may have multiple paths to a given decider via different relays, it indexes any channel-related variables over both relays and deciders. This is intuitive, as the relays act as virtual channels between a chooser's representative and its deciders. Therefore, channel-related variables should be indexed over the entire channel, relays *and* deciders. Also, recall that a decider indexes its *chosen* and *last_seq* maps over choosers. To support distributed choosers, a decider indexes these maps over the entire chooser, both the relays and the representative.

Finally, changing network conditions may affect connectivity between a chooser's representative and its relays, which can cause the representative to change. Any node that could ever be the representative for a chooser watches the Receive queue and maintains any channel-related state for that chooser. Such a node also executes a modified version of Action C (Listing 5.5) that properly updates state upon receipt of an acknowledgment so that if it subsequently becomes the chooser's representative, it will have correct acknowledgment and channel capacity information.

5.6.2 The Decider/Chooser Protocol in ALIAS

Figure 5.8 shows an example multi-rooted tree of switches. In the figure, hosts have been omitted for space and clarity. Switches are categorized as being at levels L_1 through L_3, from the bottom of the tree upwards, and the S_1 through S_{10} notations indicate switches' unique identifiers.

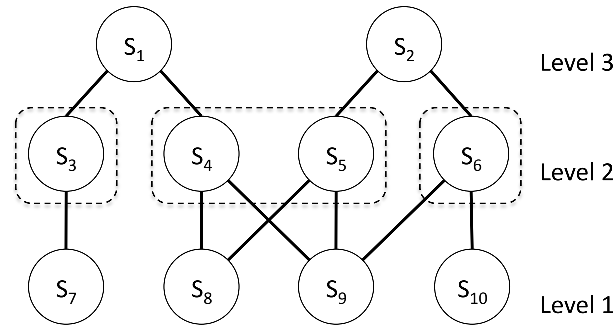


Figure 5.8: Multi-Rooted Tree Topology

In ALIAS, a host h's label is a pair of coordinates c_2c_1, where c_1 is the coordinate of the level L_1 switch s_1 to which h is connected and c_2 is the coordinate of a switch at level L_2 that neighbors s_1.[4] Since there are multiple paths from the root of the tree to a host h, hosts in ALIAS have multiple labels. ALIAS forwarding sends data packets to the root of the tree, at which point a packet's destination label specifies a path to the destination. This is based on up*/down* style forwarding, as introduced in Autonet [65].

Since switches forward packets downward based on coordinates within the destination label, it follows that any two children of a given switch should have distinct coordinates; in this way a parent switch can select which child should be the next hop for any given destination label. This maps nicely to a simple application of simultaneous instances of DCP, one per tree level, as we show in Figure 5.9. Each instance of DCP is used to select coordinates for the instance's choosers. Since there are two levels of switches (L_1 and L_2) that need coordinates, we apply an instance of DCP for each. In the first instance, all L_1 switches act as choosers for their L_1-coordinates and all L_2 switches act as deciders. In the the second instance, all L_2 switches act as choosers for their L_2-coordinates and all L_3 switches are deciders.

The application of DCP to ALIAS L_2-coordinate assignment shown in Figure 5.9 is simple but not efficient in terms of the number of labels it assigns to each host. To address this, ALIAS leverages the hierarchical structure of the the topology in order to allow certain sets of switches located near to one another in the hierarchy to share label prefixes. This in turn leads to more compact forwarding state, a desirable property in the data center.

[4]Top-level switches are not assigned coordinates.

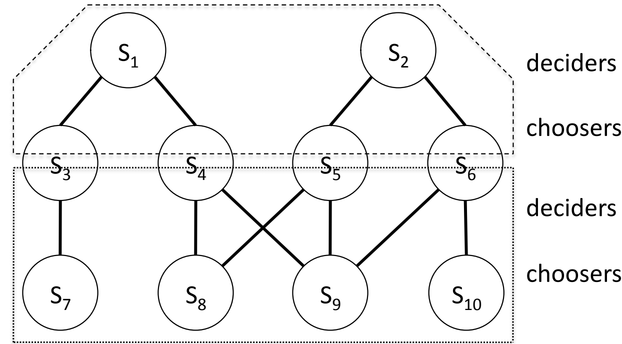


Figure 5.9: Simple DCP in ALIAS

To enable these shared label prefixes, ALIAS introduces the concept of a *hypernode*. In an n-level tree, all switches (other than those at L_n) are partitioned into hypernodes. A hypernode at level L_i is defined as a maximal set of L_i switches that connect to an identical set of L_{i-1} hypernodes below. The base case for this recursive definition has each hypernode at L_1 contain a single switch. For a 3-level tree, the only interesting hypernodes are made up of L_2 switches. Figure 5.8 shows the sample topology's hypernodes with dotted lines.

Consider a packet with destination label c_2c_1. The coordinate c_1 corresponds to an L_1 switch s_1 that is connected to the packet's destination. Since all L_2 switches in a hypernode connect to the same set of L_1 switches below, an L_3 switch can send the packet to any switch in an L_2 hypernode that neighbors s_1. Therefore, the switches in a hypernode can share a single coordinate, as all are equivalent with respect to forwarding reachability. Coordinate sharing among hypernode members reduces the number of labels assigned to an host and increases the efficiency of ALIAS.

To accommodate shared L_2-coordinates, we apply the distributed chooser version of DCP. Each hypernode corresponds to a single chooser, in which the L_2 member switches are relays. By definition, an L_2 hypernode consists of L_2 switches that connect to the same set of L_1 switches, and so we are guaranteed to have an L_1 switch that can reach all L_2 relays and therefore can act as the chooser's representative. We select between a set of possible representatives via any deterministic function, e.g. the L_1 switch with the smallest MAC address.[5] Figure 5.10 shows the three distributed choosers for

[5]In general, it is acceptable to use any deterministic function such that the result is identical at all decision points of the function.

our example topology's L_2-coordinate assignment. These choosers consist of relays $\{s_3\}$ $\{s_4,s_5\}$ and $\{s_6\}$, represented by $\{s_7\}$, $\{s_8\}$ and $\{s_9\}$, respectively.

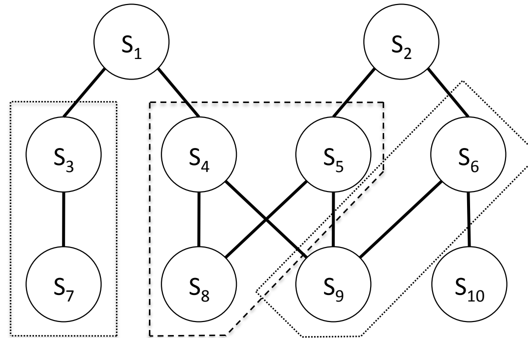


Figure 5.10: Assigning L_2-Coordinates using Distributed Choosers

We have completed a full protocol derivation from Listings 5.1 and 5.5 to a complete solution for ALIAS coordinate selection, which we present in Section 5.7. In addition to our Mace implementation of DCP [72], we have also built a second, slightly different implementation of ALIAS [73].[6] We have also model checked our second implementation with respect to the **Progress** and **Distinctness** properties, and have found through simulation that distributed choosers converge within only a few choices for the networks tested.

5.6.3 Eliminating M-Graphs in ALIAS

The up*/down* forwarding used by ALIAS separates L_1-to-L_n forwarding from L_n-to-L_1 forwarding in an n-level hierarchy. Because of this, a topology that we call an *M-graph* can lead to a forwarding ambiguity. When data forwarding follows an up-down path, two L_1 switches must be no more than $2(n-1)$ hops apart to directly communicate with one another. An M-graph occurs when two L_2 hypernodes hn_1 and hn_2 do not have an L_3 decider in common, and thus may select the same coordinate, but an host h can communicate with descendants of both hn_1 and hn_2.

An example M-graph is shown in Figure 5.11. Each switch is marked with a unique identifier (S_1 through S_9) as well as its coordinate if at levels L_1 or L_2. Each

[6] Our second implementation does not operate in rounds. Choosers and deciders continuously send messages, ignoring incoming messages that are redundant with respect to already processed information.

host is marked with its unique identifier (H_1 through H_3) and its label (created by concatenating ancestor switches' coordinates). The L_2 hypernodes in the figure are $\{s_3\}$, $\{s_4,s_5\}$ and $\{s_6\}$ and they form distributed choosers represented by $\{s_7\}$, $\{s_8\}$ and $\{s_9\}$, respectively.

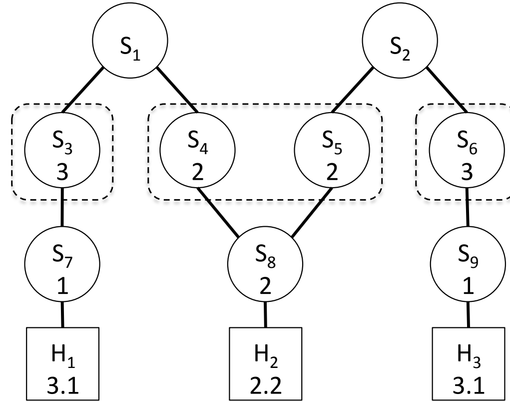


Figure 5.11: Example M-Graph

Because data forwarding follows an up-down path, s_7 and s_9 cannot communicate directly with one another. They can, though, both communicate with a third L_1 switch s_8 (and its neighboring host H_2). Since the L_2 hypernodes connected to s_7 and s_9 ($\{s_3\}$ and $\{s_6\}$) do not share a parent they can have the same L_2-coordinate, in this case 3. And, since s_7 and s_9 have no parent in common, they can have the same L_1-coordinate, in this case 1. This is the ambiguity: s_8 can communicate with two different switches, s_7 and s_9, that may legally be assigned the same label.

In practice, this is not a problem because of the randomness of DCP: ambiguous labels are rarely generated. When ALIAS finds such labels, it follows a simple detection-and-recovery approach. If desired, though, we can prevent this ambiguity in two different ways, each involving an application of DCP. First, we can simply add the set of L_1 switches that are 3 hops away from each L_2 hypernode to the set of deciders for that hypernode's chooser.[7] For example, in Figure 5.11, s_8 would be a decider for hypernodes $\{s_3\}$ and $\{s_6\}$. This removes the possibility of ambiguity by ensuring that any two hypernodes both reachable from a third L_1 switch have distinct labels. This solution increases implementation complexity slightly, because L_2 relays are not directly

[7]More generally, for an L_i hypernode, we add to the deciders all L_1 switches that are $2n-i-1$ hops from L_1.

connected to all L_1 deciders and so send messages to deciders via tunneling or other similar mechanisms.

Alternatively, one can prevent this ambiguity by assigning coordinates to L_3 switches. In our example, the labels of s_7 and s_9 (and therefore H_1 and H_3) would differ in this new coordinate. To do this, L_3 switches are grouped into hypernodes based on connectivity to L_2 hypernodes. L_3 hypernodes then form distributed choosers, using, for example, common L_1 descendants as representatives. L_1 switches reachable in 2 hops from the L_3 hypernodes are the deciders for this instance of DCP. This approach increases the distance between a chooser's representative and relays. Like the previous solution, this approach leads to indirect connections between relays and deciders. However, unlike the first solution, this method introduces the additional complexity and costs of grouping L_3 switches into hypernodes and assigning L_3-coordinates. For this reason, we would favor the former solution.

5.7 From DCP to ALIAS Coordinate Selection

In this section we present the full derivation of the ALIAS protocol (Chapter 4) from the basic version of DCP. We first review significant ALIAS environment and details, as well as the basic chooser and decider algorithms. Next, we discuss hypernode calculation, and we refine the chooser to select multiple coordinates simultaneously. Finally, we apply the distributed chooser refinement described in Section 5.6.1. We present our derivation in the context of a 3-level tree. Though our solution extends to trees of arbitrary depth, we use this limitation for readability.

5.7.1 ALIAS and DCP Review

Recall that ALIAS switches form an indirect hierarchical topology [66] of n levels, with end hosts connected to switches at the lowest level, L_1. Switches select coordinates that are combined to form topologically meaningful labels; coordinates concatenate along a path from the root of the tree to an end host in order to form a label for that end host. Since there are multiple paths from the root of the tree to any given end host, end hosts have multiple labels.

ALIAS switches are grouped into *hypernodes*: L_i switches that connect to identical sets of L_{i-1} hypernodes form L_i*-hypernodes* that share a single coordinate. Each switch at L_1 is in its own hypernode, and switches at the root of the tree are not grouped into hypernodes as they do not require coordinates. Each L_i switch is a member of exactly one hypernode,[8] and L_i switches may be connected to L_{i+1} switches in multiple L_{i+1}-hypernodes. Coordinate sharing within hypernodes serves to ultimately reduce the number of labels per end host in ALIAS. In a 3-level topology, only L_2 switches are grouped into hypernodes; L_1 hypernodes are trivial, with one L_1 switch per hypernode, and L_3 switches are at the root of the hierarchy and do not require coordinate assignments or hypernodes.

We begin our derivation by repeating the basic algorithms for the decider's actions (Listing 5.1) and the chooser's actions (Listing 5.5) and channel code (Listing 5.4), in Listings 5.6, 5.7, and 5.8, respectively. There is one small change to the chooser's channel code: we add routines to clear a chooser's channel corresponding to a particular decider, and to copy channel state from one of chooser's deciders to another. Also, we replace the null coordinate value $\perp$ with -1, as this corresponds to the null value of a coordinate in the implementation of ALIAS.

5.7.2 Computing Hypernodes

Prior to assigning coordinates, ALIAS hypernodes need to be identified. We select a *representative* L_1 *switch* for each L_i hypernode via a deterministic function, e.g. the L_1 switch with the smallest UID (in our implementation, MAC address) among those reachable via $(i-1)$ downward hops from switches in the hypernode. This L_1 switch functions as a distributed chooser's representative (Section 5.6).

Listings 5.9 and 5.10 show the actions executed by L_2 switches and L_1 switches, respectively, for computing hypernodes and representative L_1 switches. In Action P, each time an L_2 switch's set of neighboring L_1 switches changes, it sends this set of neighboring L_1 switches to all of its L_1 neighbors.[9] An L_1 switch stores this set (Action Q) and computes the sending L_2 switch's hypernode. Regardless of whether they rep-

[8] The set of L_i hypernodes forms a set of equivalence classes over the L_i switches in a topology.

[9] It also sends this set to neighboring L_3 switches to facilitate its own hypernode's coordinate assignment, as explained in Section 5.7.4.

Listing 5.6: Decider Algorithm
(Repeated from Listing 5.1)

```
1  set⟨Chooser⟩ choosers = ...
2  Choice[choosers] chosen = all[-1]
3  int[choosers] last_seq = all[0]

   // when connected to new chooser c
4  F: when new chooser c
5      choosers ← choosers ∪ {c}
6      chosen[c] ← -1
7      last_seq[c] ← 0

   // respond to a message from chooser c
8  G: when receive ⟨s, x⟩ from c
9      if s≥last_seq[c]
10         last_seq[c] ← s
11         if ∃ c' ∈ (choosers \ {c}): chosen[c'] == x
12             chosen[c] ← -1
13         else
14             chosen[c] ← x
15     hints ← {chosen[c']_∀ c' ∈ (choosers \ {c})} \ {-1}
16     send ⟨s, chosen[c], hints⟩ to c
```

Listing 5.7: Chooser Algorithm: Actions and State
(Bounded Channels, Repeated from Listing 5.5)

```
1  set⟨Decider⟩ deciders = ...
2  int seq = 0
3  Choice me = -1
4  (set⟨Choice⟩)[deciders] hints = all[∅]

   // when needs to make a choice
5  A: when me == -1
6      choices ← domain(Choice) \ {-1} \ {hints[d]∀ d ∈ deciders}
7      me ← choose from choices
8      seq ++
9      SendTo(seq,me,deciders)
10     TO_arm

   // retransmit last msg sent to deciders yet to acknowledge
11 B: when timeout ∧ (me ≠ -1)
12     ResendTo({d ∈ deciders: ¬HasReceivedAck(d)})
13     TO_arm

   // receive response from d
14 C: when receive ⟨s, chosen, hint⟩ from d
15     ReceiveAck(s,d)
16     if RecentAck(s,d)
17         hints[d] ← hint
18     if CurrentChoice(s,d) ∧ (chosen == -1)
19         me ← -1
20     if OldChoice(s,d) ∧ (me ≠ -1)
21         SendTo(last_choice[d],me,{d})

   // learn of decider d and round is active
22 D: when detect new decider d ∧ (me ≠ -1)
23     SendTo(seq,me,{d})
```

Listing 5.8: Chooser Channel Predicates and Routines (Bounded Channels, Repeated from Listing 5.4)

```
1  int[deciders] last_ack = all[0]
2  (set⟨int⟩)[deciders] sent = all[∅]
3  ⟨int,Choice⟩[deciders] last_sent = all[⟨0,-1⟩]
4  int[deciders] last_choice = all[0]
5  int max_in_chan = a non-zero constant

   // ⟺ c has an ack from d for its latest choice
6  boolean HasReceivedAck (d):
7      last_ack[d] == last_choice[d]

   // ⟺ s acknowledges c's most recent choice for d
8  boolean CurrentChoice (s,d):
9      s == last_choice[d]

   // ⟺ s acknowledges an obsolete choice for d
10 boolean OldChoice (s,d):
11     s < last_choice[d]

   // ⟺ there is room in the channel to send to d
12 boolean CanSendTo (d):
13     | sent[d] | < max_in_channel

   // ⟺ c has sent its most recent choice to d
14 boolean SentLatest (d):
15     last_sent[d][0] == last_choice[d]

   // ⟺ s acknowledges c's most recent message to d
16 boolean RecentAck (s,d):
17     s == last_sent[d][0]

18 SendTo (s,x,D):
19     foreach d ∈ D do
20         if CanSendTo(d)
21             send ⟨s,x⟩ to d
22             sent[d] ← sent[d] ∪ {s}
23             last_sent[d] ← (s,x)
24         last_choice[d] ← s

25 ResendTo (D):
26     foreach d ∈ D do
27         if | sent[d] |>0
28             send ⟨last_sent[d]⟩ to d

29 ReceiveAck (s,d):
30     sent[d] ← sent[d] \ {i: i≤s}
31     last_ack[d] ← s

32 ClearChannel (d):
33     last_ack[d]←0
34     sent[d].clear()
35     last_sent[d] ← (0,-1)
36     last_choice[d]←0

37 CopyChannel (d,ref):
38     last_choice[d]←last_choice[ref]
```

resent any hypernodes, all L_1 switches perform computation to determine the set of L_2 hypernodes to which they are connected. An L_1 switch runs nearly identical code (omitted for space) when it detects the disconnection of an L_2 switch. There is also logic to ensure that messages are eventually delivered, and that they are delivered in order. This code is also omitted from the listings for brevity.

Listing 5.9: Hypernode Computation: L_2 Switches

```
1  set⟨Switch⟩ L1s = ...                    // corresponds to choosers of Listing 5.6
2  set⟨Switch⟩ L3s = ...

   // when L1 neighbors change
3  P: when detect change in L1s
4      foreach n ∈ {L1s ∪ L3s} do
5          send ⟨L1s⟩ to n
```

Listing 5.10: Hypernode Computation: L_1 Switches

```
1  set⟨Switch⟩ L2s = ...                    // corresponds to deciders of Listing 5.12
2  (set⟨Switch⟩)[L2s] L1_sets = all[∅]
3  (set⟨Switch⟩)[L2s] HN = all[∅]           // corresponds to HN of Listing 5.12

   // on notification from L2 switch
4  Q: when receive ⟨L1s⟩ from s ∈ L2s
5      L1_sets[s] ← L1s
6      HN[s] ← {s}
7      foreach n ∈ {L2s \ {s}} do
8          if L1_sets[n] == L1_sets[s]
9              HN[s] ← HN[s] ∪ {n}
10     foreach n ∈ HN[s] do
11         HN[n] ← HN[s]
```

5.7.3 L_1-Coordinate Assignment: Basic DCP

In this section, we discuss the assignment of L_1-coordinates to ALIAS switches using DCP. We consider two options for L_1-coordinate selection and discuss the tradeoffs associated with each.

Recall that to assign L_1-coordinates in ALIAS, we can simply apply DCP, with L_1 switches as choosers and L_2 switches as deciders. Note that a single L_1 switch may be participating as a chooser with respect to several different sets of shared deciders. That is, chooser c_1 may share deciders d_1 and d_2 with chooser c_2 and deciders d_3 and

d_4 with chooser c_3. In fact, these sets of shared deciders correspond exactly to the L_2 hypernodes in the topology.

There are two options for L_1-coordinate selection in ALIAS. Both satisfy the **Distinctness** property of LSP amongst L_1 switches:

1. **Single L_1-Coordinate**: On one hand, we can assign a single L_1-coordinate c_1 to each L_1 switch. In this case, the set of labels for an L_1 switch s_1 will be of the form $\{(c_{2_1},c_1),...,(c_{2_m},c_1)\}$ where c_{2_1} through c_{2_m} are the L_2-coordinates of each of the m hypernodes to which s_1 is connected.

2. **L_1-Coordinate Per L_2 Hypernode**: Another option is to assign to s_1 multiple L_1-coordinates, one per neighboring L_2 hypernode. Here, s_1's label set will be of the form $\{(c_{2_1},c_{1_1}),...,(c_{2_m},c_{1_m})\}$, and s_1 will have an L_1-coordinate corresponding to each neighboring L_2 hypernode (and therefore each L_2-coordinate c_{2_i}).

There are tradeoffs between these two options. With option (1), we have a simpler protocol; s_1 only needs to select and keep track of one coordinate. However, this scheme may unnecessarily restrict s_1's coordinate choices, forcing the coordinate domain to be larger than necessary. This is because s_1 may compete with every other L_1 switch in the topology for its coordinate, even if it shares a different set of L_2 deciders with each other L_1 switch. Additionally, this scheme may result in extra communication on topology changes. A topology change that introduces a connection between an L_1 switch s_1 and L_2 switch s_2 forces s_1 and all of its neighboring L_2 switches to rerun DCP. This could potentially involve all L_2 switches in the topology, even those outside of s_2's hypernode. Option (2) provides the complement of these tradeoffs; it is more complex to implement, but reduces the required size of the coordinate domain to the largest set of L_1 switches all connected to an L_2 hypernode. Additionally, after a topology change, an L_1 switch only needs to communicate with the L_2 switches in a single hypernode.

We illustrate these tradeoffs in Figure 5.12. Suppose the dotted link is initially not present. In this case, regardless of the option used, each L_1 switch has only a single coordinate, as each only connects to one L_2 hypernode. Because S_5 and S_6 do not share deciders, they are free to have the same coordinate, in this case 7. Initially, S_5 has only

a single label in its set, $\{3.7\}$. Suppose that the dotted link now appears, causing S_5 to share a decider with S_6. Under option (1), S_5 will have to select a new coordinate, and will have to communicate with all neighboring L_2 switches (in this example, all L_2 switches in the topology) to discover that it cannot select 1 or 7. If it selects $x{\neq}1,7$, its new label set becomes $\{3.x, 4.x\}$, and the coordinate domain must include at least 3 choices. On the other hand, with option (2), S_5 only reselects its coordinate with respect to hypernode $\{S_3\}$, and can select a second coordinate that is anything other than 7. S_5 only communicates with S_3 to accomplish this, and its new label set is $\{3.7, 4.x\}$, with $x{\neq}7$, giving an overall coordinate domain size of 2.

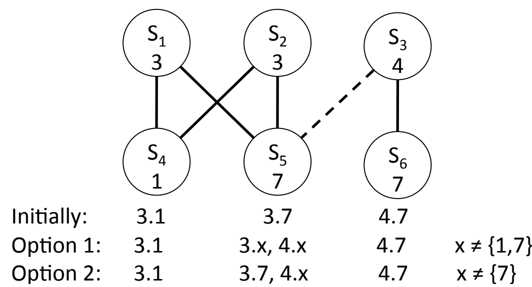


Figure 5.12: Two Options for L_1-Coordinate Selection

We can implement the first option by simply running a single instance of DCP: L_1 switches take the role of choosers and L_2 switches are deciders. This approach uses the exact algorithms of Listings 5.6 through 5.8. However, because of the tradeoffs discussed above, ALIAS adopts the second option for L_1-coordinate selection; it assigns to each L_1 switch s_1, a set of coordinates, one for each of s_1's neighboring L_2 hypernodes. To implement this, we could run multiple simultaneous instances of DCP at each L_1 switch s_1, one instance for each neighboring L_2 hypernode, in separate processes on s_1. However, this can be costly in terms of performance. Additionally, hypernode membership changes may cause complicated interactions between these DCP instances. Instead, we modify the chooser process to keep track of multiple coordinates at once. We perform this refinement in two steps.

In the first step, we introduce the concept of per-hypernode coordinates into the chooser's actions and state. This is shown in Listing 5.11. Rather than storing just the set of neighboring deciders (*c.deciders* of Listing 5.7), a chooser stores the set of neighboring hypernodes in *c.HNs* and a map of hypernodes to their member deciders in

c.deciders. The chooser indexes *c.me* over its set of neighboring hypernodes, and so all instances of *c.me* from Listing 5.7 are replaced with *c.me*[*h*] in Listing 5.11. Note that it is not necessary to index *c.seq* over hypernodes, because the only requirement of *c.seq* is that it increase with each choice; it need not increase by exactly 1.

Listing 5.11: Chooser Algorithm: Actions and State (Multi-Hypernode Refinement 1)

```
1  set⟨HN⟩ HNs
2  (set⟨Switch⟩)[HNs] deciders =...
3  int seq = 0
4  Choice[HNs] me = all[-1]
5  (set⟨Choice⟩)[deciders] hints = all[∅]

   // when needs to make a choice
6  A: when ∃ h ∈ HNs: me[h] == -1
7      choices ← domain(Choice) \ {-1} \ {hints[d]∀ d ∈ deciders[h]}
8      me[h] ← choose from choices
9      seq ++
10     SendTo(seq,me[h],deciders[h])
11     TO_arm

   // retransmit last msg sent to deciders yet to acknowledge
12 B: when timeout
13     dests ← {deciders[h]∀ h ∈ HNs: (me[h] ≠ -1) ∧ (¬HasReceivedAck(h))}
14     ResendTo(dests)
15     TO_arm

   // receive response from d
16 C: when receive ⟨s, chosen, hint⟩ from d
17     choose h ∈ HNs: d ∈ deciders[h]
18     ReceiveAck(s,d)
19     if RecentAck(s,d)
20         hints[d] ← hint
21     if CurrentChoice(s,d) ∧ (chosen == -1)
22         me[h] ← -1
23     if OldChoice(s,d) ∧ (me[h] ≠ -1)
24         SendTo(last_choice[d],me[h],{d})

   // decider d joins HN h and round is active
25 D: when ∃ d ∈ deciders, h ∈ HNs: (d joins deciders[h])∧(me[h] ≠ -1)
26     choose d' ∈ deciders[h]: d' ≠ d
27     hints[d] ← ∅
28     ClearChannel(d)
29     CopyChannel(d,d')
30     SendTo(seq,me[h],{d})
```

The guards and pseudocode for Actions *A*, *B*, and *D* change to incorporate the notion of a hypernode; when a chooser needs to make a choice *for a particular hypern-*

ode, Action A executes, Action B resends to only those *hypernodes* that require retransmission[10], and Action C is updated to determine the hypernode to which the sending decider belongs. When a chooser learns that a new decider has joined a hypernode, Action D executes and uses channel routines *CopyChannel* and *ClearChannel* to enable a new hypernode member to "catch up" with the other members. Here, we define *joins* as the moment when d moves from $deciders[h_1]$ to $deciders[h_2]$, with $h_1 \neq h_2$ and $|h_2| \geq 2$.

The refinement above is intuitive, but not directly implementable, as we have no concrete representation for a hypernode. We address this with our second step in Listing 5.12, by introducing the following representation: To index a variable over a hypernode, we index it over all individual member switches of the hypernode. To read a value of a hypernode (e.g. $c.me[h]$), we read the corresponding value from any decider in the hypernode, and to write a value to a hypernode, we write to all members of the hypernode.

To keep track of neighboring deciders and hypernodes, a chooser c stores the set of neighboring deciders ($c.deciders$) and a map of each decider d to the set of deciders in d's hypernode ($c.HN$). While $c.me$ was indexed over hypernodes in Listing 5.11, it is indexed over all deciders in Listing 5.12. When the value of $c.me$ is to be written for a particular hypernode, it is written for all deciders in that hypernode, and when it is read, it is read from a single member of the hypernode. The guard for Action A, the set of deciders to receive resent messages in Action B, and the operations in Action D are all updated to accommodate these changes. In Action D, we define"joins" as the moment at which d moves from $HN[d_1]$ to $HN[d_2]$, with $d_1 \neq d_2$ and $|HN[d_2]| \geq 2$.

Note that hypernode computation runs simultaneously with this instance of DCP, with L_1s of Listing 5.9, L_2s of Listing 5.10, and HN of Listing 5.10 corresponding to *choosers* (Listing 5.6), and *deciders* and HN (Listing 5.12) respectively. We transition to these variable names in our next refinement. Each L_2 switch belongs to exactly one hypernode and therefore participates in exactly one instance of DCP. So, the code for the decider does not change from that of Listing 5.6 for this refinement. The chooser's channel-related code also remains as in Listing 5.8.

[10]The astute reader may notice that the channel predicate *HasReceivedAck* operates over a hypernode rather than a decider. This temporary inconsistency will be resolved in our next refinement.

Listing 5.12: Chooser Algorithm: Actions and State (Multi-Hypernode Refinement 2)

```
1  set⟨Switch⟩ deciders = ...                   // corresponds to L2s of Listing 5.9
2  (set⟨Switch⟩)[deciders] HN = ...             // corresponds to HN of Listing 5.9
3  int seq = 0
4  Choice[deciders] me = all[-1]
5  (set⟨Choice⟩)[deciders] hints = all[∅]

   // when needs to make a choice
6  A: when ∃ d ∈ deciders: me[d] == -1
7      choices ← domain(Choice) \ {-1} \ {hints[d']∀ d' ∈ HN[d]}
8      ME ← choose from choices
9      foreach d' ∈ HN[d] do
10         me[d'] ← ME
11     seq ++
12     SendTo(seq,ME,HN[d])
13     TO_arm

   // retransmit last msg sent to deciders yet to acknowledge
14 B: when timeout
15     dests ← {d ∈ deciders: (me[d] ≠ -1) ∧ (¬HasReceivedAck(d))}
16     ResendTo(dests)
17     TO_arm

   // receive response from d
18 C: when receive ⟨s, chosen, hint⟩ from d
19     ReceiveAck(s,d)
20     if RecentAck(s,d)
21         hints[d] ← hint
22     if CurrentChoice(s,d) ∧ (chosen == -1)
23         foreach d' ∈ HN[d] do
24             me[d'] ← -1
25     if OldChoice(s,d) ∧ (me[d] ≠ -1)
26         SendTo(last_choice[d],me[d],{d})

   // decider d joins d′'s HN and round is active
27 D: when ∃ d, d' ∈ deciders: (d joins HN[d']) ∧ (me[d'] ≠ -1)
28     me[d] ← me[d']
29     hints[d] ← ∅
30     ClearChannel(d)
31     CopyChannel(d,d')
32     SendTo(seq,me[d],{d})
```

5.7.4 L_2-coordinate Assignment: Distributed DCP

We next discuss the assignment of L_2-coordinates to L_2 hypernodes. We use the extension of DCP introduced in Section 5.6.1 to allow each L_2 hypernode to function as a distributed chooser, with neighboring L_3 switches as deciders. However, before giving the refinement for this extension, we first consider the necessity of a distributed chooser for L_2-coordinate selection.

A tempting approach is to use one instance of DCP in which L_3 switches are deciders and a single L_2 switch from each hypernode is a chooser. However, this does not work. For example, refer to the network in Figure 5.8 (Section 5.6). There are three hypernodes: $\{S_3\}$, $\{S_4,S_5\}$, and $\{S_6\}$. The L_2-coordinate shared by S_4 and S_5 must be distinct from that of S_3 and that of S_6. Thus, whatever implements the chooser for the hypernode $\{S_4,S_5\}$ needs to communicate with the deciders at S_1 and at S_2. Neither S_4 nor S_5 is connected to both deciders, and so S_4 and S_5 must together implement a chooser for their hypernode.

Given that we need the cooperation of all L_2 switches in a hypernode, we apply the extension of DCP introduced in Section 5.6.1 for L_2-coordinate selection. Recall that this extension distributes a chooser $\mathscr{C}$ into a set, $Relays(\mathscr{C})$, of processes that all share a common coordinate as well as a single process, $Repr(\mathscr{C})$, that performs the choosers actions. Listings 5.13 and 5.14 contain the chooser's actions and state for $Repr(\mathscr{C})$ and $Relays(\mathscr{C})$, respectively.

As shown in Listing 5.13 a chooser's representative maintains the set of L_2 switches to which it connects ($c.L_2relays$), the hypernode membership of each neighboring L_2 switch ($c.HN$), and the L_3 deciders to which each neighboring L_2 switch connects ($c.deciders$). Since it will compute a value of $c.me$ to be shared by an entire hypernode, a representative needs to index $c.me$ over the set of neighboring hypernodes (in case it represents multiple hypernodes). As in our previous refinement, we index over hypernodes by writing a value for a hypernode to all of its L_2 members and by reading a hypernode's value via any of its L_2 members. Therefore, $c.me$ is indexed over the representative's neighboring L_2 switches. The $c.hints$ variable is index similarly.

Action A is triggered by a hypernode with a null value for $c.me$ (indicated by an L_2 switch with a null value). The representative collects all hints for this hypernode,

Listing 5.13: Chooser Algorithm: Actions and State (Distributed Chooser, Representative L_1 Switch)

```
1  set⟨Switch⟩ L2relays
2  (set⟨Switch⟩)[L2relays] HN = ...
3  (set⟨Switch⟩)[L2relays] deciders = ...
4  int seq = 0
5  Choice[L2relays] me = all[-1]
6  ((set⟨Choice⟩)[L2relays] hints = all[∅]

   // when needs to make a choice
7  A: when ∃ l2 ∈ L2relays: me[l2] == -1
8       choices ← domain(Choice) \ {-1} \ {hints[l2']∀ l2' ∈ HN[l2]}
9       ME ← choose from choices
10      foreach l2' ∈ HN[l2] do
11           me[l2'] ← ME
12      seq ++
13      dests ← {d ∈ deciders[l2']∀ l2' ∈ HN[l2]}
14      SendTo(seq,ME,l2,dests)
15      TO_arm

   // retransmit last message sent to deciders yet to acknowledge
16 B: when timeout
17      foreach l2 ∈ L2relays: me[l2] ≠ -1 do
18           dests ← {d ∈ deciders[l2]: ¬HasReceivedAck(d,l2)}
19           ResendTo(dests,l2)
20      TO_arm

   // receive response from d
21 C: when ¬Receive.empty()
22      [s,chosen,hint,rep_l1,d,l2] ← Receive.removeHead()
23      ReceiveAck(s,d,l2)
24      if RecentAck(s,d,l2)
25           hints[l2] ← hint
26      if CurrentChoice(s,d,l2) ∧ (chosen == -1)
27           foreach l2' ∈ HN[l2] do
28                me[l2'] ← -1
29      if OldChoice(s,d,l2) ∧ (me[l2] ≠ -1)
30           SendTo(last_choice[d][l2],me[l2],l2,{d})
```

selects a new choice for the hypernode, and writes this choice to all of the hypernode's L_2 members. As in previous version of the protocol, it then updates its sequence number, determines the deciders that neighbor this hypernode, and sends its choice to the deciders via the appropriate relays.[11] Action B differs slightly from previous version of the protocol, in that it checks for whether a hypernode has made a choice in a for loop rather than in the Action's guard. This is so the chooser can resend on behalf of all necessary hypernodes in one execution of Action B, rather than only resending for a single hypernode when the timer fires. Action C is triggered by a non-empty *Receive* queue rather than by direct receipt of a message from a decider. The representative does not run its own copy of Action D, rather all L_1 switches run Action D as discussed below.

We next consider the L_2 relays of the distributed chooser, as shown in Listing 5.14. This listing introduces the two chooser Actions S and R that partially implement the *Send* and *Receive* queues between the chooser's relays and representative. When a representative sends its choice to a decider, it includes the sequence number, the choice itself, the current hypernode's members for which it is choosing, its own identity, and the decider for which the message is intended. The third and fourth arguments are new in this refinement and are used at the decider for book-keeping. In Action S, an L_2 switch passes the first four parameters to the appropriate decider. When a decider responds to a representative's choice, it includes the sequence number, the choice (null if the message is a rejection), a set of hints, and the representative L_1 switch for which the message is intended. An L_2 relay adds the decider's and its own identities and enqueues a message on the *Receive* queue for retrieval by the representative via Action C.

Recall from Section 5.6 that all L_1 switches, including non-representatives, execute a version of of Action D, as shown in in Listing 5.15. Action D captures situations in which an L_1 switch l_1 newly represents an L_2 relay l_2, either because l_1 has just become a chooser $\mathscr{C}$'s representative or because l_2 has just joined $Relays(\mathscr{C})$. Via Action D, the representative resets and copies the associated state, and then resends choices to deciders (via relays) as necessary. Non-representative L_1 switches also maintain and read *Receive* queues for neighboring hypernodes, in Action C'. This is so they have current channel capacity information should they become a representative in the future.

[11]The representative includes the L_2 switch that triggered this action as an argument for the *SendTo* channel routine, so that the routine can determine the appropriate set of relays for the message.

Listing 5.14: Chooser Algorithm: Actions and State
(Distributed Chooser, L_2 Relays)

```
1  Switch myID
   // when data to send
2  S: when ¬Send.empty()
3      [s,x,hn,rep_l1,d] = Send.removeHead()
4      send ⟨s,x,hn,rep_l1⟩ to d
   // when data to receive
5  R: when receive ⟨s,chosen,hint,rep_l1⟩ from d
6      Receive.append([s,chosen,hint,rep_l1,d,myID])
```

Listing 5.15: Chooser Algorithm: Actions and State
(Distributed Chooser, All L_1 Switches)

```
   // receive response from d
1  C': when ¬Receive.empty()
2      [s,chosen,hint,l1rep,d,l2] ← Receive.removeHead()
3      if ¬(AmRepL1(l2))
4          ReceiveAck(s,d,l2)
   // when AmRepL1(l2) changes or l2's HN changes
5  D: when ∃ l2 ∈ L2relays: AmRepL1(l2) becomes true ∨
6             ∃ l2, l2' ∈ L2relays: (l2 joins HN[l2']) ∧ (me[l2'] ≠ -1) ∧(AmRep(l2'))
7      me[l2] ← -1
8      hints[l2] ← ∅
9      ClearChannel(l2)
10     if ∃ l2' ∈ L2relays: (l2 joins HN[l2']) ∧ (me[l2'] ≠ -1) ∧ (AmRep(l2'))
11         CopyChannel(l2,l2')
12         seq++
13         dests ← {d ∈ deciders[l2']∀ l2' ∈ HN[l2]}
14         SendTo(seq,me[l2],l2,dests)
```

The remainder of the changes to a chooser are in its channel routines and predicates, as shown in Listing 5.16 and 5.17.[12] Since a relay provides a virtual channel to a decider from a representative, the representative indexes all channel variables over the entire virtual channel, decider and relay. This affects all channel-related variables (*sent*, *last_sent*, *last_ack*, and *last_choice*) and the channel-bounding predicates.

The channel code houses the new *Send* and *Receive* queues, and the *SendTo* and *ResendTo* routines append to the *Send* queue rather than sending a message directly to a decider as in previous versions of the protocol. Note that the *SendTo* and *ResendTo* routines enqueue a message intended for a decider *d* onto the *Send* queue of every L_2

[12]The code is separated into two listings due to space constraints.

switch that reaches d. As discussed in Section 5.6, a distributed chooser's representative has the option to send a message to a decider d via:

1. every L_2 switch that it neighbors, letting the L_2 switches filter unroutable messages

2. all of its neighboring L_2 switches that reach d, possibly sending the choice to d via multiple relays

3. a subset of its neighboring L_2 switches that reach d, possibly sending the choice to d via multiple relays

4. only one of its neighboringL_2 switch that reaches d.

These options have tradeoffs between synchronization complexity and message load; we favor option (2) as a middle ground.

Finally, the *ClearChannel* function becomes more complicated, as a result of the fact that we represent a hypernode with its constituent L_2 members. Because of this representation, the channel bounding variables may include entries for decider-L_2 switch pairs (d, l_2) for which l_2 is not connected to d, but there is some l_2' in l_2's hypernode that is connected to d. If l_2' leaves the hypernode containing l_2, then any (d, l_2) values need to be removed.

Listing 5.16: Chooser Channel Predicates and Routines, Part 1 (Bounded Channels, Distributed Chooser)

```
1  int[deciders][L2relays] last_ack = all[0]
2  (set⟨int⟩)[deciders][L2relays] sent = all[∅]
3  ⟨int,Choice⟩[deciders][L2relays] last_sent = all [⟨0,-1⟩]
4  int[deciders][L2relays] last_choice = all[0]
5  int max_in_chan = a non-zero constant

6  queue[L2relays] Send
7  queue[L2relays] Receive

   // ⟺ c has an ack from d via l2 for its latest choice
8  boolean HasReceivedAck (d,l2):
9      last_ack[d][l2] == last_choice[d][l2]

   // ⟺ s acknowledges c's most recent choice to d via l2
10 boolean CurrentChoice (s,d,l2):
11     s == last_choice[d][l2]

   // ⟺ s acknowledges an obsolete choice sent to d via l2
12 boolean OldChoice (s,d,l2):
13     s < last_choice[d][l2]

   // ⟺ there is room in the channel to send to d via l2
14 boolean CanSendTo (d,l2):
15     | sent[d][l2] | < max_in_channel

   // ⟺ c has sent its most recent choice to d via l2
16 boolean SentLatest (d,l2):
17     last_sent[d][l2][0] == last_choice[d][l2]

   // ⟺ s acknowledges c's most recent message to d via l2
18 boolean RecentAck (s,d,l2):
19     s == last_sent[d][l2][0]
```

Listing 5.17: Chooser Channel Predicates and Routines, Part 2 (Bounded Channels, Distributed Chooser)

```
1  SendTo (s,x,l2,D):
2      foreach d ∈ D do
3          if CanSendTo(d,l2)
4              foreach l2' ∈ HN[l2]: d ∈ deciders[l2'] do
5                  Send[l2'].append([s,x, HN[l2],myID,d])
6              foreach l2' ∈ HN[l2] do
7                  sent[d][l2'] ← sent[d][l2'] ∪ {s}
8                  last_sent[d][l2'] ← (s,x)
9          foreach l2' ∈ HN[l2] do
10             last_choice[d][l2'] ← s
11 ResendTo (D,l2):
12     foreach d ∈ D do
13         if |sent[d][l2]|>0
14             foreach l2' ∈ HN[l2]: d ∈ deciders[l2'] do
15                 Send[l2'].append([last_sent[d][l2'],HN[l2],myID,d])
16 ReceiveAck (s,d,l2):
17     foreach l2' ∈ HN[l2] do
18         sent[d][l2'] ← sent[d][l2'] \ {i: i≤s}
19         last_ack[d][l2'] ← s
20 ClearChannel (l2):
21     foreach d ∈ deciders[l2] do
22         last_ack[d][l2] ← 0
23         last_choice[d][l2] ← 0
24     foreach l2' ∈ L2relays, d ∈ deciders do
25         connects_to_d ← {l2'' ∈ HN[l2']: d ∈ deciders[l2'']}
26         if connects_to_d == ∅
27             last_sent.erase(d,l2')
28             last_choice.erase(d,l2')
29             last_ack.erase(d,l2')
30             sent.erase(d,l2')
31 CopyChannel (l2,ref,D):
32     foreach d ∈ D do
33         last_choice[d][l2] ← last_choice[d][ref]
```

For L_2-coordinate assignment, the decider becomes more complex as well. A decider keeps a record of all L_2 and L_1 switches it has seen ($d.L_2relays$ and $d.L_1reps$). It indexes the choosers that it has seen over $d.L_2relays$ and $d.L_1reps$, representing a chooser via its constituent L_2 members (*d.chooser*). Finally, the decider indexes its choice variables (*chosen*, and *last_seq*) over entire choosers, L_2 relays and L_1 representatives. This is necessary because the representative switch for a hypernode can change. Thus, deciders may maintain duplicate information for a hypernode, namely information obtained from two different switches claiming to represent that hypernode. Recall from Section 5.7.2 that an L_2 switch sends its current set of neighboring L_1 switches to L_3 switches when this set changes. As such, a decider d always knows the most recent set of L_1 switches to which a neighboring L_2 is connected, and d can compute the current representative switch for the hypernode and select the appropriate value of *d.chosen* to pass to an overlying communication protocol. Deciders employ a similar representation for hypernodes as do choosers; they simply index over hypernodes by indexing over the hypernodes' member switches (as shown in Action G).

A decider may be connected to a chooser via multiple L_2 switches, and thus needs to make a decision on whether to accept a value received via an L_2 switch based on the hypernode of the L_2 switch. This adds a small amount of complexity to the decider's Action G; A decider compares a requested value x to those held by L_2 switches in all other hypernodes, regardless of the representative switches for those hypernodes. As such, a decider compares x to $chosen[l'_2][l'_1]$ for any value of l'_1. Listing 5.18 shows the modified decider code.

5.7.5 Derivation Summary

This completes the protocol derivation from the basic DCP to a solution for coordinate selection in ALIAS. L_1 switches function as choosers for L_1-coordinates (Listings 5.8 and 5.12), as potential representatives for L_2-coordinate selection (Listings 5.13, 5.15, 5.16, and 5.17) and as hypernode calculators (Listing 5.10). L_2 switches act as relays for L_2-coordinate selection (Listing 5.14), as deciders for L_1-coordinate selection (Listing 5.6) and as hypernode change notifiers (Listing 5.9). Finally L_3 switches are deciders for L_2-coordinate selection (Listing 5.18).

Listing 5.18: Decider Algorithm (Distributed Chooser)

```
1  set⟨Switch⟩ L2relays = ...
2  set⟨Switch⟩ L1reps = ...
3  (set⟨Switch⟩)[L2relays][L1reps] choosers = ...
4  Choice[L2relays][L1reps] chosen = all[-1]
5  int[L2relays][L1reps] last_seq = all[0]

   // when connected to new L2 switch
6  F: when new l2 ∈ L2relays with representative l1
7      L2relays ← L2relays ∪ {l2}
8      L1reps ← L1reps ∪ {l1}
9      choosers[l2][l1] ← {l2}
10     chosen[l2][l1] ← -1
11     last_seq[l2][l1] ← 0

   // respond to a message from L2 switch l2
12 G: when receive ⟨s,x,hn,l1⟩ from l2
13     L1reps ← L1reps ∪ {l1}
14     if s ≥last_seq[l2][l1]
15         foreach l2' ∈ choosers[l2][l1] do
16             choosers[l2'][l1] ← hn
17             last_seq[l2'][l1] ← s
18         if ∃ l2'∈ L2relays, l1'∈ L1reps: (l2'∉ choosers[l2][l1]) ∧ (chosen[l2'][l1'] == x)
19             foreach l2' ∈ choosers[l2][l1] do
20                 chosen[l2'][l1] ← -1
21         else
22             foreach l2' ∈ choosers[l2][l1] do
23                 chosen[l2'][l1] ← x
24     hints ← {chosen[l2'][l1']∀ l1' ∈ L1reps, l2' ∈ (L2relays \ choosers[l2][l1])}
25     hints ← hints \ {-1}
26     send ⟨s,chosen[l2][l1],hints,l1⟩ to l2
```

5.8 The Decider/Chooser Protocol in Wireless Networks

In this section, we describe another example of label assignment based on shared connectivity. This case arises in the context of assigning IP addresses to wireless devices. We offer this example to illustrate a plausible use of DCP outside of the context of data center networking.

A local wireless network, e.g., within a building or a corporation, consists of a set of fixed wireless access points and mobile devices that move around within the network. At any time, a mobile device may be within range of (and may use the same channel as) several access points (APs), but it associates with a single access point at a time. A *handoff* occurs when the device changes its association from one AP to another. If, as a result of handoff, the device needs to acquire a new IP address, then ongoing communication sessions can be disrupted.

There are different ways to avoid this need for a new IP address. For example, the set of access points in a network may utilize a wired distribution system to synchronize with each other, ensuring that an IP address given to a device by AP_1 is permittable for use with AP_2 as well. Or, the APs in a network may communicate with a central server responsible for ensuring IP address uniqueness among all network devices. Managing centralized state or requiring a separate distribution system between APs places a significant additional management burden on the network operator.

A key difficulty of address assignment in this type of network is the dynamism of the network; the set of mobile devices varies over time, as does the set of access points visible to each mobile device. In fact, we learned from speaking with network operators that the issues of changing sets of devices and difficulty with handoff are significant pain points for some types of wireless networks.

This dynamism suggests a solution using the Decider/Chooser abstraction. In wireless networks, we run an instance of DCP with mobile devices as choosers and access points as deciders, wherein a link between device *md* and AP *ap* indicates that *md* is within range of *ap*, as shown in Figure 5.13. A mobile device selects an IP address that is acceptable with respect to all APs within range, i.e. all of its deciders. As a device moves throughout the network, its set of deciders change, and if at any time it finds its IP address to be in conflict (as reported by one of its deciders) it reselects. This

application of DCP has the benefit of removing the requirement of a central authority or separate wired distribution system between APs, but without the need for IP address reassignment on every handoff.

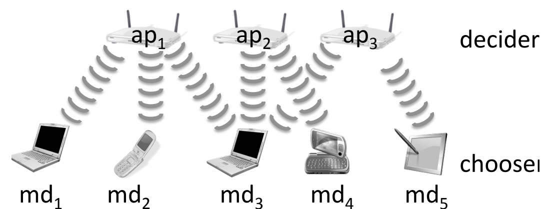


Figure 5.13: Multiple AP Example

5.9 Related Work

Our solution uses a Las Vegas type randomized algorithm: the labels that are computed always satisfy the problem specification, but the algorithm is only probabilistically fast. It is also a fully dynamic algorithm [33], in that it makes use of previous solutions to solve the problem more quickly than by recomputing from scratch.

Assigning labels to nodes is not a new problem. For example, in [25] the authors consider the issues of assigning labels to nodes in an anonymous network of unknown size. The quality of an assignment algorithm depends on the size of the label domain and the algorithm's efficiency is based on the convergence time and message load. The authors' approach uses a special *source* node (the sole source of asymmetry) to root a spanning tree of the anonymous network, and explores the cost of propagating enough information to label all nodes. We consider networks with significant symmetry: each network can be partitioned into bipartite graphs of processes, even if a process may be made up of multiple nodes. This symmetry and the use of randomization allows us to devise an algorithm in which nodes only communicate with immediate neighbors. This reduces the overall message load relative to that of a network with only a single designated node.

Our solution can also be considered an instance of the *renaming* problem [8, 9, 15] in which a set of processes, each with a unique name chosen from some large name

space, together assign themselves unique names from a smaller name space. The protocol in [15]–which is for a shared memory model–has a similar structure to DCP with a single decider process: our decider has a role similar to a shared atomic snapshot object in their protocol. Their protocol differs in that they sought a deterministic solution; DCP can rename into a smaller name space because it is randomized. Also, LSP differs from the renaming problem: in LSP , two processes can assign themselves the same (shorter) name if they don't share a decider.

Finally, the Label Selection Problem also relates to the graph coloring problem (GCP). In fact, GCP is reducible to LSP. The mapping from GCP to LSP is quite simple; vertices in an instance of GCP, $G = (V,E)$, correspond in a one-to-one mapping to choosers in LSP, and for any pair of vertices in G that are connected by an edge in E we create a decider d and connect each of the corresponding choosers to d. In this way, pairs of vertices that require different colors in GCP correspond to pairs of choosers that require distinct coordinates in LSP. The mapping from LSP to GCP is equally simple. Even though LSP can be mapped to GCP, the LSP structure arises naturally in many protocol problems—like those given in this chapter—and the separation of processes into choosers and deciders has helped us to refine DCP for more practical application. However, some techniques for graph coloring could be applied to LSP; for instance one could apply the multi-trials technique introduced by Schneider and Wattenhofer [64] to LSP.

5.10 Summary

We present, in this chapter, a theoretical analysis of the basic building block of ALIAS. We first formalize a sub-problem of ALIAS, the Label Selection Problem (LSP). We then provide the Decider/Chooser Protocol (DCP) as a solution to this problem. Through model checking and proofs, we show that DCP satisfies the requirements of LSP. We use mathematical analysis and simulations to show that DCP converges quickly under the expected conditions. Finally, we apply DCP to ALIAS, using protocol refinements to extend DCP to solve the more complicated issues in ALIAS.

This analysis allows us to formally reason about the correctness of ALIAS, the

interactions among ALIAS components, and the interactions between ALIAS and other data center network protocols. This is a crucial step in ensuring that data center network protocols are correct as well as feasible to deploy, configure and debug.

5.11 Acknowledgment

Chapter 5, in part, contains material as it appears in the Proceedings of the 25th International Symposium on Distributed Computing (DISC) 2011. Walraed-Sullivan, Meg; Niranjan Mysore, Radhika; Tewari, Malveeka; Zhang, Ying; Marzullo, Keith; Vahdat, Amin. The dissertation author was the primary investigator and author of this paper.

Chapter 5, in part, contains material submitted for publication as "A Randomized Algorithm for Label Assignment in Dynamic Networks." Walraed-Sullivan, Meg; Niranjan Mysore, Radhika; Marzullo, Keith; Vahdat, Amin. The dissertation author was the primary investigator and author of this paper.

Chapter 6

Single Label Selection for ALIAS Hosts

As detailed in Chapter 4, ALIAS hosts have multiple labels. On one hand, clever use of these labels can enable interesting load-balancing, task separation and multi-path techniques. However, on the other hand, such multiple labels could prove to be a limitation of the protocol, as they make the interface of ALIAS significantly different from those of the protocols it aims to replace. In this chapter, we explore a technique for selecting and using only a single label per host for routing and forwarding in ALIAS.

We begin with a short review of the relevant ALIAS details. We then consider how to select a single, optimal[1] label from an ALIAS host's set of labels and how to use this label exclusively, in ALIAS for routing and forwarding. Selecting a single label for forwarding affects both multi-path support and peer link usage in ALIAS. To mitigate these effects, we introduce a forwarding concept that we coin a *super table*. The super table is stored in software and contains all forwarding information known to a switch. It is used along with local policy to populate a switch's hardware forwarding table with a subset of the super table entries. We then perform simulations to measure the sizes of the resulting forwarding tables for a number of sample topologies. We find that the selection of a single label in ALIAS leads to an explosion in forwarding state, the very property that ALIAS seeks to reduce. As such, we conclude that the benefits of choosing a single label for each ALIAS host are outweighed by the associated costs.

[1] The definition of "optimal" will vary based on the requirements for any given network.

6.1 Background and Environment

ALIAS operates over the multi-rooted tree topologies that underlie many data center networks today [4, 13, 18, 28, 47, 56]. Figure 6.1 shows a sample multi-rooted tree topology: a fat tree [19, 47] made up of 4-port switches. As shown in the figure, ALIAS organizes its input trees into levels, with hosts at Level L_0 and switches at levels L_1 through L_n, from the bottom of the tree upwards. Switches are grouped into hypernodes, wherein a hypernode is defined as a maximal set of switches at one level such that each member switch connects to the same set of hypernodes below. Each L_1 switch forms its own hypernode, and switches at the topmost level of the tree are not grouped into hypernodes. With the exception of L_1 switches, we denote a hypernode in the following figures by physically grouping its constituent switches together.

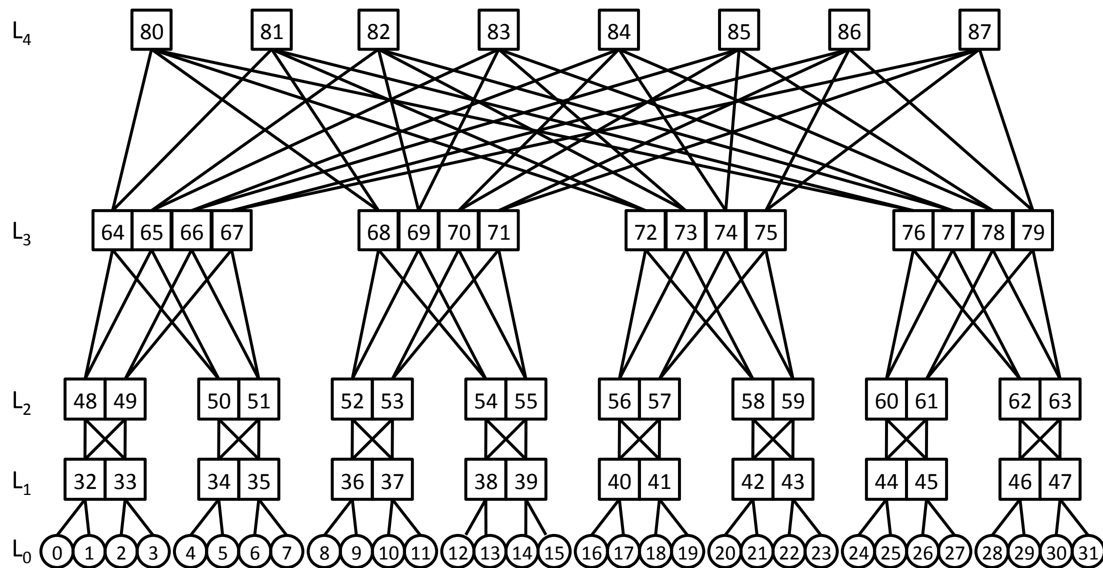


Figure 6.1: Fat Tree Topology

ALIAS assigns to each hypernode a set of coordinates to be shared among its member switches; each L_i switch has one coordinate per neighboring L_{i+1} hypernode. Coordinates are concatenated from the root of the tree downward to form switch and host labels. Since there may be paths through multiple different sets of hypernodes to any given switch (host), a switch (host) in ALIAS has multiple labels.

In Figure 6.1, the numbers marked on each switch and host indicate the nodes' unique identifiers (UIDs).[2] We refer to a switch with marked in the figures with UID x as S_x in the exposition. In ALIAS, a hypernode's coordinates are chosen based on a distributed, randomized algorithm. However, for the purpose of this chapter, we introduce the simplifying convention that a hypernode's coordinate is based on its member switches' UIDs as well as on the physical location of the hypernode and its member switches in our figures as follows: A hypernode *hn*'s coordinate corresponding to the leftmost neighboring hypernode above is the UID of *hn*'s leftmost switch member. For the upper-level neighboring hypernode second from the left, *hn*'s coordinate is the UID of the member switch second from the left. This process continues as we move to the right among *hn*'s neighboring upper-level hypernodes. We clarify this via example in Figure 6.2.

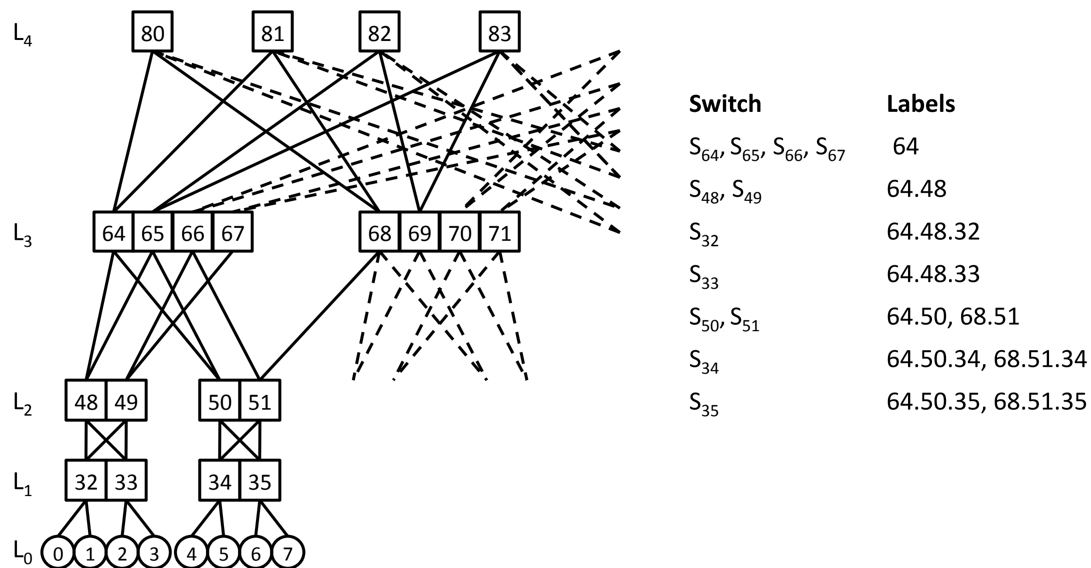


Figure 6.2: Labeling Conventions
(Dashed links are for switches not shown.)

The figure shows a subset of a topology with four levels of switches. L_4 switches are at the top level of the topology and are therefore not grouped into hypernodes. Since there are no hypernodes at L_4, each L_3 hypernode has only a single coordinate. Our convention is to use the UID of the leftmost member switch as the hypernode's coordinate.

[2]In an implementation of ALIAS a UID might be, for instance, a MAC address.

So the coordinate for hypernode $\{S_{64},S_{65},S_{66},S_{67}\}$ is 64 and the coordinate for hypernode $\{S_{68},S_{69},S_{70},S_{71}\}$ is 68. Since L_2 hypernode $\{S_{48},S_{49}\}$ only neighbors a single L_3 hypernode, it has a single coordinate, 48. So, hypernode $\{S_{48},S_{49}\}$ and switches S_{48} S_{49} have one label each, namely 64.48. S witches S_{32} and S_{33} are each in their own hypernodes, and each neighbor one L_2 hypernode, so their labels are 64.48.32 and 64.48.33, respectively. On the other hand, hypernode $\{S_{50},S_{51}\}$ neighbors two L_3 hypernodes, and therefore needs a coordinate to correspond to each. It uses the UID of its leftmost member switch, S_{50} as the coordinate corresponding to 64 and the UID if its second-to-leftmost switch, S_{51} as the coordinate corresponding to 68. Therefore, switches S_{50} and S_{51} each have two labels, {64.50,68.51}. Similarly, the label sets for S_{34} and S_{35} are {64.50.34,68.51.34} and {64.50.35,68.51.35}, respectively.

In ALIAS, a forwarding table entry (FTE) consists of a label or label prefix and a next hop for all packets destined to that prefix or label. FTEs can also include $*$ values, indicating a match for anything not matched by a more specific entry. FTEs may consist of multiple next hops, for use with multi-path forwarding protocols. In Figure 6.1, S_{48} might have the following forwarding entries: 64.48.32$\rightarrow S_{32}$, 64.48.33$\rightarrow S_{33}$, 64.$*\rightarrow\{S_{64},S_{65}\}$, 68.$*\rightarrow\{S_{64},S_{65}\}$, 72.$*\rightarrow\{S_{64},S_{65}\}$ and 76.$*\rightarrow\{S_{64},S_{65}\}$. In general, an ALIAS switch has two types of non-peer link forwarding entries:

- *Downward Entries* that match a label that is one coordinate longer than one of the switch's labels to a downward neighbor of that switch, e.g. 64.48.32$\rightarrow S_{32}$
- *Upward Entries* that match a single L_{n-1} coordinate to a subset of the switch's upward neighbors, e.g. 76.$*\rightarrow\{S_{64},S_{65}\}$

ALIAS accommodates *peer links* between switches at the same level. Peer links can be used, for instance, to create shortcuts between switches that communicate frequently or to connect top-level (L_n) switches directly to one another. The use of peer links leads to forwarding entries with variable length labels; these entries are handled as special cases.

6.2 ALIAS Protocol Modifications

Selecting a single, optimal label for each ALIAS host and enabling exclusive use of that label within ALIAS routing and forwarding requires changes throughout the label assignment and communication portions of the ALIAS protocol. In this section, we first consider one of the many possible definitions for an "optimal" label ℓ for a host h, and we present a protocol to select labels that match this definition. We then provide a protocol to propagate the selected label throughout the tree, so that nodes that previously reached h via labels other than ℓ can now reach h via ℓ.

Throughout this section, we use the example topology shown in Figure 6.3. This topology is nearly a perfect 4-level, 4-port fat tree, with the exception that switches S_{64} and S_{76} have been disconnected from switches S_{50} and S_{62}, respectively, moving S_{64} and S_{76} into their own hypernodes. Because of this, switches S_{48} and S_{49} and their descendants each have two labels, one corresponding to each L_3 hypernode that neighbors L_2 hypernode $\{S_{48}, S_{49}\}$. Given our convention for using hypernode members' UIDs to form labels, S_{32} has the label set $\{64.48.32, 65.49.32\}$.

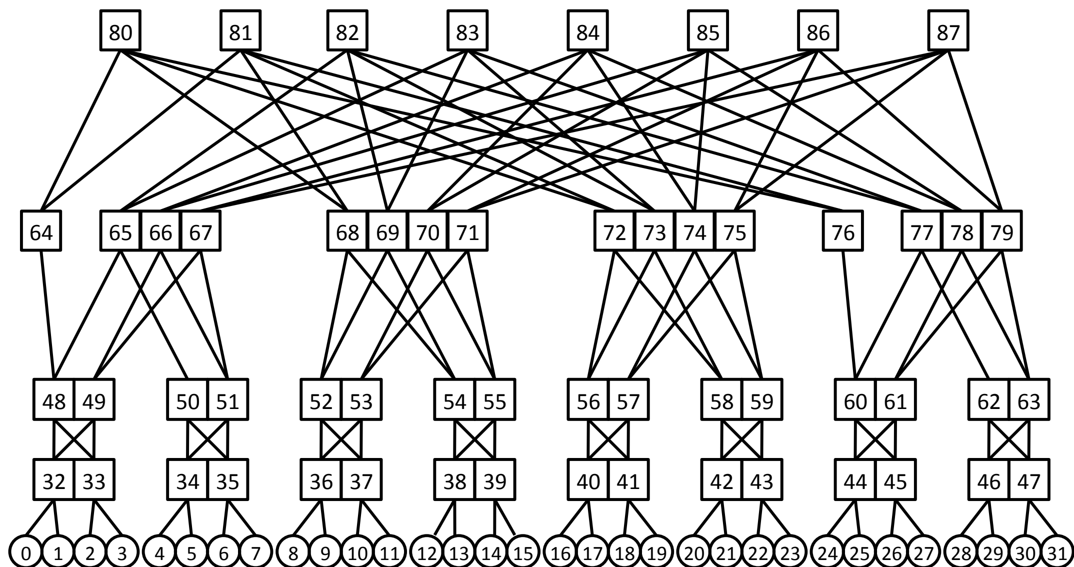


Figure 6.3: Topology with Multiple Labels per Host

6.2.1 Selecting a Single Label

There are a number of methods for selecting the "best" label for a given host, h. To optimize for multi-path support, one might select the label ℓ corresponding to the largest number of paths to h from the top level, L_n, of the network. In this way, if we restrict access to h to only those paths using ℓ, we still retain a large percentage of paths from L_n switches to h. Or, to prioritize reachability, one might take into consideration both the number of downward paths to h via ℓ from each L_n switch as well as the number of other hosts reachable by such L_n switches; this has the effect of counting the total number of hosts that reach h via ℓ. To prioritize availability, one might perform the above calculations in terms of link-disjoint paths, in order to minimize the disruption that a link failure would have on h's availability via ℓ. In the current implementation, we prioritize reachability while retaining as much multi-path support as possible. However, this is just one of many possibilities; a network designer could certainly define "optimal" to suit the requirements for a particular situation.

To determine a host's optimal label according to the metrics above, we begin by calculating an *L_n-value (LNV)* for each L_n switch s_n. We then compute the number of paths from s_n to each individual L_1 switch label ℓ in the network. Finally, we compute ℓ's overall *connectivity-value (CV)* by combining the LNVs of each of its reachable L_n switches with the number of paths to ℓ from that L_n switch. An L_1 switch can then select the label with the highest CV from its label set , and can use this to create a single label for each of its neighboring hosts.

We first calculate an LNV for each L_n switch s_n by counting the number of hosts reachable by s_n. This adds no message complexity and an insignificant amount of computation overhead when injected into the ALIAS protocol; for ALIAS address resolution, L_n switches store a mapping of UID-to-ALIAS label for each reachable host. In order to determine its own LNV, an L_n switch simply counts the number of hosts represented in these mappings. In Figure 6.3, all L_n switches except S_{80} and S_{81} reach all hosts and therefore have LNVs of 32. L_n switches S_{80} and S_{81} do not reach hosts H_4-H_7 or H_{28}-H_{31} and therefore have LNVs of 24.

We next calculate the number of paths from L_n switches to each of an L_1 switch's (and therefore its neighboring hosts') labels. We accomplish this as follows; each L_{n-1}

switch s_{n-1} stores for each neighboring L_n switch s_n a tuple $(\ell, s_n.links, s_n.id, s_n.lnv)$, where:

- ℓ is s_{n-1}'s label (in this case, a single coordinate corresponding to the UID of the leftmost switch in s_{n-1}'s hypernode),
- $s_n.links$ is a count of the links from s_n to s_{n-1}[3],
- $s_n.id$ is s_n's UID and
- $s_n.lnv$ is s_n's LNV, as calculated above.

L_{n-1} switches pass these tuples to L_{n-2} switches as part of their periodically exchanged Topology View Messages (TVMs).

Each L_{n-2} switch s_{n-2} selects a coordinate per neighboring L_{n-1} hypernode. For each member s_{n-1} of a neighboring L_{n-1} hypernode, s_{n-2} creates a label for itself by concatenating its own coordinate (with respect to s_{n-1}'s hypernode) to s_{n-1}'s coordinate. s_{n-2} then multiplies the number of links it has to s_{n-1} by the $s_n.links$ value in each per-L_n switch tuple it has received from s_{n-1}, in order to count the number of paths from itself to s_n via s_{n-1}. Finally, s_{n-2} sums path counts to each L_n switch across all of s_{n-1}'s hypernode members to which it is directly connected, in order to create a set of tuples of the form $(\ell, s_n.links, s_n.id, s_n.lnv)$ for each upper neighboring (L_{n-1}) hypernode. In these tuples, ℓ is s_{n-2}'s label with respect to a particular L_{n-1} hypernode, $s_n.links$ gives the number of paths from s_{n-2} to an L_n switch s_n via members of this hypernode, and as before, $s_n.id$ and $s_n.lnv$ carry s_n's identity and LNV, respectively.

This process continues moving down the tree, so that each L_1 switch s_1 can eventually compute the CV of each of its labels based on the number of paths from each L_n switch to s_1 via that label. Including these operations in the ALIAS protocol adds little complexity and overhead. Labels are passed downward in ALIAS TVMs in order to support restriction of peer-link coordinates (see Chapter 4 for details) and so the per-L_n switch ID, LNV, and path count fields in these new tuples for each label are the only additions to outgoing TVMs.

[3]If parallel links between two switches are not allowed, this will always be 1 at level L_{n-1}.

Finally, for each L_1 label ℓ, we combine reachable L_n switches' LNVs with the counts of paths from each L_n switch to ℓ, and sum across all L_n switches in order to determine an overall CV for each label. Currently, we use the following formula to calculate a label's CV:

$$\sum_{s_n \in L_n switches} LNV(s_n) \times pathsFrom(s_n)$$

.

In Figure 6.3, the only L_1 switches with multiple labels are S_{32}, S_{33}, S_{44} and S_{45}, with label sets {64.48.32,65.49.32}, {64.48.33,65.49.33}, {76.60.44,77.61.44} and {76.60.45,77.61.45}, respectively. There is a single path from each of L_n switches S_{80} and S_{81} to S_{32}, and these L_n switches have an LNV of 24, giving label 64.48.32 a CV of 48. Similarly, there is one path from each of L_n switches S_{82} through S_{87} to S_{32}, and these six L_n switches each have an LNV of 32. So label 65.49.32 has a CV of $6 \times 32 = 196$, and it is the selected label for S_{32}. The cases are similar for S_{33}, S_{44} and S_{45}.

We refer to the label chosen by the protocol above as the "optimal label."

6.2.2 Propagating Single Label Selections

Since a host h's label corresponds to a particular set of paths from the top level of the network to h, it is necessary to update the forwarding tables of switches along different paths, if we allow only one of h's labels to be used for routing and forwarding. In the current ALIAS implementation, L_1 switches pass mappings of UID-to-ALIAS label up towards L_n switches to support efficient address resolution. We leverage the infrastructure already in place for passing label mappings upward as follows: in its outgoing TVMs, each L_1 switch s_1 passes only a single mapping upwards for each neighboring host. This mapping includes a host's optimal label and its UID. These mappings are passed through all available upward paths, and therefore may encounter switches that reach s_1 (and its hosts) via a label other than the optimal label. When an $L_{i:2 \le i \le n}$ switch t encounters a mapping for which none of its labels is a prefix, it knows that this must be an optimal label mapping for one of its descendants. t adds a (downward facing) FTE $\ell \rightarrow q$ that will subsequently point messages destined for the optimal label in question (ℓ)

to the sender of the TVM (q). This process continues up to the top level of the network, so that any switch along the way that does not reach a host via its optimal label is able to add a new entry to its forwarding state.

For example in Figure 6.3, S_{64} adds extra FTEs that map optimal labels 65.49.32 and 65.49.33 to its downward neighbor S_{48}. This way, when it receives a packet destined for one of these labels, it can pass the packet to the appropriate next hop despite the fact that it does not itself have a label matching a prefix of one of these destinations. Note that S_{64} cannot combine these labels into a single FTE for prefix 65.49.$*$; we defer discussion of why this is not possible until Section 6.2.3. Switches S_{80} and S_{81} also add exceptions to their forwarding state to map 65.49.32 and 65.49.33 to S_{64}. Similar FTEs are in place at S_{76}, S_{80} and S_{81}, for S_{44} and S_{45}.

This adds little overhead to the basic ALIAS protocol, as switches only have to check their own labels for prefixes of each incoming mapping's label. However, it does increase the forwarding state of switches that need to add exceptions for optimal labels. We analyze the increase in forwarding table size in Section 6.5.

By propagating label selections upward, we have ensured that the L_n switches with downward paths to a given host still maintain forwarding connectivity to that host in the context of single label selection, even if they did not previously reach the host by its optimal label. However, in order to provide full connectivity between all pairs of hosts, similar label mappings need to be passed downwards through the tree as well. Labels are passed downwards in a similar manner to that for passing labels upwards, and a label ℓ stops moving downward once it reaches a switch that already has an FTE for ℓ or for a prefix of ℓ.

For example, in Figure 6.3, switch S_{76} connects to only two L_n switches, S_{80} and S_{81}. Therefore, it only has upward FTEs for the coordinates of the L_3 hypernodes reached by S_{80} and S_{81}, i.e. 64, 68 and 72. If it receives a packet destined for 65.49.32, it does not know how to move the packet upwards towards its destination. Therefore, when S_{80} and S_{81} learn of the optimal label mapping for 65.49.32, they push this mapping downward to S_{68}, S_{72} and S_{76}. As an optimization, S_{80} and S_{81} do not send the mapping to S_{64} since they know that they originally learned of the mapping from S_{64}. S_{68}, S_{72} and S_{76} push this mapping to their lower neighbors, S_{52}, S_{54}, S_{56}, S_{58} and S_{60}, but these

L_2 switches do not store related FTEs as they already have the label prefix 65 in their forwarding tables.

This process increases ALIAS TVM size as it adds a set of optimal label mappings to each downward message. The added overhead is dependent on the number of L_1 switches (regardless of how many hosts connect to each such L_1 switch) with multiple labels as well as on the number of switches that previously only reached a multiple-label L_1 switch through suboptimal labels (and therefore need to pass on forwarding exceptions). This process also increases the forwarding table sizes, as explored in Section 6.5.

6.2.3 Combining Single Label Forwarding Table Entries

In general, ALIAS switches cannot combine two optimal label FTEs that share a prefix into a single, shorter FTE. That is, if a switch has two optimal label FTEs of the form $x.y.z$ and $x.y.q$, it cannot necessarily combine these FTEs into a single $x.y.*$ FTE. This would imply that the switch has access to the entire $x.y$ hypernode, which may not be the case. There are cases in which optimal label FTEs could be combined given a global view of the topology, but switches in ALIAS do not have global information.

Figure 6.4 shows a small topology subset to exemplify this concept. In the figure, switches S_6, S_7 and S_8 have label sets {9.6,10.6}, {9.7,10.7} and {10.8}, respectively. Suppose that due to connections to L_n switches not shown in this subset of the topology, switches S_6 and S_7 have selected optimal labels associated with hypernode 10. In this case, S_{11} requires downward facing exceptions that map 10.6 and 10.7 to S_9. However, despite the shared prefix, S_{11} cannot combine these labels into a single FTE for $10.*\rightarrow S_9$, as this would imply that S_{11} can reach 10.8 via S_9, which is not the case.

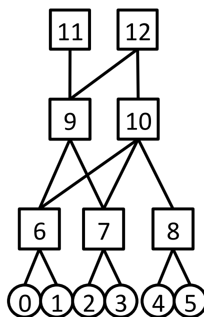


Figure 6.4: Combining Optimal Label Forwarding Entries

6.3 Effects of Single Label Selection on ALIAS

Reducing an ALIAS host's label set down to a single, optimal label affects the efficiency of communication within ALIAS. In particular, we consider the interactions of optimal label selection with multi-path forwarding support, peer link usage and reactivity after topology changes.

6.3.1 Effect on Multi-Path Support

One of the benefits of multi-rooted tree topologies is the inherent path multiplicity between pairs of nodes in the hierarchy; this benefit is especially pronounced in fat tree topologies. However path multiplicity does come at the cost of increased hardware and wiring complexity, and so it is important that single label selection does not interfere with multi-path support in ALIAS. As it is described in Section 6.2, the use of single label selection can reduce the multi-path support available in ALIAS's multi-rooted tree topologies.

For instance, consider the tree of Figure 6.5. The topology is identical to that of Figure 6.3 with the exception of the newly added link between S_{64} and S_{82}.[4] This new link increases the CV of S_{32}'s label 64.48.32 by 32, since the LNV of S_{82} is 32 and there is one path from S_{82} to S_{32} via label 64.48.32. The new CV of 64.48.32 is 80, which is still less than 65.49.32's CV of 192, so 65.49.32 which remains as S_{32}'s optimal label.

Recall that mappings corresponding to optimal labels are propagated upwards and then downwards through the tree, stopping once they arrive at switches that are able to reach an an optimal label without the help of special optimal label FTEs. In the case of Figure 6.5, S_{82} reaches S_{32} via FTE $65.*\rightarrow S_{65}$. Because of this, S_{82} does not need an additional FTE that maps 65.49.32 to S_{64} for reachability, and so the algorithm described above will not create an FTE of the form $65.49.32\rightarrow S_{64}$. However, this means that anytime a packet destined for label 65.49.32 arrives at S_{82}, S_{82} sends the packet downward via S_{65}, even though S_{64} can also reach the packet's destination. This essentially renders the link between S_{64} and S_{82} useless, removing half of S_{82}'s options for label 65.49.32.

[4] This assumes that S_{82} has an additional port available to make this connection.

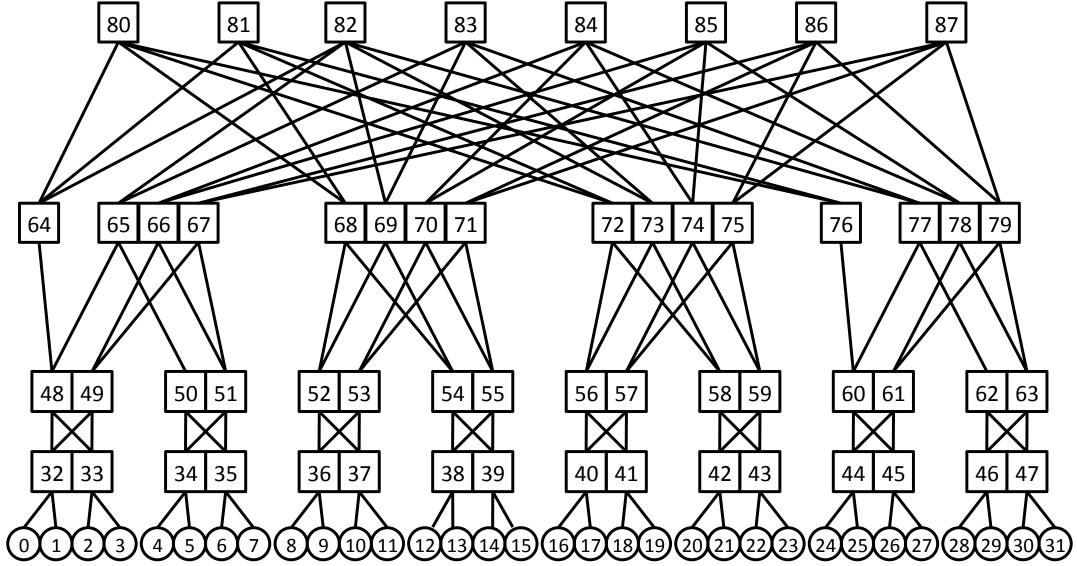


Figure 6.5: Multi-Path Support with Optimal Label Selection

This exemplifies a case in which mappings are not propagated upward sufficiently for multi-path support; The case is similar for optimal label mappings moving downwards. If an L_{n-1} switch s_{n-1} has a non-optimal label FTE that matches a label ℓ or prefix of ℓ, and it does not store an FTE of the form $\ell \rightarrow s_n$ pushed down from an L_n switch s_n, s_{n-1} cannot forward packets through s_n towards ℓ. It will instead use its existing FTEs. For instance, in Figure 6.5, S_{56} does not store an FTE of the form $65.49.32 \rightarrow S_{72}$, because it already reaches hypernode 65 via S_{73}, prior to the propagation of optimal label mappings. However, this limits the upward path for packets at S_{56} that are destined for 65.49.32 to the single link between S_{56} and S_{73}, rather than allowing S_{56} to select between both of the upper neighbors that reach S_{32}.

These effects can be characterized as follows: Let Q be the set of labels associated with an L_1 switch s_1. Let ℓ be s_1's optimal label and $R = Q \setminus \{\ell\}$ be the other, suboptimal labels of s_1. We partition switches in the topology as follows: S_ℓ represents switches that only reach s_1 via ℓ, S_R represents switches that only reach s_1 via labels $r \in R$ and not via ℓ, and $S_{\ell \wedge R}$ represents switches that reach s_1 via ℓ and also via at least one label $r \in R$. In order to provide full connectivity between hosts in the face of optimal label selection, all switches $s \in S_R$ require FTEs to map label ℓ to the appropriate neighbor. Switches in S_ℓ only reach s_1 via label ℓ and do not need additional forwarding

information generated during optimal label selection. In the label propagation scheme described in Section 6.2.2, switches in $S_{\ell \wedge R}$ do not receive additional mappings for optimal label selection. However, these switches correspond directly to those in the example given above; in order to provide full multi-path support, these switches would indeed need additional mappings to enable use of paths that initially corresponded to labels in R. For instance, in Figure 6.5, S_{56} is in the set $S_{\ell \wedge R}$ for label $\ell = 65.49.32$, as it reaches hypernode 65 and label ℓ with regular FTEs via S_{73} and it reaches label $r = 64.48.32$ via S_{72}. Therefore, for full multi-path support, S_{56} would need an FTE of the form $65.49.32 \rightarrow S_{72}$.

This brings to light a tradeoff between multi-path support and forwarding table size with single label selection. In order to provide full multi-path support, any switch that can reach an L_1 switch s_1 via a label $r \in R$ must add an FTE for s_1's optimal label ℓ, *regardless of whether it also reaches* ℓ *with regular FTEs.* Since these optimal label mappings add to the control overhead of ALIAS, a decision of whether to use S_R or $S_{\ell \wedge R}$ type mappings should be made prior to deployment. This decision can be changed on the fly at any time, but if S_R is the current choice, $S_{\ell \wedge R}$ entries should not be passed throughout the tree unnecessarily.

The propagation of optimal label mappings to switches in $S_{\ell \wedge R}$ can also interfere with longest prefix match forwarding. Since optimal label mappings include full labels rather than prefixes, and cannot be combined based on shared prefixes, they will be at least as long as regular FTEs, if not longer. A longest prefix match forwarding style will cause switches to favor paths corresponding to optimal label FTEs as opposed to other paths. This is not a concern for switches in S_R, as optimal label FTEs are the only entries that match a target destination. However, for switches in $S_{\ell \wedge R}$, regular and optimal label FTEs will coexist, and in many cases, optimal label entries will be longer than regular entries. We address this via our forwarding protocol. We accumulate information for all FTEs, including optimal label FTEs, in what we call a *super table*, stored in software. We then use this super table along with local policy, to populate hardware forwarding tables. We provide further details in Section 6.4.

6.3.2 Effect on Peer Link Usage

The interaction between optimal label selection and peer links is complicated and deserves special attention. In particular, we consider the question of whether optimal label mappings should be passed across peer links.

We first consider the case in which a switch s is able to reach an L_1 switch's optimal label both with and without traversing peer links. Here, we use the topology of Figure 6.6, which is similar to our initial example (Figure 6.3) but with a peer link added between S_{80} and S_{82}. This link gives S_{80} access to hosts $H_4 - H_7$ and $H_{28} - H_{31}$, increasing S_{80}'s LNV to 32. This changes the CV of label 64.48.32 from 48 to 56, which still does not beat 65.49.32's CV of 192, leaving S_{32}'s optimal label as 65.49.32.

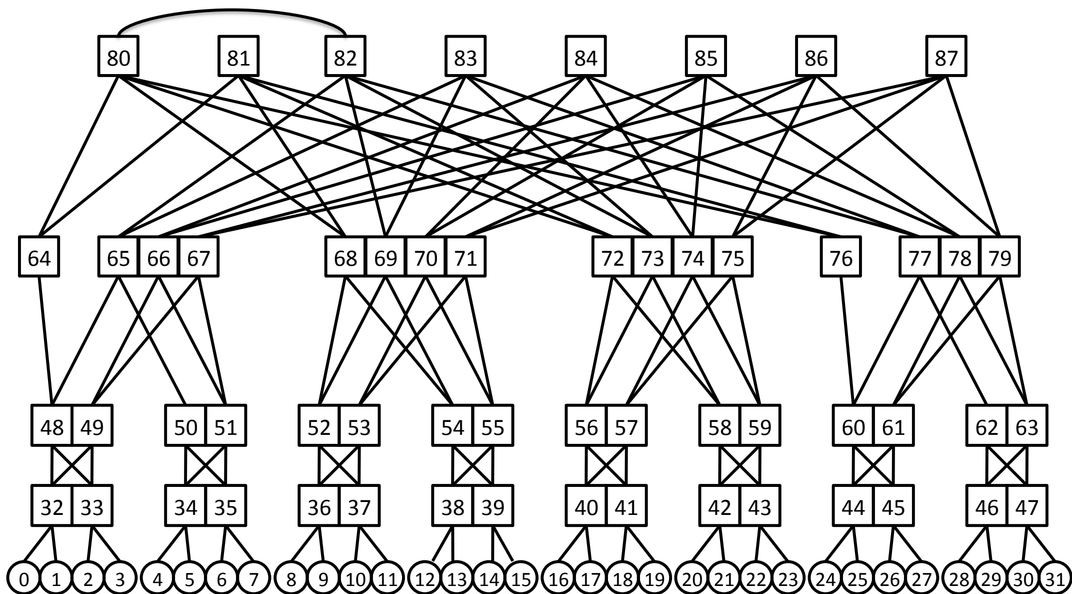


Figure 6.6: Peer Links and Optimal Label Selection

In the figure, S_{82} reaches S_{32} via its peer link to S_{80} and also via its downward link to S_{65}. Note that the former of these two links would require an optimal label mapping to propagate across the peer link, whereas the latter already uses S_{32}'s optimal label. If we applied the more constrained scheme for optimal label propagation discussed in Section 6.2.2, S_{82} would not store an optimal label FTE of the form 65.49.32$\rightarrow S_{80}$, as it already has the regular FTE 65.$*\rightarrow S_{65}$. However, this limits the paths that a packet can take from S_{82} to S_{32} in the same way as described in Section 6.3.1.

On the other hand, if S_{82} is given a mapping from 65.49.32 to S_{80}, longest prefix matching will favor the peer link, and the FTE 65.$*\rightarrow S_{65}$ will effectively be lost. This would be especially problematic if S_{82} had multiple connections to hypernode 65 and yet still always chose a single peer link when forwarding to S_{32}. In fact, this problem is not unique to optimal label mappings across peer links and can occur with any mappings passed across peer links.

As described in Section 6.3.1, ALIAS addresses this by creating a super table that includes all possible forwarding information, including optimal label FTEs as well as peer link FTEs. ALIAS uses the information in this table along with local policies to populate a switch's hardware forwarding tables. An example of a policy for optimal label mappings received via peer links would be to inject only those optimal label FTEs for which other routes are not available. A user may also elect to favor certain intentional peer links over others, or to favor peer links at certain levels of the topology.

We next consider the case in which a switch s reaches an L_1 switch's optimal label only via peer links, as is the case in Figure 6.7. In the figure, the links between S_{64} and S_{81} and between S_{76} and S_{80} have failed and a peer link between S_{80} and S_{81} has replaced that between S_{80} and S_{82}.

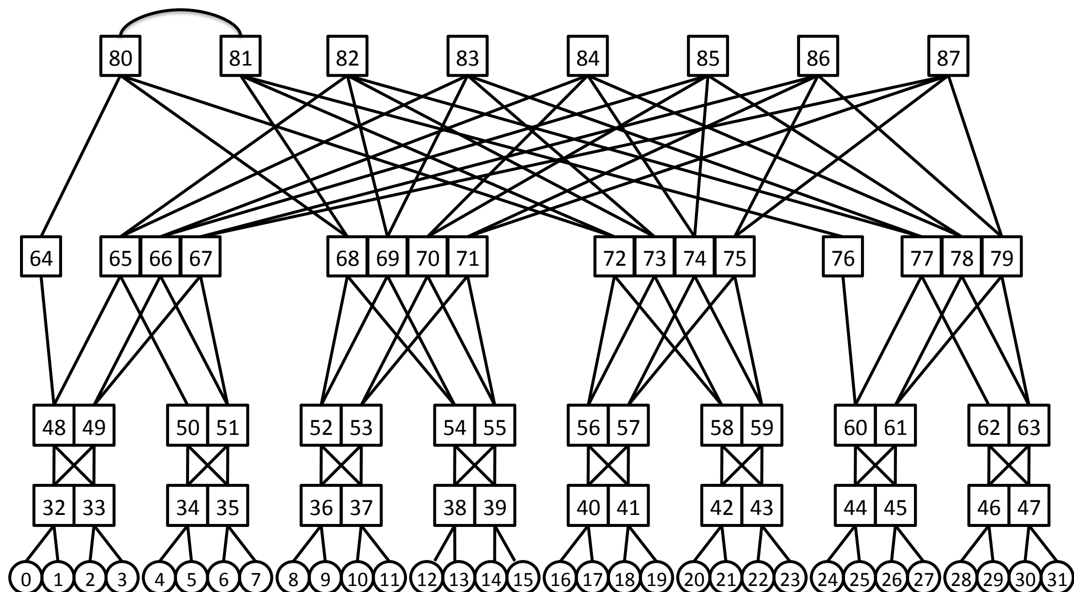


Figure 6.7: Connectivity-Providing Peer Links and Optimal Label Selection

As in our initial example, S_{80} and S_{81} are disconnected from hosts $H_4 - H_7$ and $H_{28} - H_{31}$ and therefore each have an LNV of 24. Thus, the CVs of labels 64.48.32 and 65.49.32 become 48, and 192, respectively. S_{81} only reaches label 65.49.32 via its peer link to S_{80}, and therefore it requires an optimal label FTE of the form 65.49.32$\rightarrow S_{80}$ in order to maintain connectivity with S_{32}. S_{81} must pass a similar mapping down to S_{76} as well. Because these optimal label FTEs are the only ways for switches S_{76} and S_{81} to reach S_{32}, they do not share prefixes with regular FTEs in the tables of S_{76} and S_{81}, and so they will be copied directly from the super table into the hardware forwarding tables of these switches.

6.3.3 Effect on Reactivity to Topology Dynamics

In general, ALIAS is designed to react quickly to topology changes. Because an ALIAS host label is based on paths to that host, certain topology changes cause this label to change. This begs the question of whether optimal label selection should follow suit. If a host's set of labels changes, and the CVs of these labels change as well, should the selected optimal label change as a result? In our experience, there are some cases in which the optimal label should change and others in which it should not. For instance, if a topology change causes a host's optimal label to disappear, the host's L_1 switch must necessarily select a new optimal label and propagate corresponding mappings throughout the tree. In this case, the convergence time is bounded by twice the number of levels in the tree. On the other hand, if a topology change simply causes the CV of a host's labels to change, care must be taken in deciding whether to select a new optimal label.

Figure 6.8 depicts this type of scenario. The figure shows a nearly perfect 4-level, 4-port fat tree, with the exception of a "flaky" link that toggles on and off between S_{64} and S_{50}. When this link is working, the topology is a perfect fat tree, and switches S_{32} and S_{33} have one label each, 64.48.32 and 64.48.33, respectively. However, when the link is off, each switch has a set of two labels, $\{64.48.32, 65.49.32\}$ and $\{64.48.33, 65.49.33\}$. In this case, L_n switches S_{80} and S_{81} do not reach hosts H_4-H_7 and therefore have LNVs of 28, giving labels 64.48.32 and 64.48.33 each a CV of 56. The other six L_n switches reach all hosts and have LNVs of 32, making the CVs of labels 65.49.32 and 65.49.33 equal to $6 \times 32 = 192$. As the flaky link toggles on and off,

the selected labels of S_{32} and S_{33} (and therefore their connected hosts) change back and forth between the default labels 64.48.32 and 64.48.33 and the optimal labels 65.49.32 and 65.49.33. Note that while this scenario involves a host moving from a set of two labels to only one, there are also cases in which a host has a set of several labels and the optimal label out of this set changes along with a flaky link or other topology dynamics. Because such fluctuation in hosts' optimal labels is likely undesirable, mechanisms such as hysteresis or other local policy are necessary to prevent constant changes to hosts' optimal labels.

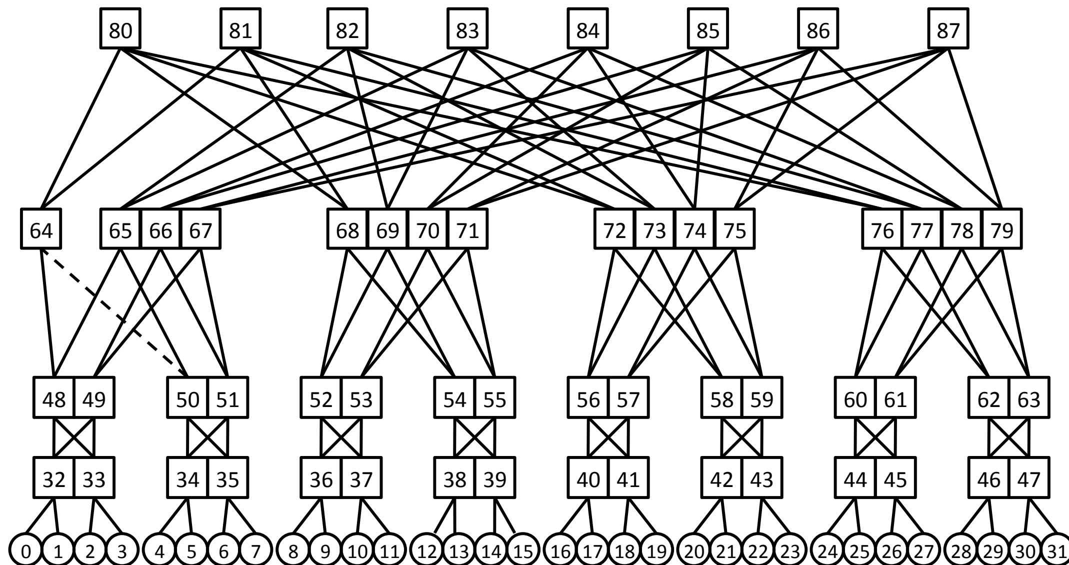


Figure 6.8: Optimal Label Selection and Topology Changes

6.4 Building Forwarding Tables with Single Label Selection

As discussed in Section 6.3, ALIAS with single label selection generates overlapping forwarding table entries at switches, and certain forwarding schemes (e.g. longest prefix matching) will not necessarily prioritize these entries as a network administrator would prefer.

6.4.1 Super Tables

In order to correctly implement the expected types of policies for selecting between overlapping FTEs, ALIAS needs to know the origin of the entry, that is, whether it is a regular entry, an optimal label entry, or an entry corresponding to a peer link. It may also need to know other information such as the direction of the entry or whether a peer link entry overlaps with one or more regular entries.

To support this, ALIAS computes a *super table* in software with all possible FTEs, including those that might not be used in the actual hardware forwarding table. Entries in the super table come with a tag that indicates the type of entry, i.e. whether the entry points upwards or downwards or comes from a peer link or optimal label, etc. ALIAS then uses several different policies to generate hardware forwarding tables from super tables, such that the hardware forwarding table can be accessed in a "first match" manner.

We first give an example of a simplified super table for a single switch and then discuss specific super table entries in more detail. We repeat, in Figure 6.9, the topology of Figure 6.6. In the figure, S_{64} has both upward- and downward-facing regular forwarding entries, as well as entries corresponding to paths across peer links and entries generated by optimal label selection. An initial super table for S_{64} is shown in Figure 6.10. This table includes all possible FTEs for S_{64}.

Since S_{64} connects to S_{80}, which has downward connections to hypernodes 68 and 72, S_{64} has *regular upward* entries in its super table pointing to S_{80} for labels that begin with these coordinates. S_{64} also has a *regular downward* entry for its only L_2 neighbor, 48. S_{64} has no peer links itself, and therefore no *peer link across* entries. However, it does have *peer link upward* entries corresponding to the hypernodes 68, 72 and 76, which it reaches via its neighbor S_{80} through S_{80}'s peer link to S_{81}. Finally, because S_{32} and S_{33} use labels other than those obtained from hypernode 64, S_{64} has downward facing *optimal label* entries for these switches.

Note the varying sizes of the label prefixes in each entry type; for a switch at L_i:

- *regular upward* entries map L_{n-1} labels (length = 1) to upper neighbors,
- *regular downward* entries refer to switches at L_{i-1} and therefore map label pre-

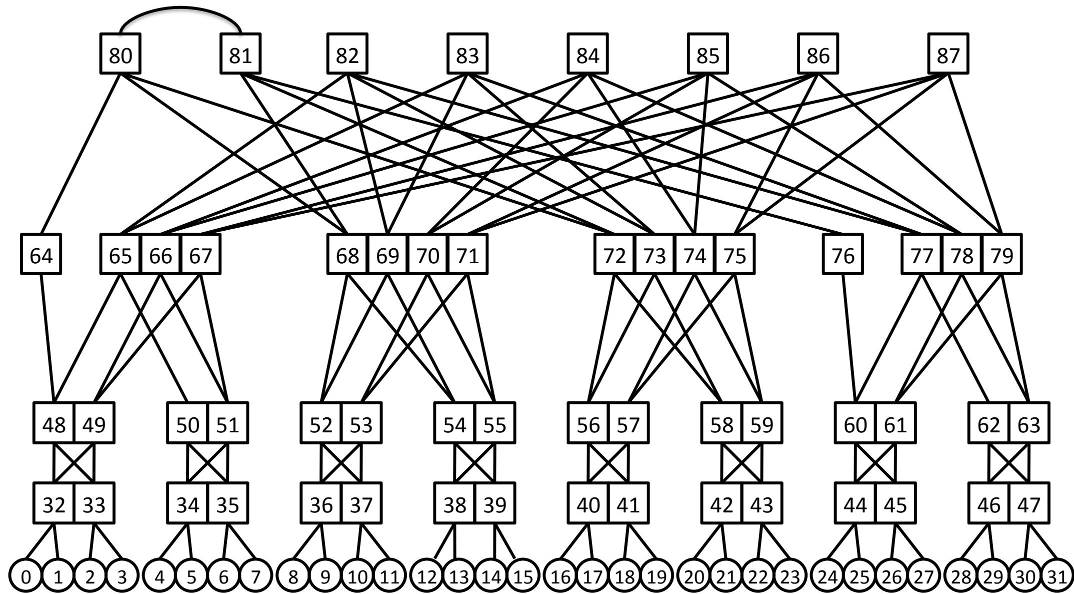


Figure 6.9: Super Table Example Topology

Regular		Peer Link		Optimal Label
Up	Down	Across	Up	
68.*→S_{80}	64.48.*→S_{48}		68.*→S_{80}	65.49.32→S_{48}
72.*→S_{80}			72.*→S_{80}	65.49.33→S_{48}
			76.*→S_{80}	

Figure 6.10: Initial Super Table for S_{64}

fixes of length $n-(i-1)=n-i+1$,

- *peer link across* entries refer to switches at L_i and map prefixes of length $n-i$,
- *peer link upward* entries refer to switches at higher levels than L_i and map label prefixes of length $n-j, j>i$, and finally,
- *optimal label* entries refer to L_1 switches and therefore map labels of length $n-1$.

Before computing entries for a switch's actual forwarding tables, ALIAS takes steps to consolidate super table entries when possible. For instance, there is no need for S_{64} to maintain its single *regular downward* entry, as it has been replaced by the *optimal label* entries, though one can construct a topology in which some *regular downward*

entries are not replaced by *optimal label* entries. Also, several of S_{64}'s *peer link upward* entries are redundant with its *regular upward* entries and can be removed. Optimal label entries cannot in general be combined into shorter shared prefixes. For instance, the labels 65.49.32 and 65.49.33 in the above super table cannot be combined into 65.49.$*$, as this indicates that S_{64} reaches all of hypernode 65.49, which is not necessarily true. Figure 6.11 shows the consolidated super table for S_{64} of Figure 6.9. Once the super table has been consolidated, ALIAS can use local policy to populate a switch's hardware forwarding tables.

Regular		Peer Link		Optimal Label
Up	Down	Across	Up	
68.*→S_{80}			76.*→S_{80}	65.49.32→S_{48}
72.*→S_{80}				65.49.33→S_{48}

Figure 6.11: Consolidated Super Table for S_{64}

There are eight different types of FTEs that can appear in a switch's super table. We introduce Figure 6.12 to provide examples of the types of FTEs. Table 6.1 lists each type of FTE, its direction and various options, and an example from Figure 6.12. We consider each type of entry in turn below.

Regular Forwarding Table Entries

An ALIAS switch s at level L_i has *regular* FTEs that correspond to paths that do not cross peer links and paths that do not require optimal label mappings. These entries are further divided by direction. *Regular upward* entries are used to pass packets upwards when s does not have a label that is a prefix of the destination label. These entries are of the form $c_{n-1} \rightarrow upper_neighbors$, where c_{n-1} is the coordinate (label) of an L_{n-1} hypernode, and the set of upper neighbors indicated includes those L_{i+1} neighbors of s that have connectivity to hypernode c_{n-1}.

In some cases, ALIAS is able to combine multiple *regular upward* entries into a "star entry" that is mapped to several (or all) upward neighbors. This entry is of the form $* \rightarrow upper_neighbor(s)$ and indicates that any destination label not matching

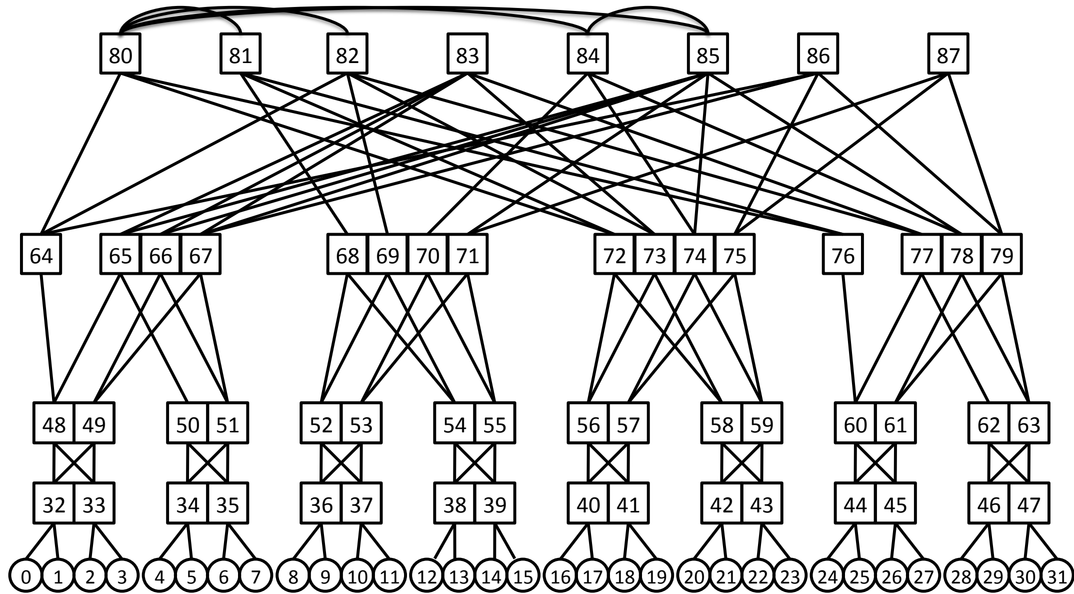


Figure 6.12: Topology for Super Table Entry Examples

another forwarding entry can be passed up to any of the target upper neighbors in the mapping. With this optimization, the number of *regular upward* forwarding entries may be reduced. In the super table of Figure 6.10, all of S_{64}'s upward FTEs point to the same upper neighbor, S_{80}. Therefore, they can be combined into a single star entry, $*\rightarrow S_{80}$. On the other hand, in Figure 6.12, none of S_{64}'s L_n neighbors reach all of the L_{n-1} hypernodes that S_{64} can reach, so S_{64} cannot use a star entry.

A switch contains *regular downward* entries to move packets towards the switch's descendants. A sample *regular downward* entry at S_{64} of Figure 6.12 is that mapping $64.48.*$ to S_{48}. A good policy for *regular* forwarding in ALIAS is longest prefix matching. When there is a path downwards to a destination, it does not make sense to push the packet up the tree and back down unnecessarily.

Peer Link Forwarding Table Entries

Super tables also contain *peer link* entries. A *peer link across* entry at s corresponds to a label reachable by traversing a peer link between s and one of its neighbors. Labels in *peer link* entries at L_i contain coordinates from L_{n-1} through L_i. There are two types of *peer link* entries in the super table:

Table 6.1: Summary of Super Table Entries
(Examples are from Figure 6.12.)

<table>
<tr><th>Category</th><th>Direction</th><th colspan="2">Options</th><th>Example</th></tr>
<tr><td rowspan="3">Regular</td><td rowspan="2">up</td><td colspan="2">No * Combo</td><td>S_{64}: 68.$*$→S_{82}</td></tr>
<tr><td colspan="2">With * Combo</td><td>(N/A in Fig. 6.12)</td></tr>
<tr><td>Down</td><td colspan="2"></td><td>S_{64}: 64.48.$*$→S_{48}</td></tr>
<tr><td rowspan="4">Peer</td><td rowspan="2">Across</td><td colspan="2">Unrestricted</td><td>S_{81}: 72.56.40→S_{80}</td></tr>
<tr><td colspan="2">Restricted</td><td>S_{80}: 68.$*$→S_{81}</td></tr>
<tr><td rowspan="2">Up</td><td colspan="2">Unrestricted</td><td>S_{68}: 64.48.32→S_{81}</td></tr>
<tr><td colspan="2">Restricted</td><td>S_{68}: 64.48.32→S_{81}</td></tr>
<tr><td rowspan="12">SingleLabel</td><td rowspan="2">Down</td><td colspan="2">S_R only</td><td>S_{64}: 65.49.32→S_{48}</td></tr>
<tr><td colspan="2">$S_{\ell\wedge R}$</td><td>S_{86}: 65.49.32→S_{64}</td></tr>
<tr><td rowspan="2">Up</td><td colspan="2">S_R only</td><td>S_{69}: 65.49.32→S_{82}</td></tr>
<tr><td colspan="2">$S_{\ell\wedge R}$</td><td>S_{56}: 65.49.32→S_{72}</td></tr>
<tr><td rowspan="4">Across</td><td rowspan="2">S_R</td><td>Restricted</td><td>S_{81}: 65.49.32→S_{80}</td></tr>
<tr><td>Unrestricted</td><td>S_{82}: 65.49.32→S_{80}</td></tr>
<tr><td rowspan="2">$S_{\ell\wedge R}$</td><td>Restricted</td><td>S_{84}: 65.49.32→S_{80}</td></tr>
<tr><td>Unrestricted</td><td>S_{85}: 65.49.32→S_{80}</td></tr>
<tr><td rowspan="4">Up</td><td rowspan="2">S_R</td><td>Restricted</td><td>S_{68}: 65.49.32→S_{81}</td></tr>
<tr><td>Unrestricted</td><td>S_{69}: 65.49.32→S_{82}</td></tr>
<tr><td rowspan="2">$S_{\ell\wedge R}$</td><td>Restricted</td><td>S_{70}: 65.49.32→S_{84}</td></tr>
<tr><td>Unrestricted</td><td>S_{71}: 65.49.32→S_{85}</td></tr>
</table>

- *Unrestricted peer links* include entries for all labels reachable within the system-wide peer link hop-count limit, while
- *restricted peer links* include entries only for those labels not reachable without traversing peer links.

Additionally, *peer link* entries have two directions, corresponding to peer links at s's level as well as those above.[5]

In Figure 6.12, S_{80} can only reach hypernode 68 and its descendants via its peer links to S_{81}, S_{82}, S_{84} and S_{85} and so a *peer link across* entry for this would appear in the super table whether we chose the *restricted* or *unrestricted* option. On the other hand, S_{81} can reach the label 72.56.40 via its peer link to S_{80} or directly through S_{72} and so a *peer link across* entry mapping this label to S_{80} would only occur in a super table with *unrestricted peer link across* entries. Similarly, S_{68} reaches label 64.48.32 through S_{81} (via its peer link to S_{80}) and cannot reach this label without the help of peer links. Therefore, a corresponding entry would appear in both the *restricted* and *unrestricted* types of super tables.

As an optimization, *peer link upward* entries can be omitted from the super table when redundant with *regular* entries. For instance, in Figure 6.12, S_{68} reaches label 72.56.40 via a *regular upward* entry to S_{81} (because S_{81} directly connects to hypernode 72) as well as via S_{81}'s peer link to S_{80}. In this case there is no need for S_{68} to store both types of entries. It simply passes packets destined for $72.*$ to S_{81} and leaves S_{81} to decide which path to use. More generally, if a destination label ℓ is reachable without using peer links via an L_n switch s_n, then the appropriate *regular upward* entries will exist at any descendant of s_n. Because of this, all *peer link upward* entries in the *unrestricted* case will be redundant with *regular upward entries* and will ultimately be omitted from the super table.

Note that it is not strictly necessary to determine which *peer link* entries fit into the *restricted* category when populating the super table; this option could instead be computed on the fly as the hardware forwarding table is populated. As discussed in Section 6.3.2, policies for incorporating *peer link* entries into a switch's hardware for-

[5]In ALIAS, packets are not allowed to traverse peer links after beginning a downward path in the network, so we do not require downward-facing peer link FTEs.

warding table vary and can include, for example, favoring specific peer links, preferring peer links at certain levels of the tree, or only allowing *restricted* peer link use.

Optimal Label Forwarding Table Entries

Finally, the super table includes several types of entries related to single label selection. Since the combination of optimal label propagation and peer link restriction is quite complicated, we first discuss optimal label entries not related to peer links.

The first type of *optimal label* entry, *optimal label downward*, refers to the downward-facing optimal label mappings that are passed upwards from an L_1 switch towards L_n switches, whereas *optimal label upward* entries encode the upward-facing mappings passed downwards from L_n switches (Section 6.2.2). Both of these types have two options, one in which only switches in S_R are given entries corresponding to optimal label mappings and the other in which switches in $S_{\ell \wedge R}$ also have optimal label-related FTEs (Section 6.3.1). For the following examples, we do not assume a particular metric for the selection of an optimal label. Rather, we simply select optimal labels that will provide for the clearest examples.

Suppose that the optimal label for S_{32} in Figure 6.12 is 65.49.32. An example of an S_R *optimal label downward* entry is the mapping at S_{64} from 65.49.32 to S_{48}, as S_{64} does not reach S_{32} with this label via *regular downward* entries and so belongs to S_R for S_{32}. A *optimal label upward* entry with the S_R option is the mapping at S_{69} from 65.49.32 to S_{82}, since without optimal label FTEs, S_{69} can only reach S_{32} via label 64.48.32 (even with the help of peer links) making it a member of S_R for S_{32}. A *optimal label upward* entry for the $S_{\ell \wedge R}$ option is the mapping at S_{56} from 65.49.32 to S_{72}. Since S_{56} can reach hypernode 65 without optimal label forwarding (via S_{73}), and can reach the label 64.48.32 (and thus 65.49.32 with the help of optimal label entries) via S_{72}, it belongs to $S_{\ell \wedge R}$ for S_{32}. Finally, S_{86} can reach S_{32} with label 65.49.32 via S_{67} as well as with label 64.48.32 via S_{64}. Therefore, S_{86} belongs to $S_{\ell \wedge R}$ for S_{32} and has a *optimal label downward* entry mapping 65.49.32 to S_{64} in its super table.

The choice between the two types of *optimal label* policies is based on the required multi-path support for a topology, as described in Section 6.3.1. A similar policy to that for *peer link* entries is applied to *optimal label* entries; *optimal label upward*

entries that are redundant with *regular upward* entries are removed, and higher level switches as left to make the decision between the two types of paths.

Optimal Label Forwarding Table Entries with Peer Links

We next turn our attention to the combination of optimal label entries with peer link entries. Since *optimal label* mappings are passed across peer links, the super table contains both *across* and *upward* entries for such mappings, and includes *restricted*, *unrestricted*, S_R, and $S_{\ell \wedge R}$ options for each. For simplicity, the system-wide peer-link hop limit is 1 for the following examples.

Optimal label across and *optimal label upward* entries with the S_R and *restricted* options correspond to optimal label mappings that are passed across peer links to a switch s, for which s cannot otherwise reach the mapped label, via other non-peer-related forwarding entries or via other, suboptimal labels. In Figure 6.12, S_{81} can reach S_{32} only via label 64.48.32 and only via its peer link to S_{80}, so its mapping from 65.49.32 to S_{80} is an example of a *optimal label across, S_R, restricted* entry. S_{81}'s downward neighbor S_{68} has a similar mapping from 65.49.32 to S_{81}, an example of a *optimal label upward, S_R, restricted* entry. *Optimal label across* and *optimal label upward* entries with the S_R and *unrestricted* options correspond to optimal label mappings that are passed across peer links to a switch s, for which s can reach only a suboptimal label for a particular host, but can do so via both *regular* and *peer link* entries. In the figure, S_{82} can reach S_{32} only via label 64.48.32, but can do so using its peer link to S_{80} or its downward link to S_{64}, so its mapping from 65.49.32 to S_{80} is an example of a *optimal label across, S_R, unrestricted* entry. S_{82}'s downward neighbor S_{69} has a similar mapping from 65.49.32 to S_{82}, an example of a *optimal label upward, S_R, unrestricted* entry. However, this entry would be redundant with an existing *regular upward* entry for S_{69} and therefore would likely be removed on a consolidation pass through the super table.

Optimal label across and *optimal label upward* entries with the $S_{\ell \wedge R}$ and *restricted* options correspond to optimal label mappings that are passed across peer links to a switch s, for which s can reach a host via multiple labels (including the selected optimal label), but can do so only via *peer link* entries. In the figure, S_{84} can reach S_{32} via label 64.48.32 using its peer link to S_{80} and via label 65.49.32 using its peer link to

S_{85}. In fact, it can only reach S_{32} via peer links. Therefore, its mapping from 65.49.32 to S_{80} is an example of a *optimal label across, $S_{\ell \wedge R}$, restricted* entry. S_{84}'s downward neighbor S_{70} has a similar mapping from 65.49.32 to S_{84}, an example of a *optimal label upward, $S_{\ell \wedge R}$, restricted* entry. *Optimal label across* and *optimal label upward* entries with the $S_{\ell \wedge R}$ and *unrestricted* options correspond to optimal label mappings that are passed across peer links to a switch s, for which s can reach a host via multiple labels (including the selected optimal label), and can do so via both *regular* and *peer link* entries. In the figure, S_{85} can reach S_{32} via label 64.48.32 using its peer link to S_{80} and via label 65.49.32 using its downward link to S_{65}. Since it can reach S_{32} via both the optimal and other labels, and via *regular* and *peer link* entries, its mapping from 65.49.32 to S_{80} is an example of a *optimal label across, $S_{\ell \wedge R}$, unrestricted* entry. S_{85}'s downward neighbor S_{71} has a similar mapping from 65.49.32 to S_{85}, an example of a *optimal label upward, $S_{\ell \wedge R}$, unrestricted* entry.

It turns out that the combination of *optimal label* and *peer link* entries introduces a somewhat circular logic into ALIAS, in that the choice of whether to consider a particular peer entry to be part of the *restricted* or *unrestricted* option depends on which option (S_R or $S_{\ell \wedge R}$) is used for optimal labels. However, with the S_R option, optimal label entries are inserted based on necessity and this necessity depends on the treatment of peer links. The key to resolving this cycle is to consider first the base, non-peer link case: With the S_R option, upward and downward *optimal label* entries are only inserted when necessary for reachability, and with the $S_{\ell \wedge R}$ option, such entries are inserted when beneficial for multi-path support. Therefore, we apply the policy that peer link restrictions can "overrule" optimal label options. With the S_R option, *optimal label across* entries are never inserted unless necessary for reachability, even in the case of unrestricted peer links, since this is the intention of the S_R option. On the other hand, with the $S_{\ell \wedge R}$ option, *optimal label across* are inserted for non-reachability reasons (i.e. multi-path support) only when unrestricted peer links are used or when a label is not reachable without traversing peer links, thus making it part of the *restricted peer links* category.

6.4.2 Creating Forwarding Tables from Super Tables

The actual forwarding table at a switch *s* is created as follows: *regular downward* entries are inserted into a table in alphanumeric order. This order is chosen for ease in locating these entries when subsequently inserting other types of entries. At this point *regular upward* entries are inserted such that any *regular upward* entry *u* that is a prefix of a set of *regular downward* entries D_FTEs appears immediately after the last entry for $d_fte \in D_FTEs$. If compatible with the *peer link* and *optimal label* entries to be used, an optimal *regular upward star* entry may replace one or more *regular upward* entries. At this point, a "first match" forwarding table has been built.

If *peer link* entries are to be used, but with the *restricted* option, these entries can simply be added to the end of the table, before the *regular upward star* entry, as they represent otherwise unreachable hosts and there will be no prefixes of such entries already in the table. If *unrestricted peer link* entries are used, they are placed either immediately before or after corresponding *regular* entries, based on local policy stating when to prefer each type of link. The case for *optimal label* entries is nearly identical.

There is one final optimization to take into account, based on the type of forwarding table used. Consider an entry that points to multiple next hops in the super table. Some types of forwarding tables allow for multiple entries for a label (e.g. for use with ECMP [35]). In this case, such entries are split into multiple entries in the forwarding table. If this sort of multiplicity is not supported, a single next-hop is selected to represent the entry in the forwarding table.

6.5 Evaluation

In the sections above, we introduced several additions to the computation, control overhead and forwarding table sizes in ALIAS. All computational overhead is fairly small. Also, the control overhead varies with the same factors as do the forwarding table sizes. Since limiting the sizes of forwarding tables is a primary goal of ALIAS, we focus on that aspect in this evaluation.

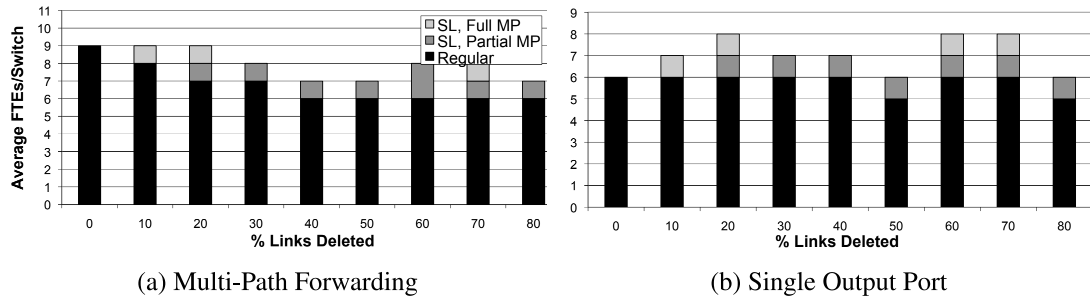
To evaluate forwarding table sizes with single label selection, we use our ALIAS simulator (Chapter 4). We first generate full fat tree topologies with *n* levels and *k*-port

switches. We then delete between $d = 0\%$ and $d = 80\%$ of the links at each level. For each topology, i.e. each combination of n, k and d, we calculate ALIAS hypernodes, coordinates, labels and forwarding tables. We extend the ALIAS simulator to generate and propagate optimal label mappings as well as the corresponding FTEs. We then compute number of FTEs at each switch, averaged over all switches in the topology and across 100 runs. Because of the way the graphs are created, there are no peer links. In fact, our results will show that even without the addition of peer link entries, the use of optimal label selection significantly increases forwarding table sizes.

We present forwarding table sizes, broken down by type of entry, for a variety of input topologies, policies, and optimizations in Figures 6.13 through 6.22. The x-axis shows the percentage of links deleted, and the y-axis shows the average number of FTEs per switch. The bottom portion of each column, denoted “regular”, shows the FTEs that would be present without single label selection. The middle portion, denoted “SL, partial MP”, gives the additional FTEs that would need to be added to incorporate support for single label selection with full reachability but without full multi-path support. That is, only switches in S_R have optimal label entries for a given L_1 switch. Finally, the top portion of each column, marked “SL, full MP”, shows the additional entries that would be necessary for full multi-path support, wherein switches in $S_{\ell \wedge R}$ have optimal label entries.

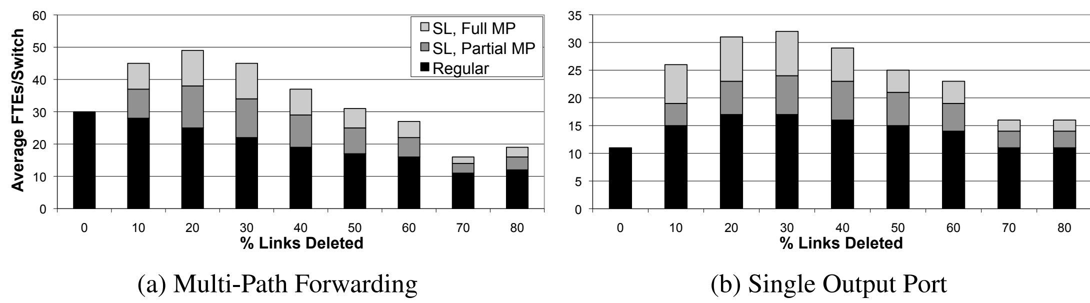
Each figure is additionally presented for both multi-path forwarding such as ECMP and single output port forwarding. In the multi-path forwarding case, FTEs in which one label points to multiple neighbors are counted once for each neighbor. In the single output port case, “duplicate” FTEs in which one label points to multiple neighbors, are counted only as single entries. This is because without ECMP-style forwarding, a single output port for each such entry would be selected. However, in these cases, all multi-path support is not completely lost. The super table contains all path entries possible, and so the switch CPU can use this information to update the hardware forwarding tables as an approximation of slow-path multi-path support.[6]

[6]These two classifications of multi-path support differ in that the S_R versus $S_{\ell \wedge R}$ option represents the amount of information known to each switch whereas the ECMP versus Single Output Port cases refer to the forwarding protocol’s use of a switch’s multi-path knowledge.



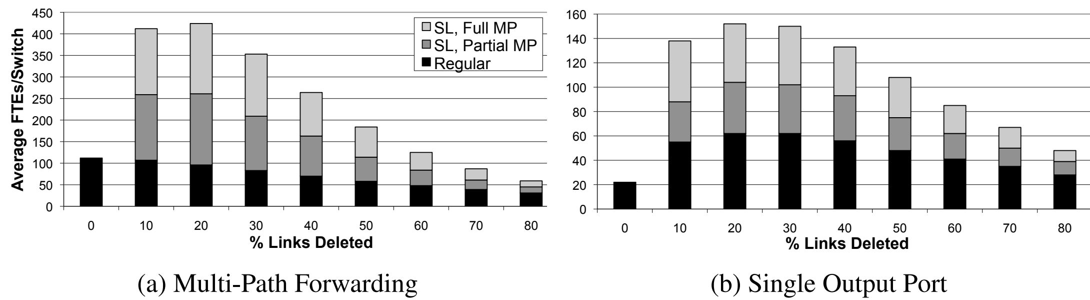
(a) Multi-Path Forwarding (b) Single Output Port

Figure 6.13: Forwarding Table Sizes for n=3, k=4



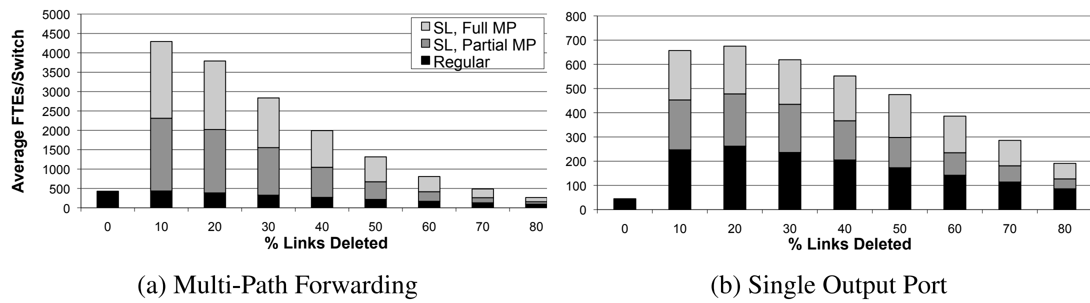
(a) Multi-Path Forwarding (b) Single Output Port

Figure 6.14: Forwarding Table Sizes for n=3, k=8



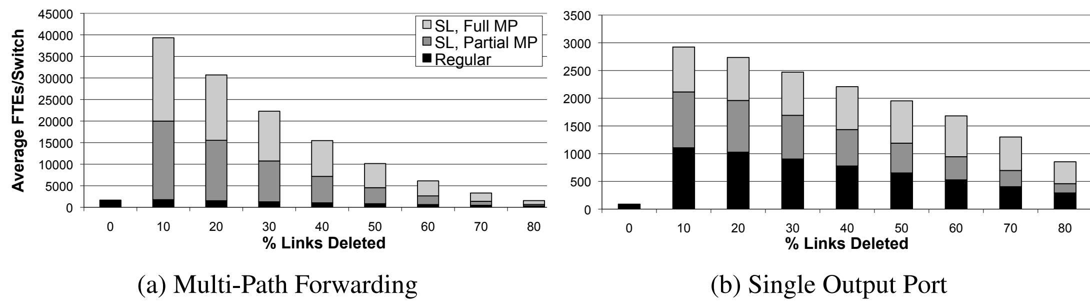
(a) Multi-Path Forwarding (b) Single Output Port

Figure 6.15: Forwarding Table Sizes for n=3, k=16



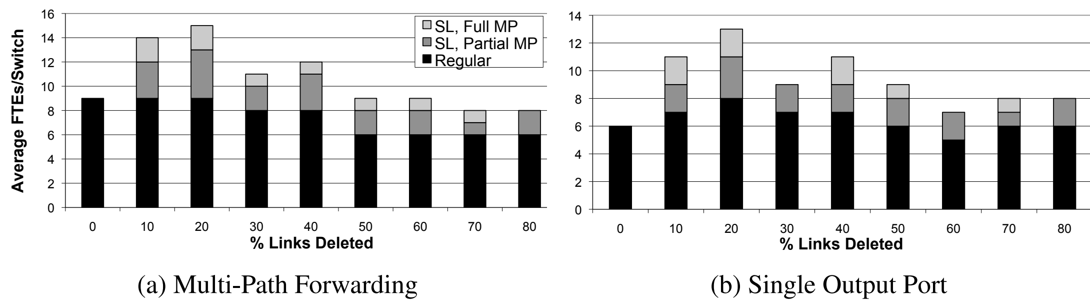
(a) Multi-Path Forwarding (b) Single Output Port

Figure 6.16: Forwarding Table Sizes for n=3, k=32



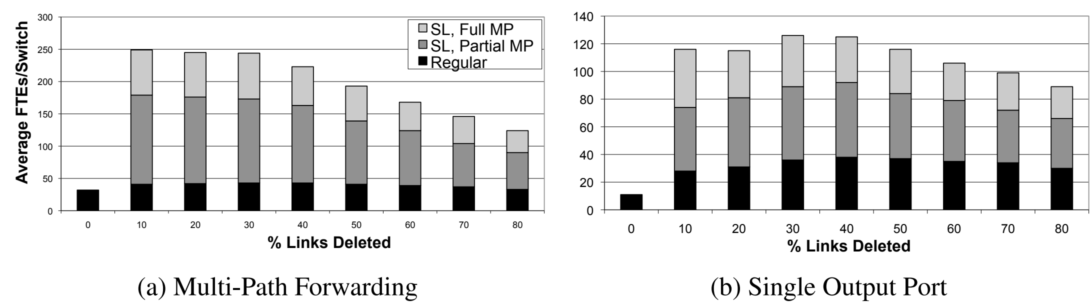
(a) Multi-Path Forwarding (b) Single Output Port

Figure 6.17: Forwarding Table Sizes for n=3, k=64



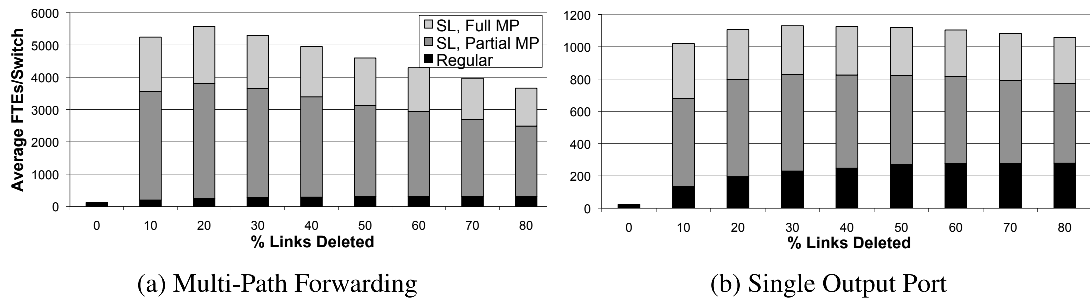
(a) Multi-Path Forwarding (b) Single Output Port

Figure 6.18: Forwarding Table Sizes for n=4, k=4



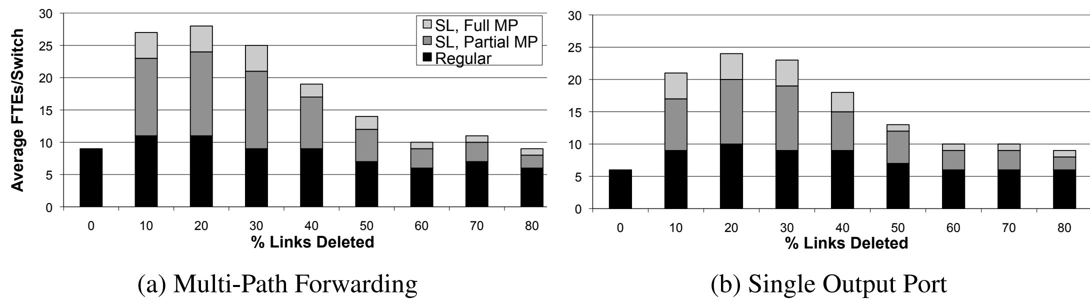
(a) Multi-Path Forwarding (b) Single Output Port

Figure 6.19: Forwarding Table Sizes for n=4, k=8



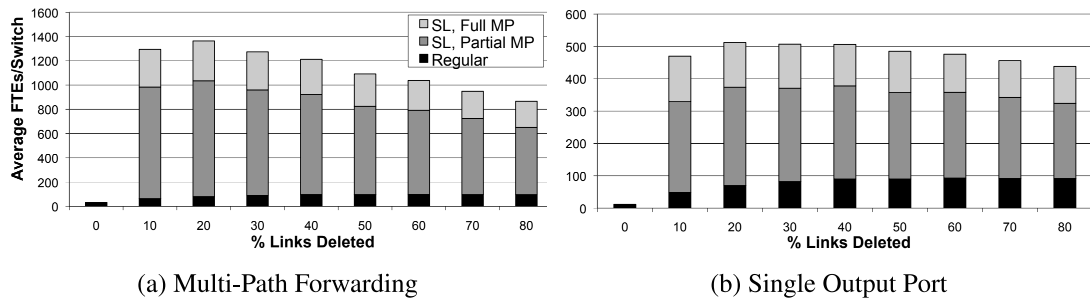
(a) Multi-Path Forwarding (b) Single Output Port

Figure 6.20: Forwarding Table Sizes for n=4, k=16



(a) Multi-Path Forwarding (b) Single Output Port

Figure 6.21: Forwarding Table Sizes for n=5, k=4



(a) Multi-Path Forwarding (b) Single Output Port

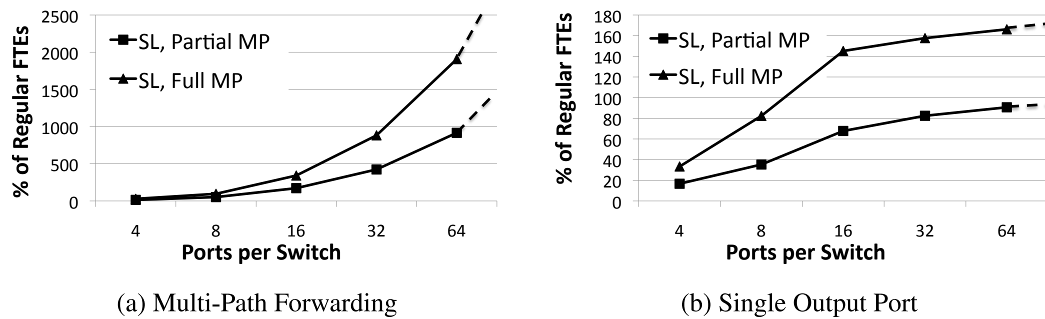
Figure 6.22: Forwarding Table Sizes for n=5, k=8

As the figures show, especially for larger trees, the cost in forwarding state associated with selecting a single, optimal label per host is significant, especially when full multi-path support is necessary. Since the graphs shown are for small values of k, we extrapolate to determine what forwarding table sizes might be like for larger networks as follows. We consider the case in which 20% of links fail for each graph where $n = 3$ and where k varies from 4 to 64. We compute the number of optimal label entries as a function of regular entries, for both partial and full multi-path support. That is, if there are 100 regular entries and 400 optimal label entries, we give a value of 400%, indicating that there are 4 times as many optimal label entries as regular entries. We show the case with 20% failed links as it appears to frequently represent the worst case for optimal label entries and because it does not correspond to an overly exaggerated number of failures. The results of this extrapolation appear in Figure 6.23.

In the multi-path forwarding case (Figure 6.23a), the multiplicative factor of single label vs. regular entries seems to grow exponentially with k. We perform an extrapolation to $k = 128$ with a dotted line, to show that the number of optimal label entries required for full multi-path support (i.e. $S_{\ell \wedge R}$) for $k = 128$ could be as much as 25 times that of regular entries. The case for partial multi-path support (i.e. S_R) is not much better. In this case, for $k = 128$, we would need about 15 times as many optimal label FTEs as we would regular FTEs.

On the other hand, in the single output port case (i.e. no ECMP-style forwarding) the number of optimal label FTEs grows more slowly with respect to the number of regular FTEs. For $k = 128$, with full multi-path support, we extrapolate to show that there might be as many as 1.8 times as many optimal label FTEs as regular FTEs, and with only partial multi-path support, the number of optimal label entries does not surpass that of regular FTEs.

In order to estimate numerical values corresponding to these extrapolations, we perform another extrapolation, this time showing the number of regular forwarding entries as a function of k, in Figure 6.24. This allows us to estimate the number of forwarding entries for $k = 128$. Based on this figure, we might expect a $k = 128$ switch to have around 2300 regular entries. This means that incorporating optimal label selection would introduce an additional 34,000 forwarding entries with partial multi-path support,



(a) Multi-Path Forwarding

(b) Single Output Port

Figure 6.23: Optimal Label Entries as a Percentage of Regular Entries

and another 57,000 entries with full multi-path support.

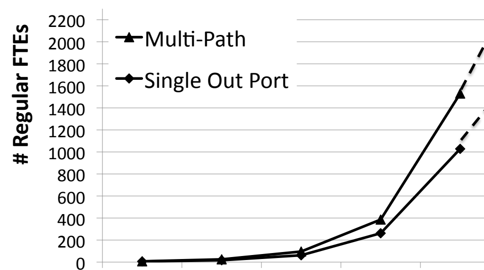


Figure 6.24: Regular Entries vs. Number of Ports Per Switch for $n = 3$

Given these results, the only cases that might correspond to usable networks are those with no ECMP-style multi-path support or perhaps those that incorporate ECMP-style forwarding, but that force single label selection to significantly diminish multi-path options in the forwarding tables. In the former case, we can not use a forwarding protocol that leverages any multi-path options calculated by ALIAS, and in the latter case we only calculate a small subset of multi-path options and pay a significant price in forwarding state. As such, it appears that the benefits of selection of a single optimal label per host in ALIAS are outweighed by the cost in terms of forwarding state.

6.6 Summary

ALIAS labels encode not only the locations of hosts and switches in the network, but also ways to reach (i.e. paths to) these hosts and switches. Because there are poten-

tially multiple ways to reach a host in a multi-rooted tree, ALIAS hosts have multiple labels. This is a significant departure from the interfaces of most modern data center communication protocols and so in this chapter, we consider the possibility of selecting a single label per host and subsequently propagating this selection through the ALIAS topology. We first offer a protocol for selecting an optimal label per host, prioritizing reachability and multi-path support. We then give an algorithm for propagating selected labels throughout the tree for two cases: (1) one case in which minimal forwarding state is a priority and selected labels are propagated only when necessary for connectivity and (2) one case in which selected labels are propagated everywhere necessary to take full advantage of the topology's inherent path multiplicity. We use simulations to evaluate the resulting forwarding state for a variety of topologies and find that in general, the selection of a single label is prohibitively expensive in terms of added forwarding state.

Chapter 7

Conclusions and Future Work

In this chapter, we summarize the challenges that we have addressed in this dissertation and describe our approaches to solving each. We then discuss directions for continuing research in these areas.

7.1 Summary

This dissertation sets out to show that we can have scalable, efficient and fault-tolerant communication in hierarchical data center networks, despite the data center's scale and complexity. That is, with tunable topology design and tailored communication protocols, we can overcome the following three challenges:

1. Building scalable, fault-tolerant topologies that allow network designers to tune scalability and fault tolerance tradeoffs according to the requirements for a particular situation.

2. Providing scalable and efficient addressing and communication.

3. Formalizing the underlying protocols and their interactions in order to make configuration and debugging feasible.

Here, we review our approaches to each of these challenges. In Chapter 3, we consider the issue of improving failure recovery in the data center by modifying fat tree

topologies to enable local failure reactions. A single link failure in a fat tree can disconnect a portion of the network's hosts for a substantial period of time while updated routing information propagates to every switch in the tree. This is unacceptable in the data center, where the highest levels of availability are required. To this end, we introduce the Aspen tree, a type of multi-rooted tree topology with the ability to react to failures locally. Aspen trees provide decreased convergence times to improve a data center's availability, at the scalability cost of reduced host count and hierarchical aggregation. We explore the tradeoffs between Aspen trees' fault tolerance and scalability properties and offer a set of "middle ground" trees that provide improved fault tolerance via localized failure reaction while still maintaining much of the fat tree's scalability.

With Aspen trees, we overcome challenge (1) by providing a class of multi-rooted tree topologies with tunable scalability and fault tolerance. A data center operator can design an Aspen tree that meets a particular scalability requirement while maintaining the highest possible fault tolerance. Alternatively, the operator can specify fault tolerance requirements and design an Aspen tree that supports as many hosts as possible. In this way, the data center network meets only those requirements necessary, and does not sacrifice one feature (i.e. fault tolerance or scalability) for an unnecessary increase in the other.

In Chapter 4, we address challenge (2). We discuss the fact that current data center naming and communication protocols rely on manual configuration or centralization to provide communication between end hosts. Such manual configuration is costly, time-consuming and error prone, and centralized approaches introduce the need for an out-of-band control network.

We take advantage of particular characteristics of data center topologies to design and implement ALIAS. We show how to automatically overlay appropriate hierarchy on top of a data center network such that end hosts can automatically be assigned hierarchical, topologically meaningful labels using only pair-wise communication, with no central components. Our evaluation indicates that ALIAS simplifies data center management while simultaneously improving overall scalability.

In Chapter 5, consider the issue of formalizing data center communication protocols. We study ALIAS in more depth, formalizing its basic building block so that

we can reason about the protocols' correctness and efficiency. We formally define the Label Selection Problem as a version of the network node labeling problem where (1) labels are restricted based on connectivity and (2) connectivity can change. We give a Las Vegas-style protocol, which we call the Decider/Chooser Protocol (DCP), that solves the Label Selection Problem in an efficient manner. We then apply this protocol to the problem of automatic label assignment in data center networks. We verify the correctness of DCP via proof and model checking, and explore its performance through analysis and simulation. We find that DCP is quick to converge, even with a small label domain, due to the random nature of the protocol.

Our formalization of DCP and our derivation of ALIAS from DCP allow us to reason about the correctness of ALIAS and its interactions with other data center network protocols. This is a step towards solving challenge (3), the formalization of networking protocols. Because of the scale and complexity of the data center, it is crucial that we build protocols that are provably correct and that have understandable configuration and debugging processes.

Finally, in Chapter 6, we return to the design of ALIAS and consider a modification to reduce an ALIAS host's label set to a single label for routing and forwarding use. We first propose an algorithm for selecting a single label per host, choosing this label based on reachability and multi-path support. We then show how to propagate selected labels throughout the topology and we discuss the effects of these changes on multi-path support, peer link usage and reactivity to topology changes. Finally, we use simulations to examine the forwarding state necessary to support single label selection. We find that the selection of a single label per host in ALIAS results in prohibitively large forwarding tables, and so we favor a version of ALIAS with multiple labels per host.

7.2 Open Problems and Future Work

The field of data center networking is still a young area, and new and innovative ideas appear continuously. Here, we discuss open problems related to the work in this dissertation as well as ideas for ongoing research in the areas of data center network topology, communication and fault tolerance.

7.2.1 Data Center Network Topologies

Random Topologies

Jellyfish [67] was recently proposed as a new way of thinking about data center network topologies. A Jellyfish topology is connected entirely randomly, without intentional hierarchy or symmetry. Despite the fact that paths between pairs of hosts are not likely to be uniform in Jellyfish topologies, the research shows promise for efficient forwarding and short paths. However, a random topology does not have the opportunity for the hierarchical label aggregation enjoyed by multi-rooted trees. Thus, there is a need for careful analysis of the forwarding state stored by Jellyfish switches.

This area is wide open for future research. While much effort has gone into designing regular, symmetrical, hierarchical structures, little has gone towards exploring and evaluating random topologies. In particular, it would certainly be interesting to run ALIAS over a Jellyfish topology and to examine the resulting hypernodes and labels. Perhaps this might lead to alternate definitions for hypernodes, wherein the definition selected for a given data center network might depend on the type of topology in use.

Improving the Scalability versus Fault Tolerance Tradeoff

Aspen trees enable fast, local failure reactions by leveraging the structure of the topology. In contract, there is a large body of research that studies the introduction of backup paths a priori or the use of alternative routing techniques such as bounce routing and data-driven connectivity. In these cases, the focus is generally on arbitrary topologies rather than on those frequently seen in the data center.

These two areas of research could be combined in order to improve fault tolerance in the data center at a smaller scalability cost than that of Aspen trees. That is, we could leverage information about a network topology to selectively use backup paths or alternative routing techniques. For instance, a network operator could design an Aspen tree with added fault tolerance only at a subset of tree levels. Then, alternative routing techniques could be used to “pick up the slack” at other levels of the tree. By limiting the use of alternative routing techniques and by leveraging topology information, we can reduce the complexity inherent to alternative routing techniques. At the same time, by

only introducing added fault tolerance to a subset of tree levels, we limit the scalability cost in terms of host support.

7.2.2 Scalable Communication

End-To-End Protocols

End-to-end protocols for flow scheduling and load balancing have become increasingly important in modern data centers, where a key priority is achieving full utilization of the topology's offered bisection bandwidth. A number of recent research efforts focus on related problems [5, 28, 55, 60].

ALIAS introduces the idea of encoding path information in a host's address. In particular, each ALIAS label corresponds to a set of paths to a host from the top level of a multi-rooted tree. When a sender chooses a particular label for a flow, it effectively selects a subset of all possible paths to the flow's destination. The interaction between this path encoding and load balancing or flow scheduling protocols is an area of ongoing research.

Software Defined Networking

Software-Defined Networking (SDN) has gained significant traction in recent years, in part due to the widespread adoption of OpenFlow [1]. SDN gives networking researchers the opportunity to separate the control and data planes, thus enabling a logically centralized control plane. This introduces an interesting middle ground between centralized and fully distributed protocols. On one hand, centralized protocols can be undesirable in the data center due to the corresponding need for a separate out-of-band control network to connect each node to a single centralized component. On the other hand, fully distributed protocols can be complex, making them difficult to configure and debug. With SDN, it is possible to have logically, but not physically, centralized control.

We could operate in this middle ground by introducing partial centralization in ALIAS. For instance, the topology could be divided into regions, with a centralized OpenFlow-style controller per region. Controllers could even be co-located with ALIAS switches. In this case, a simple distributed protocol could be used to elect a controller for

each region of the topology. Then, each controller could handle the more complicated tasks for its region, such as hypernode computation and coordinate assignment.

7.2.3 Formalizing Data Center Protocols

Tight Upper Bound on DCP Convergence Time

While we have shown with simulations and mathematical analysis that the Decider/Chooser Protocol converges quickly in practice, it would be useful to calculate a tight upper bound for this convergence time as a function of the coordinate domain size and the number of choosers. This upper bound is difficult to compute for several reasons. First, we introduce the notion of rounds into our analysis, and we compute the probability that a subset of the remaining choosers finish during each round. As the protocol is inherently asynchronous, it would be ideal to analyze the convergence time without relying on the construct of rounds. Additionally, the input to each round depends on the results of the previous round. That is, the number of choosers that remain before the start of round x is based on the number of choosers that remain before round $x-1$ as well as on the number of choosers that finish during round $x-1$. The results of a round form a probability distribution, making it difficult to use these results to generate an input for the following round.

7.3 Acknowledgment

Chapter 7, in part, contains material submitted for publication as “Scalability vs. Fault Tolerance in Apsen Trees.” Walraed-Sullivan, Meg; Vahdat, Amin; Marzullo, Keith. The dissertation author was the primary investigator and author of this paper.

Chapter 7, in part, contains material as it appears in the Proceedings of the ACM Symposium on Cloud Computing (SOCC) 2011. Walraed-Sullivan, Meg; Niranjan Mysore, Radhika; Tewari, Malveeka; Zhang, Ying; Marzullo, Keith; Vahdat, Amin. The dissertation author was the primary investigator and author of this paper.

Chapter 7, in part, contains material as it appears in the Proceedings of the 25th International Symposium on Distributed Computing (DISC) 2011. Walraed-Sullivan,

Meg; Niranjan Mysore, Radhika; Tewari, Malveeka; Zhang, Ying; Marzullo, Keith; Vahdat, Amin. The dissertation author was the primary investigator and author of this paper.

Chapter 7, in part, contains material submitted for publication as “A Randomized Algorithm for Label Assignment in Dynamic Networks.” Walraed-Sullivan, Meg; Niranjan Mysore, Radhika; Marzullo, Keith; Vahdat, Amin. The dissertation author was the primary investigator and author of this paper.

Bibliography

[1] OpenFlow. http://www.openflowswitch.org, 2011.

[2] D. Abts and B. Felderman. A Guided Tour through Data-center Networking. *Queue*, 10(5):10:10–10:23, May 2012.

[3] J. H. Ahn, N. Binkert, A. Davis, M. McLaren, and R. S. Schreiber. HyperX: Topology, Routing, and Packaging of Efficient Large-Scale networks. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 41:1–41:11, New York, NY, USA, 2009. ACM.

[4] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, SIGCOMM '08, pages 63–74, New York, NY, USA, 2008. ACM.

[5] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, NSDI '10, pages 19–19, Berkeley, CA, USA, 2010. USENIX Association.

[6] Amazon.com, Inc. Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2/, 2012.

[7] M. Armbrust, A. Fox, R. Griffith, A. D. J. R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A View of Cloud Computing. *Communications of the ACM*, 53(4):50–58, April 2010.

[8] H. Attiya, A. Bar-Noy, D. Dolev, D. Peleg, and R. Reischuk. Renaming in an Asynchronous Environment. *Journal of the ACM*, 37:524–548, July 1990.

[9] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley & Sons, 2004.

[10] A. Banerjea. Fault Recovery for Guaranteed Performance Communications Connections. *IEEE/ACM Transactions on Networking (TON)*, 7(5):653–668, October 1999.

[11] A. Banerjea, C. J. Parris, and D. Ferrari. Recovering Guaranteed Performance Service Connections from Single and Multiple Faults. In *Global Telecommunications Conference*, GLOBECOM '94, pages 162–168. IEEE, November 1994.

[12] C. L. Belady. Projecting Annual New Datacenter Construction Market Size. White paper, Microsoft Global Foundation Services, March 2011.

[13] T. Benson, A. Akella, and D. A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *Proceedings of the 10th annual conference on Internet measurement*, IMC '10, pages 267–280, New York, NY, USA, 2010. ACM.

[14] L. N. Bhuyan and D. P. Agrawal. Generalized Hypercube and Hyperbus Structures for a Computer Network. *IEEE Transactions on Computers*, 33:323–333, April 1984.

[15] S. Chaudhuri, M. Herlihy, and M. R. Tuttle. Wait-Free Implementations in Message-Passing Systems. *Theoretical Computer Science*, 220(1):211–245, June 1999.

[16] K. Chen, C. Guo, H. Wu, J. Yuan, Z. Feng, Y. Chen, S. Lu, and W. Wu. Generic and Automatic Address Configuration for Data Center Networks. In *Proceedings of the ACM SIGCOMM 2010 conference on SIGCOMM*, SIGCOMM '10, pages 39–50, New York, NY, USA, 2010. ACM.

[17] Cisco Systems, Inc. OSPF Design Guide. http://www.ciscosystems.com/en/US/tech/tk365/technologies_white_paper09186a0080094e9e.shtml, 2005.

[18] Cisco Systems, Inc. Cisco Data Center Infrastructure 2.5 Design Guide. http://www.cisco.com/univercd/cc/td/doc/solution/, 2008.

[19] C. Clos. A Study of Non-Blocking Switching Networks. *Bell System Technical Journal*, 32(2):406–424, March 1953.

[20] W. J. Dally and H. Aoki. Deadlock-Free Adaptive Routing in Multicomputer Networks Using Virtual Channels. *IEEE Transactions on Parallel and Distributed Systems*, 4(4):466–475, April 1993.

[21] D. Dao, J. Albrecht, C. Killian, and A. Vahdat. Live Debugging of Distributed Systems. In *Proceedings of the 18th International Conference on Compiler Construction: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, CC '09, pages 94–108, Berlin, Heidelberg, 2009. Springer-Verlag.

[22] A. DeHon. Compact, Multilayer Layout for Butterfly Fat-Tree. In *Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*, SPAA '00, pages 206–215, New York, NY, USA, 2000. ACM.

[23] J. Dix. Google's software-defined/OpenFlow backbone drives WAN links to 100% utilization. http://www.networkworld.com/news/2012/060712-google-openflow-vahdat-259965.html, June 2012.

[24] Facebook, Inc. Open compute project. http://opencompute.org/, 2011.

[25] P. Fraigniaud, A. Pelc, D. Peleg, and S. Pérennes. Assigning Labels in Unknown Anonymous Networks (Extended Abstract). In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, PODC '00, pages 101–111, New York, NY, USA, 2000. ACM.

[26] A. V. Goldberg, B. M. Maggs, and S. A. Plotkin. A Parallel Algorithm for Reconfiguring a Multibutterfly Network with Faulty Switches. *IEEE Transactions on Computers*, 43(3):321–326, March 1994.

[27] Google, Inc. Google data centers. http://www.google.com/about/datacenters, 2009.

[28] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, SIGCOMM '09, pages 51–62, New York, NY, USA, 2009. ACM.

[29] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu. BCube: A High Performance, Server-Centric Network Architecture for Modular Data Centers. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, SIGCOMM '09, pages 63–74, New York, NY, USA, 2009. ACM.

[30] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu. DCell: A Scalable and Fault-Tolerant Network Structure for Data Centers. In *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, SIGCOMM '08, pages 75–86, New York, NY, USA, 2008. ACM.

[31] S. Guo and O. W. Yang. Performance of Backup Source Routing in Mobile Ad Hoc Networks. In *Wireless Communications and Networking Conference*, WCNC '02, pages 440 – 444. IEEE, March 2002.

[32] S. Han and K. G. Shin. Fast Restoration of Real-time Communication Service from Component Failures in Multi-hop Networks. In *Proceedings of the ACM SIGCOMM '97 conference on Applications, technologies, architectures, and protocols for computer communication*, SIGCOMM '97, pages 77–88, New York, NY, USA, 1997. ACM.

[33] M. R. Henzinger and V. King. Randomized Fully Dynamic Graph Algorithms with Polylogarithmic Time per Operation. *Journal of the ACM*, 46(4):502–516, July 1999.

[34] J. N. Hoover. Inside Microsoft's $550 Million Mega Data Centers. http://www.informationweek.com/news/208403723, June 2008.

[35] C. Hopps. Analysis of an Equal-Cost Multi-Path Algorithm, 2000.

[36] Juniper. What's Behind Network Downtime? http://f.netline.junipermarketing.com/netline000s/?msg=msg.htm.txt&_m=26.11dk.1.ua06p11znd.3to, May 2008.

[37] M. Karol, S. J. Golestani, and D. Lee. Prevention of Deadlocks and Livelocks in Lossless Backpressured Packet Networks. *IEEE/ACM Transactions on Networking (TON)*, 11(6):923–934, Dec. 2003.

[38] Z. Kerravala. As the Value of Enterprise Networks Escalates, So Does the Need for Configuration Management. https://acc.dau.mil/CommunityBrowser.aspx?id=176818, January 2004.

[39] C. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *Proceedings of the 4th USENIX conference on Networked systems design & implementation*, NSDI '07, pages 18–18, Berkeley, CA, USA, 2007. USENIX Association.

[40] C. Killian, J. W. Anderson, R. B. R. Jhala, and A. Vahdat. Mace: Language Support for Building Distributed Systems. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 179–188, New York, NY, USA, 2007. ACM.

[41] C. Killian, K. Nagarak, S. Pervez, R. Braud, J. W. Anderson, and R. Jhala. Finding Latent Performance Bugs in Systems Implementations. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, FSE '10, pages 17–26, New York, NY, USA, 2010. ACM.

[42] C. Kim, M. Caesar, and J. Rexford. Floodless in SEATTLE: A Scalable Ethernet Architecture for Large Enterprises. In *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, SIGCOMM '08, pages 3–14, New York, NY, USA, 2008. ACM.

[43] S. Kim and K. G. Shin. Improving Dependability of Real-Time Communication with Preplanned Backup Routes and Spare Resource Pool. In *Proceedings of the 11th international conference on Quality of service*, IWQoS '03, pages 195–214, Berlin, Heidelberg, 2003. Springer-Verlag.

[44] A. Kvalbein, A. F. Hansen, T. Cicic, S. Gjessing, and O. Lysne. Fast IP Network Recovery Using Multiple Routing Configurations. In *Proceedings of the 25th IEEE International Conference on Computer Communications.*, INFOCOM '06, pages 1 –11. IEEE, April 2006.

[45] K. Lakshminarayanan, M. Caesar, M. Rangan, T. Anderson, S. Shenker, and I. Stoica. Achieving Convergence-Free Routing using Failure-Carrying Packets. In *Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '07, pages 241–252, New York, NY, USA, 2007. ACM.

[46] F. T. Leighton and B. M. Maggs. Fast Algorithms for Routing Around Faults in Multibutterflies and Randomly-Wired Splitter Networks. *IEEE Transactions on Computers - Special issue on fault-tolerant computing*, 41(5):578–587, May 1992.

[47] C. E. Leiserson. Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing. *IEEE Transactions on Computers*, 34(10):892–901, October 1985.

[48] K. Levchenko, G. M. Voelker, R. Paturi, and S. Savage. XL: An Efficient Network Routing Algorithm. In *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, SIGCOMM '08, pages 15–26, New York, NY, USA, 2008. ACM.

[49] H. Liu. Amazon data center size. http://huanliu.wordpress.com/2012/03/13/amazon-data-center-size/, March 2012.

[50] J. Liu, B. Yan, S. Shenker, and M. Schapira. Data-Driven Network Connectivity. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, HotNets '11, pages 8:1–8:6, New York, NY, USA, 2011. ACM.

[51] J. W. Lockwood, , McKeown, G. Watson, G. Gibb, P. Hartke, J. Naous, R. Raghuraman, and J. Luo. NetFPGA–An Open Platform for Gigabit-Rate Network Switching and Routing. In *Proceedings of the 2007 IEEE International Conference on Microelectronic Systems Education*, MSE '07, pages 160–161, Washington, DC, USA, 2007. IEEE Computer Society.

[52] Microsoft Corporation. Windows Azure. https://www.microsoft.com/windowsazure, 2012.

[53] J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, A. R. Curtis, and S. Banerjee. DevoFlow: Cost-Effective Flow Management for High Performance Enterprise Networks. In *Proceedings of the Ninth ACM SIGCOMM Workshop on Hot Topics in Networks*, HotNets '10, pages 1:1–1:6, New York, NY, USA, 2010. ACM.

[54] J. Moy. OSPF version 2. RFC 2328, IETF, 1998.

[55] J. Mudigonda, P. Yalagandula, M. Al-Fares, and J. C. Mogul. SPAIN: COTS Data-Center Ethernet for Multipathing over Arbitrary Topologies. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, NSDI' 10, pages 18–18, Berkeley, CA, USA, 2010. USENIX Association.

[56] R. N. Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. PortLand: A Scalable Fault-Tolerant Layer 2 Data Center Network Fabric. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, SIGCOMM '09, pages 39–50, New York, NY, USA, 2009. ACM.

[57] J.-H. Park, H. Yoon, and H.-K. Lee. The Deflection Self-Routing Banyan Network: A Large-Scale ATM Switch Using the Fully Adaptive Self-Routing and its Performance Analyses. *IEEE/ACM Transactions on Networks (TON)*, 7:588–604, August 1999.

[58] R. Perlman. An Algorithm for Distributed Computation of a Spanning Tree in an Extended LAN. In *Proceedings of the ninth symposium on Data communications*, SIGCOMM '85, pages 44–53, New York, NY, USA, 1985. ACM.

[59] R. Perlman. Rbridges: Transparent Routing. In *Twenty-third Annual Joint Conference of the IEEE Computer and Communications Societies*, pages 1211 – 1218. IEEE, March 2004.

[60] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving Datacenter Performance and Robustness with Multipath TCP. In *Proceedings of the ACM SIGCOMM 2011 conference*, SIGCOMM '11, pages 266–277, New York, NY, USA, 2011. ACM.

[61] T. L. Rodeheffer, C. A. Thekkath, and D. C. Anderson. SmartBridge: A Scalable Bridge Architecture. In *Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '00, pages 205–216, New York, NY, USA, 2000. ACM.

[62] Royal Pingdom. Human errors most common reason for data center outages. http://royal.pingdom.com/2007/10/30/human-errors-most-common-reason-for-data-center-outages/, October 2007.

[63] F. B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computer Surveys (CSUR)*, 22(4):299–319, December 1990.

[64] J. Schneider and R. Wattenhofer. A New Technique for Distributed Symmetry Breaking. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, PODC '10, pages 257–266, New York, NY, USA, 2010. ACM.

[65] M. D. Schroeder, A. D. Birrell, M. Burrows, H. Murray, R. M. Needham, T. L. Rodeheffer, E. H. Satterthwaite, and C. P. Thacker. Autonet: A High-speed, Self-configuring Local Area Network Using Point-to-point Links. *IEEE Journal on Selected Areas in Communications*, 9(8):1318 –1335, October 1991.

[66] H. J. Siegel and C. B. Stunkel. Inside Parallel Computers: Trends in Interconnection Networks. *IEEE Computer Science & Engineering*, 3:69–71, September 1996.

[67] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey. Jellyfish: Networking Data Centers Randomly. In *Proceedings of the 9th USENIX conference on Networked systems design and implementation*, NSDI '12, Berkeley, CA, USA, 2012. USENIX Association.

[68] L. Song and B. Mukherjee. On the Study of Multiple Backups and Primary-Backup Link Sharing for Dynamic Service Provisioning in Survivable WDM Mesh Networks. *IEEE Journal on Selected Areas in Communications*, 26(6):84 –91, August 2008.

[69] J. Touch and R. Perlman. Transparent Interconnection of Lots of Links (TRILL): Problem and Applicability Statement, RFC 5556, May 2009.

[70] P. F. Tsuchiya. The Landmark Hierarchy: A New Hierarchy for Routing in Very Large Networks. In *Symposium proceedings on Communications architectures and protocols*, SIGCOMM '88, pages 35–42, New York, NY, USA, 1988. ACM.

[71] E. Upfal. An O(logN) Deterministic Packet Routing Scheme. In *Proceedings of the twenty-first annual ACM symposium on Theory of computing*, STOC '89, pages 241–249, New York, NY, USA, 1989. ACM.

[72] M. Walraed-Sullivan, R. N. Mysore, K. Marzullo, and A. Vahdat. Brief Announcement: A Randomized Algorithm for Label Assignment in Dynamic Networks. In *Proceedings of the 25th international conference on DIStributed computing*, DISC' 11, pages 326–327, Berlin, Heidelberg, 2011. Springer-Verlag.

[73] M. Walraed-Sullivan, R. N. Mysore, M. Tewari, Y. Zhang, K. Marzullo, and A. Vahdat. ALIAS: Scalable, Decentralized Label Assignment for Data Centers. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 6:1–6:14, New York, NY, USA, 2011. ACM.

[74] Y.-H. Wang and C.-F. Chao. Dynamic Backup Routes Routing Protocol for Mobile Ad Hoc Networks. *Information Sciences: an International Journal*, 176(2):161–185, January 2006.

[75] D. Xu, Y. Xiong, C. Qiao, and G. Li. Failure Protection in Layered Networks with Shared Risk Link Groups. *IEEE Network*, 18(3):36 – 41, May-June 2004.