Practical, Efficient, and Reliable Data Center Communication

Danyang Zhuo

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2019

Reading Committee:

Thomas E. Anderson, Chair

Arvind Krishnamurthy, Chair

Xi Wang

Program Authorized to Offer Degree:
Computer Science and Engineering

University of Washington

Abstract

Practical, Efficient, and Reliable Data Center Communication

Danyang Zhuo

Co-Chairs of the Supervisory Committee:
Professor Thomas E. Anderson
Computer Science and Engineering

Professor Arvind Krishnamurthy
Computer Science and Engineering

Data center communication is a key aspect of cloud computing, as it interconnects all the data center resources. It facilitates resource sharing among servers and has become critical for constructing distributed systems that power today's popular cloud applications, such as databases, transaction systems, and video streaming services.

There are three primary requirements when designing and implementing data center communication: reliability, efficiency, and virtualization. Reliability is important because companies depend on the cloud to run their critical business functions, so even the slightest downtime can result in significant lost productivity and revenue. Efficiency means cloud providers can provide services with fewer resources, and thus can lower costs for customers. Virtualization allows cloud customers to move unmodified applications to the cloud enabling more customers to benefit from the reliability and efficiency of the cloud, while also providing resource multiplexity—multiple customers can use the cloud simultaneously with strong security isolation.

One major trend in data center communication is the rapid increase in bandwidth. To provide high network bandwidth, cloud providers build large-scale optical-based data center networks and use high-speed network interface cards to connect to servers. This

trend poses several challenges in providing a practical, efficient and reliable data center communication system. Data center networks using optical communication technologies are expensive to build and difficult to debug due to gray failures and over-engineering. Providing virtualization support at high speeds incurs high processing overheads due to additional packet handling in operating systems.

This thesis offers new techniques to achieve greater reliability and efficiency for data center communication. Our contribution is the design, implementation, and evaluation of three systems: (1) CorrOpt, a data center network monitoring and failure mitigation system that reduces packet corruption errors by three to six orders of magnitude, (2) RAIL, a network architecture that reduces the total cost of ownership of the data center network by up to 44%, and (3) Slim, an operating system kernel design that reduces the processing overheads of container network virtualization by up to 66%.

TABLE OF CONTENTS

Page

List of Figures iii

List of Tables vi

Glossary vii

Chapter 1: Introduction 1
1.1 Thesis Organization 6

Chapter 2: Background 8
2.1 Physical Network 8
2.2 Routing and Fault Tolerance 11
2.3 Network Virtualization 13

Chapter 3: CorrOpt: Understanding and Mitigating Packet Corruptions in Data Center Network 17
3.1 Extent of Packet Corruption 19
3.2 Corruption Characteristics 21
3.3 Root Causes of Corruption 28
3.4 Mitigating Corruption 33
3.5 Implementation 43
3.6 Evaluation 44
3.7 Future Extensions 51
3.8 Related Work 52
3.9 Summary 53

Chapter 4: RAIL: A Case for Redundant Arrays of Inexpensive Links in Data Center Networks 54

4.1 Optical Links' Performance in the Wild . . . 56
4.2 Reducing Over-engineering and Cost . . . 62
4.3 Overview of RAIL . . . 65
4.4 RAIL in Detail . . . 69
4.5 Implementation . . . 73
4.6 Evaluation . . . 74
4.7 Discussion . . . 87
4.8 Related Work . . . 87
4.9 Summary . . . 88

Chapter 5: Slim: OS Kernel Support for a Low-Overhead Container Overlay Network. . . . 90
5.1 Overheads in Container Overlay Networks . . . 93
5.2 Overview . . . 98
5.3 Design . . . 101
5.4 Implementation . . . 107
5.5 Evaluation . . . 108
5.6 Discussion . . . 119
5.7 Related Work . . . 121
5.8 Summary . . . 122

Chapter 6: Conclusion . . . 124

Bibliography . . . 126

Appendix A: NP-Completeness of Finding the Best Set of Links to Disable in CorrOpt 142

Appendix B: Analysis of Expected Fraction of Good Paths in RAIL . . . 146

Appendix C: Find the Worst Path in Clos Topology in RAIL . . . 148

Appendix D: Over-engineering in 100 Gbps in RAIL . . . 152

LIST OF FIGURES

Figure Number Page

2.1 Network topology for Facebook data centers. 9
2.2 Schematic of an optical transceiver. 10
2.3 Architecture of a container overlay network. 16
3.1 Mean and standard deviation of packets lost per day due to corruption across 15 data center networks, sorted by their size. 20
3.2 Packet loss rate over time for two types of loss events. 23
3.3 Correlation of packet loss rate and traffic rate for two types of loss events. . 24
3.4 Spacial locality for two types of loss events. 26
3.5 Symmetry of packet loss rates for two types of loss events. 27
3.6 Illustration of dirty optical connections. 29
3.7 An example of a dirty connection causing packet corruption. 30
3.8 Illustration of bent fiber cables. 31
3.9 An example of a damaged fiber causing packet corruption. 32
3.10 Example of problems with switch-local checking. 34
3.11 Example of topology pruning. 39
3.12 An example of unsuccessful repair actions on a link. 41
3.13 CorrOpt's system components and workflow. 43
3.14 Total penalty per second of switch-local and CorrOpt when the capacity constraint is 75% for every ToR, for traces taken from two data centers. . . . 45
3.15 Fraction of available paths to the spine for the worst ToRs when the capacity constraint is 75%, for traces taken from two data centers. 46
3.16 Fraction of available paths to the spine for the worst ToRs when the capacity constraint is 50%, for traces taken from two data centers. 47
3.17 Total penalty of CorrOpt divided by switch-local for different capacity constraints, for traces taken from two data centers. 48
3.18 Total penalty ratio of using CorrOpt versus using fast checker alone for the large data center network in our trace. 49

3.19 Penalty ratio comparing impact of CorrOpt repair recommendations on corruption loss. . . . 50
3.20 Example of topology segmentation. . . . 52

4.1 Cumulative distribution function of TxPower and RxPower for different speeds and modes across links in our trace. . . . 58
4.2 Cumulative distribution function of RxPower of 10Gbps multi-mode transceivers across five transceiver manufacturers and ten different data centers. Over-engineering is not limited to one manufacturer or data center. . . . 59
4.3 RxPower variation of individual transceivers over time. . . . 60
4.4 Attenuation of 10 Gbps multi-mode optical links. . . . 61
4.5 Links with low RxPower are uniformly and randomly scattered across switches. 63
4.6 Price and reach of standard transceivers for 10 Gbps, 40 Gbps and 100 Gbps technologies. . . . 64
4.7 Illustration of how stretching transceivers reduces cost. . . . 66
4.8 Example of RAIL's virtual topologies. . . . 68
4.9 Our 10GBASE-SR testbed. . . . 76
4.10 Validation for 40GBASE-SR4 on OM3 fiber. . . . 77
4.11 BER distribution for different link lengths. . . . 79
4.12 Total data center network cost reduction for transceiver stretch on a 3-stage fat tree network (10 Gbps, 40 Gbps) with 512 ToRs assuming 0-500m and 0-1000m uniform link length distribution. . . . 81
4.13 Total data center network cost reduction for transceiver stretch on a 3-stage fat tree network (10 Gbps, 40 Gbps) with 512 ToRs assuming uniform link length distribution. . . . 82
4.14 Link and path packet error rates when oversubscription ratio is 4: (a) cumulative distribution function of LPER. (b) cumulative distribution function of PPER. . . . 83
4.15 CDFs of the fraction of good paths for different ToR-ToR pairs across 50 different topologies. . . . 84
4.16 Normalized flow completion time. . . . 86

5.1 Packet flow in: (a) today's container overlay networks, (b) overlay networks for virtual machines. . . . 91
5.2 CPU utilization under different overlay network setups measured by number of virtual cores used for a single 10 Gbps TCP connection. . . . 96

5.3 Architecture of Slim. . . . 100
5.4 TCP connection setup between a web client and a web server atop Slim. . . . 101
5.5 CPU utilization and breakdown for a TCP connection. . . . 110
5.6 A bar graph of the combined throughput of two Slim containers, with rate limit and access control policy updates to one of the containers. . . . 111
5.7 Throughput and latency of Memcached with Weave (in various configurations) and with Slim. . . . 112
5.8 Distribution of latency for Memcached SET and GET operations, illustrating tail latency effects. . . . 113
5.9 CPU utilization of Memcached client and server. . . . 114
5.10 CPU utilization breakdown of Nginx. . . . 115
5.11 Latency of Nginx server. . . . 116
5.12 CPU utilization of PostgreSQL and Kafka. . . . 117
5.13 Latency of PostgreSQL and Kafka. . . . 118
A.1 Reduction from 3-SAT: Each clause C is represented by a ToR switch, the literals X and $\neg X$ of each variable are represented by fabric switches. . . . 145
D.1 Figure D.1a shows the reach test for 100G-SR4. . . . 153

LIST OF TABLES

Table Number Page

2.1 Comparison of container networking implementation options. 14

3.1 Comparison of the normalized distribution of links with congestion and corruption loss for different loss buckets. 22

3.2 Summary of root causes of corruption, their symptoms and their relative contribution in our data centers. 34

4.1 Total data center network cost breakdown for 512 ToR, 512 fabric and 256 spine switches organized into a Clos network with 10 Gbps technology. . . . 80

4.2 Optical technologies' maximum reach under different degrees of stretch. . . 82

5.1 Throughput and latency of a single TCP connection on a container overlay network, compared with that using host mode. 95

5.2 TCP throughput and latency (round-trip time for 32-byte TCP packets) for different packet steering mechanisms atop a container overlay network across two physical hosts. 98

5.3 Time to migrate a Memcached container. 119

GLOSSARY

ATTENUATION: The reduction of signal strength.

BIT ERROR RATE: The rate at which errors occur in the transmission of digital data.

CLOUD COMPUTING: The delivery of computer system resources through the Internet.

CONGESTION: Reduced quality of service when a network is asked to carry more data than it can handle.

CONTAINER: A standard unit of cloud computing that packages all the code needed to run an application in the cloud.

CUMULATIVE DISTRIBUTION FUNCTION: A cumulative distribution function of a random variable X, evaluated at x, is the probability of $X \leq x$.

DATA CENTER: A large-scale group of computers run by a single entity.

DATA CENTER NETWORK: A computer network that connects computing resources in a data center.

DISPERSION: The distortion of signal shape.

FABRIC SWITCH: A switch that connects a set of top-of-rack switches in order to provide communication across racks. In a typical 3-stage data center network, the fabric switches serve as the 2nd stage.

FORWARD ERROR CORRECTION: A technique that uses error-correcting code to control transmission errors over unreliable connections.

MULTI-MODE FIBER: A fiber whose core diameter is typically 50–100 micrometers. It is cheaper than single-mode fiber, but signals suffer more dispersion.

NETWORK INTERFACE CARD: A computer hardware that connects a computer to a computer network.

NETWORK VIRTUALIZATION: The ability to define logical, virtual networks in a data center, using network addresses that are decoupled from these of the underlying physical network.

OPERATING SYSTEM: System software that manages hardware, isolates applications and containers, and provides key services to applications.

PACKET CORRUPTION: A packet loss because the receiver cannot correctly decode transmitted bits.

POD: A set of server racks connected by fabric and top-of-rack switches.

REMOTE DIRECT MEMORY ACCESS: The ability to read and write memory on a remote machine without interrupting its central processing unit.

SINGLE-MODE FIBER: A fiber whose core diameter is typically 8–10.5 micrometers. It is more expensive than multi-mode fiber, but signals suffer less dispersion.

SINGLE-ROOT INPUT/OUTPUT VIRTUALIZATION: A hardware functionality that allows isolation of its resources.

SOCKET: A standard application programming interface for data communication.

SPINE SWITCH: A switch that connects a set of fabric switches to provide connectivity across different pods. In a typical 3-stage data center network, the spine switches serve as the 3rd stage.

VIRTUAL MACHINE: Emulation of a physical computer.

TOP-OF-RACK SWITCH: An in-rack switch that servers within the same rack connect to. In a typical 3-stage data center network, the top-of-rack switches serve as the 1st stage.

TRANSCEIVER: A device that both transmits and receives data using optical fiber technology.

ACKNOWLEDGMENTS

The past six years at the University of Washington have been an invaluable journey for me. This journey was challenging, and I would not have been able to finish it without the help and support I received from my mentors, friends, and family.

My greatest thanks go to my advisors, Tom Anderson and Arvind Krishnamurthy. It has been a great honor and privilege to work with these two brilliant computer scientists. I have benefited substantially from their wisdom and insights about how to generate research ideas and execute research. Tom has always encouraged me to think big and hold myself to a higher standard. Arvind's optimism about research and life has cheered me on throughout my time in graduate school.

I also want to thank my mentors at Microsoft Research: Hongqiang Harry Liu, Monia Ghobadi, Ratul Mahajan, and Yibo Zhu. I was fortunate enough to spend 1.5 years doing research with them. They have greatly shaped my research taste.

I am grateful to Vincent Liu and Dan Ports for helping me get oriented during the first two years of my Ph.D. journey. I would like to thank Jiangbo Li and Junlan Zhou for mentoring me during my Google internship, which gave me the first comprehensive view of how a data center works.

I would like to thank my closest student collaborator, Kaiyuan Zhang. We find great synergy working on research together. Kaiyuan can always help me find a way to make our research prototypes run on Linux.

I wish to acknowledge my other collaborators, Klaus-Tycho Förster, Hang Guan, Samantha Miller, Simon Peter, Amar Phanishayee, Matthew Rockett, Shibin Xu, Xin Yang, Qiao Zhang, and Xuan Kelvin Zou, for their help on research. I have learned a lot from them.

I want to thank the following faculty members, postdocs, and students in the systems lab for making my life enjoyable and providing feedback on my research: Ravi Bhoraskar, Lequn Chen, Raymond Cheng, Tianyi Cui, Pedro Fonseca, Seungyeop Han, Yuchen Jin, Antoine Kaufmann, Ed Lazowska, Nei Lebeck, Hank Levy, Jialin Li (the elder), Jialin Li (the younger), Ming Liu, Ashlie Martinez, Ellis Michael, Samantha Miller, Luke Nelson, Simon Peter, Henry Schuh, Will Scott, Naveen Sharma, Haichen Shen, Helgi Sigurbjarnarson, Adriana Szekeres, Xi Wang, Xiao Sophia Wang, Irene Zhang, Kaiyuan Zhang, Qiao Zhang, and Kevin Zhao.

I am most grateful to my parents, Junfei and Meiping, for their unconditional love and support. My girlfriend, Baolu, has been my constant source of inspiration.

DEDICATION

to my parents, Junfei and Meiping

Chapter 1

INTRODUCTION

Cloud computing has become a dominant computing paradigm, used by billions on a daily basis. Today, our most data-intensive applications—-such as file storage [6], video streaming [18], and social networking [8, 22]—-are hosted in data centers and consumed over the Internet. Applications that were previously deployed on local servers [17, 128] are now moving to the cloud [14, 13], as well.

Communication is a key component of data center design. It interconnects all computing and storage resources and enables different components of a distributed cloud application to communicate with each other.

To build communication systems in a data center, a cloud provider has to solve many practical problems. The first is to determine how to build a hardware networking infrastructure for the data center. The cloud provider purchases networking devices (e.g., switches, network links) and installs them to connect the servers. Second, the provider must properly manage these networking devices and prepare for their failures. Finally, the cloud provider needs to build solutions that let disparate cloud customers use the communication system simultaneously.

When solving these practical problems, the primary requirements are *practicality*, *cost-efficiency*, *reliability*, and *virtualization*. Practicality is one of the most important requirements. A cloud provider wants a practical design that is immediately deployable. This means the hardware network infrastructure can be assembled, using existing commodity networking hardware. Commodity networking devices are standardized [24] and manufactured at a large scale. Commodity devices from different manufacturers can interoperate with each other, making installation flexible, and they are in general cheaper than

customized hardware. Another aspect of practicality is the application interface for data communication. Today, most Linux applications use POSIX Sockets. A data center communication system that supports POSIX Socket is a necessary pre-requisite to running unmodified Linux applications in the cloud.

Cost-efficiency is another goal for data center communication design. Delivering data in the data center communication system requires resources (e.g., hardware spending, energy, and processing cycles). Cost-efficiency means how much resources need to be consumed in order to deliver data. It can also be measured by how much data can be delivered given the same amount of resources. High cost-efficiency allows cloud applications to communicate at a low cost.

Reliability is important because companies depend on the cloud to run critical businesses, for which even the slightest downtime results in lost productivity and revenue. Failures of hardware devices (e.g, switches, network links) are hard to eliminate completely, but data center communication is reliable as long as a cloud provider can mask or mitigate the failures before they exhibit customer visible effects.

Finally, cloud providers must provide each cloud customer with a virtualized/isolated view of the data center network. Network virtualization can recreate an application's desirable operating environment so that customers can move their applications using their designated network addresses to the cloud easily. Virtualization also enforces strong security isolation between different customers' networks, preventing network attacks from other customers in the same data center.

To achieve greater cost-efficiency and reliability, a cloud provider carefully selects the most cost-efficient hardware solutions to build a physical network that connects all servers. The provider also tries to drive the communication to higher utilization to fully utilize its hardware resources in a data center. The provider then carefully considers possible hardware failures and how to design mechanisms that mitigate them quickly, providing reliability to cloud applications. For example, to tolerate a link failure, the cloud provider can build a physical network with redundant paths between servers and design

and implement fault-tolerant routing in the physical network.

One major trend in data center communication is the need for high bandwidth. Google's data center network traffic doubles every 12-15 months [139], and Cisco predicts that total data center network traffic will grow from 5.13 in 2016 to 14.73 zettabytes in 2021 [12]. The need for higher bandwidth leads to two implications: (1) the deployment of large-scale optical-based data center networks, and (2) the usage of high-speed network interface cards (NICs) in the servers. Optical communication technologies have proven their value in delivering high bandwidth, low latency networking between servers. Today, data centers almost exclusively use optical communication technologies for their core networks. On the server side, cloud providers are moving from 1 Gbps NICs to 10 and 40 Gbps NICs, and now even deployments of 100 Gbps NICs are common. Futher link speed improvements are likely.

This trend poses several challenges for cloud providers to build practical, cost-efficient, and reliable data center communication systems. Optical communication technologies are expensive because links are engineered using conservative practices. The existing design objective is that an optical link in a data center network must be reliable (i.e., bit error rate $< 10^{-12}$) assuming the worst-case operating environment (e.g., temperature, signal attenuation).

Because a large-scale optical-based data center network consists of hundreds of thousands of optical network links, mitigating and diagnosing hardware-related gray failures are notoriously difficult. A cloud provider cannot simply turn off networking hardware with gray failures because doing so could significantly degrade network performance. For diagnosis, the true root cause of a failure (e.g., faulty switches, poor hardware installation, damaged optical fibers, dirty optical connectors) is hard to identify, and incorrect repair obviously cannot mitigate the failure.

Finally, providing network virtualization incurs substantial processing overheads. Today's network virtualization solutions are based on packet encapsulation, which requires each packet to be transformed whenever it is sent or received. Extra packet transforma-

tions reduce overall network throughput and increase packet latency. Packet transformations also waste considerable processing cycles, particularly at high network speeds.

Given these challenges, in this thesis, we show that **it is possible to achieve higher levels of reliability and cost-efficiency using commodity networking hardware while supporting optical-based networks and network virtualization.** This thesis offers several techniques to achieve this goal by addressing three specific problems:

How to mitigate packet corruption errors in the data center network? Packet losses in data center networks degrade application performance and can lead to millions of dollars of lost revenue. Even a packet loss rate of 0.01% can slow user-facing network traffic by 50% [46], significantly impacting the user experience. Researchers have explored a wide range of approaches to minimize packet loss, such as congestion control, load balancing, and traffic engineering [29, 30, 31, 57, 122, 142, 145, 152]. These approaches all focus on congestion loss only, i.e., packet loss when the network is overloaded. Another source of packet loss, packet corruption, has received little attention.

Working with Microsoft, we studied 350K links across 15 production data centers to show that packet corruption poses a significant problem, and its characteristics differ markedly from congestion losses. Corruption impacts fewer links but imposes higher loss rates. Also, the corruption rate is stable over time, weakly correlated with link utilization. This implies that reducing the load on network links (e.g., congestion control, load balancing) does not reduce the packet corruption rate.

To address this problem, we designed and implemented CorrOpt [154], a system that automatically detects and mitigates packet corruption. CorrOpt minimizes corruption losses by intelligently disabling corrupting links while ensuring the network has sufficient capacity for its current workload. It also recommends specific maintenance actions (e.g., replacing equipment, cleaning cable connectors) to repair disabled links. Our evaluation shows CorrOpt can reduce corruption loss by three to six orders of magnitude.

How to reduce the cost of the physical network layer by leveraging network path redundancy? Network cost is an important metric for data center network design. Costs include those for all the main contributors, such as switches, links, and power. New data center network architectures have been proposed to reduce the cost of networking, including topology design, congestion control, and load balancing [27, 76, 139, 32, 43, 108, 79]. However, the physical layer (i.e., optical fiber) is often ignored.

Using the same infrastructure as we did for CorrOpt, we monitored optical links across Microsoft's production data centers for more than 10 months. We found a remarkably conservative state of affairs: 99.9% of links had an incoming optical signal quality that exceeded the minimum threshold. Even the median was 6 times higher. This overengineering is expensive because transceivers (i.e., devices that convert signals between electrical and optical domains) account for 48–72% of the total data center network cost depending on link length distribution. Motivated by this observation, we propose to lower network cost by using transceivers beyond their specified limit. Our benchmark shows that the reach of multiple commodity transceivers can be stretched by 1.6-4 times their specification. However, 1–5% of data center network paths will suffer significant packet loss.

To address this problem, we designed and implemented RAIL [155], a system that ensures applications use only network paths that meet their reliability requirements. RAIL generates multiple virtual topologies on the same physical network topology. Each topology guarantees a maximum packet loss rate. Applications bind to different topologies based on their needs. Our evaluation shows that RAIL reduces data center network cost by 10% for a 10 Gbps network and 44% for 40 Gbps network.

How to provide low-overhead network virtualization for containers? Containers have become the mainstream method for hosting large-scale distributed applications. They are attractive because they are portable, allowing application code and configuration to run everywhere in the same way. Also, container virtualization is more lightweight than

machine virtualization because an OS virtualizes its resource at the system call level (for containers) rather than emulating the raw hardware (for virtual machines). The container overlay network is key to supporting distributed containerized applications. It allows containerized distributed applications to have their own network configurations, fully decoupled from that of the host network.

Unfortunately, today's container overlay network implementations impose significant overhead in terms of throughput, latency, and CPU utilization. They use packet encapsulation for network virtualization. This means that each packet must traverse the OS network stack twice and also traverses a virtual switch on both the sender and receiver sides. This design resembles an overlay network architecture used for virtual machines. Because the virtual machine has unilateral control of its network stack, the hypervisor must send/receive packets without the context of network connections. However, the OS kernel has full knowledge of network connections that are created inside containers.

We designed and implemented a solution called Slim [156] to virtualize Linux container networking at connection setup time by manipulating connection metadata; it is compatible with existing applications and requires minimal kernel changes. Slim achieves lower overhead because packets go through the network stack only once. Our evaluations show that Slim improves an in-memory key-value store by 66% and reduces its latency by 42%. It reduces CPU utilization of the in-memory key-value store by 54%. Slim also reduces CPU utilization of a web server by 28–40%, a database server by 25%, and a stream processing framework by 11%.

1.1 Thesis Organization

The remainder of the thesis begins with an overview of data center communication, covering the physical data center network, routing, and required system software for network virtualization in Chapter 2. In Chapter 3, we study the problem of packet corruption in the data center network. We present the design and implementation of CorrOpt to mitigate such corruption errors. Chapter 4 presents RAIL, a low-cost data center network

architecture that uses redundancy to mask under-performing network links. Chapter 5 presents Slim, an OS kernel design for a low-overhead container overlay network. We conclude in Chapter 6.

Chapter 2
BACKGROUND

Data center communication is a foundational element of cloud computing. It consists of three parts: (1) *a physical data center network*, including all networking hardware (e.g., switches, network links), (2) *a management software layer*, which implements routing and failure tolerance, and (3) *a system software solution*, which virtualizes the network for cloud applications.

2.1 Physical Network

The physical network is the hardware that supports data center communication. A large modern data center network connects thousands of racks of servers using tens of thousands of networking devices, such as switches, network links, and network interface cards (NICs).

Servers are arranged in racks for easy deployment and installation. Each rack can hold tens to hundreds of servers and has a *top-of-rack* (ToR) *switch* (also known as *tier-1 switch* or *rack switch*), which connects all servers in the rack together and to the rest of the network. Each server has a network link that connects it to the ToR switch. Communication between servers within a rack goes through a simple NIC-ToR-NIC path. A network link between a NIC and a ToR switch is usually made of copper, which is cost-effective at short distances (within a rack).

To communicate between servers in different racks, a data center network (DCN) is organized as a multi-stage, multi-rooted tree network where ToR switches are the leaf nodes [27, 76, 139]. Figure 2.1 shows the 3-stage tree topology deployed in Facebook's data centers. *Fabric switches* (also known as *tier-2 switches* or *cluster switches*) are the switch

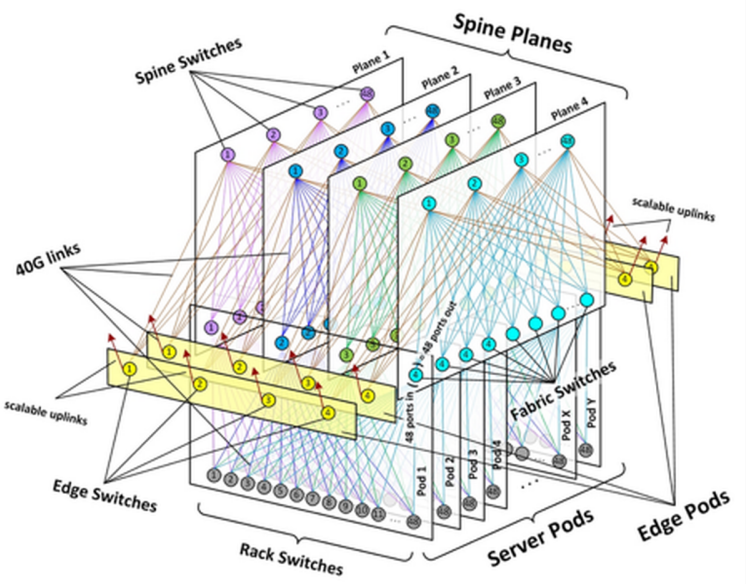


Figure 2.1: Network topology for Facebook data centers [81]. It is a multi-root, 3-stage tree network. The first, second, and third stage consist of rack switches, fabric switches, and spine switches, respectively. Each rack switch is connected to a rack of servers. A pod consists of 48 server racks and 48 rack switches and 4 fabric switches. Fabric switches are connected to rack switches to provide connectivity across racks within a pod. A plane contains up to 48 spine switches and a set of fabric switches. Spine switches are connected to fabric switches to provide connectivity across pods. Edge switches connect the spine switches of the data center to an external network.

stage above ToR switches. A cloud provider uses a *pod* to define a set of server racks that are connected to the same set of fabric switches. Fabric switches provide connectivity with a pod across different racks. Each ToR connects to multiple fabric switches for high network capacity and fault tolerance. *Spine switches* (also known as *tier-3 switches*) cross-connect fabric switches to provide connectivity across different pods. Overall, servers on different racks communicate through either a NIC-ToR-Fabric-ToR-NIC path or a NIC-ToR-Fabric-Spine-Fabric-ToR-NIC path depending on whether they belong to the same

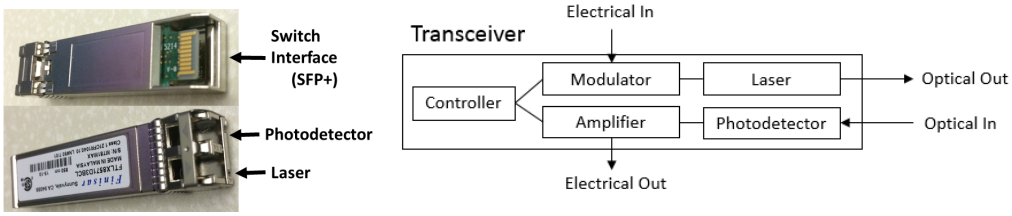


Figure 2.2: Schematic of an optical transceiver. A transceiver has two sides. The electrical side interfaces to the switch while the optical side interfaces two fibers—one with a laser and one with a photodetector. The transceiver has a modulator and a laser to encode bits into an optical signal. It also has a photodetector and an amplifier to decode an optical signal back to bits.

pod. Edge switches connect the spine switches to the external network.

A network link between two switches in modern data centers is almost always an *optical fiber link*. An optical link consists of two transceivers, i.e., devices that transform signal between optical and electrical forms, and two fibers. Figure 2.2 shows a transceiver and its internal components; it has a transmit and a receive pipeline, each attached to an independent fiber. The transmit side has a laser that emits light and a modulator that modulates it according to the electrical input from the switch (the bits to be encoded). The receive pipeline decodes incoming optical signals using a photodetector and an amplifier. When a switch sends data to another switch over an optical link, the sender side's transceiver first encodes the data into an optical signal via modulation. The optical signal traverses the fiber, and the receiver side's transceiver transforms the optical signal back to an electrical form.

The performance of an optical link is quantified using a *bit error rate* (BER), the probability of a single bit being corrupted at the receiver. Corruption happens when the optical power gap between bits 0 and 1 is too small to be reliably differentiated at the receiver. The gap can be low because of poor signal characteristics at the transmitter (e.g., a poor-

quality laser that spreads light over a wide spectrum) or because of attenuation and dispersion as light travels through fiber. *Attenuation* refers to a reduction in average signal power, e.g., caused by the optical connectors attaching fibers to transceivers. *Dispersion* is distortion of the signal's shape.

To simplify manufacturing and deployment, IEEE standardizes transceiver characteristics. Apart from wavelength (e.g., 850, 1310, 1510 nm) and transmission speed (e.g., 10, 40, 100 Gbps), an important characteristic of each standard is the maximum reach at which a transceiver is guaranteed to deliver a BER of 10^{-12} or better.

Two main categories of optical technologies are used in data center networks (DCNs) today: multi-mode and single-mode. *Multi-mode transceivers* are cheaper because they have relaxed constraints on laser spectral width and the coupling of laser with fiber. But they also have a shorter reach because they suffer from modal dispersion, i.e., signal distortion due to differing path lengths taken by light waves in the fiber.

From different optical communication technologies, a cloud provider picks the most cost-efficient way to connect switches in the data center. The provider considers both the required transmission speed and the reach limit. Optical communication technologies are discrete in their reach distance. For example, IEEE has standardized two types of 40 Gbps technologies: 100m 40GBASE-SR4 uses 850nm wavelength with multi-mode fiber and 10km 40GBASE-LR4 1310nm wavelength with single-mode fiber. Because these optical technologies are based on different physical design and implementation choices (e.g., laser technologies), their reach limits are highly different. To connect two switches that are 200 meters away in a 40 Gbps link, the cloud provider would pick the 10km 40GBASE-LR technology because the 100m 40GBASE-SR4 technology cannot reach far enough.

2.2 Routing and Fault Tolerance

A server must first name the destination server to send data to it. Data centers today use layer-3 routing (i.e., routing based on IP address). Each physical server has a unique IP

address assignment, and the entire physical network's routing tables are configured to deliver network packets based on IP addresses.

With the multi-rooted tree topology shown in Figure 2.1, two servers on different racks have multiple paths to choose from even when restricted to shortest path routing. To leverage the redundant paths for higher network throughput, data center networks can use Equal Cost Multipath (ECMP) routing to pick one of the shortest paths for each server pair. In ECMP, switches compute a hash value based on the 5 tuple (i.e., sender IP address, sender port, network-layer protocol, receiver IP address, receiver port) of a packet header and then use the hash value to decide the path on which to place the packet. Note that the hash function is deterministic, placing every packet in the same connection (i.e., having the same 5 tuple) on the same path to prevent packet reordering at the receiver side.

While the data center network decides the path of a connection, servers control the amount of traffic they want to send into the network. To prevent excessive traffic, servers use congestion control mechanisms, such as TCP and its variants (e.g., CUBIC [78]). These protocols detect congestion via either out-of-order packets or packet latencies. When congestion is detected, congestion control slows the transmission rate, reducing network load and preventing packet loss. Congestion losses are a well-known problem, and many improvements have been made in congestion control, active queue management, load balancing, and traffic engineering [29, 30, 31, 57, 122, 142, 145, 152, 102].

In a real deployment, multiple types of failures or other management issues can occur in data centers; thus, routing has to be dynamic. Failures can be classified into two broad categories: fail-stop failures and gray failures. *Fail-stop failures* are easy to detect: if a link or switch fails, neighboring networking devices quickly notice because the failed device can no longer send to them. To mitigate fail-stop failures, the data center network must modify its routing behavior to route around any failed devices. Today, dynamic fault-tolerant routing can be achieved through either a distributed route management protocol (e.g., Border Gateway Protocol [97]) or a centralized routing table computation [108, 151].

Gray failures occur when a fraction of ongoing data packets are dropped on a switch

or link. Detecting this type of failure is harder because the neighboring device can still detect that the failed device is alive. Therefore, the cloud provider must monitor network performance metrics (e.g., packet loss rates on all links); gray failures reduce network performance and can cause application-level services to crash [34]. To mitigate them, a network operator first diagnoses the root cause (e.g., faulty switches, poor hardware installation, damaged optical fibers, dirty optical connectors) and then applies repairs, accordingly.

2.3 Network Virtualization

Virtualization is a key feature clouds provide. It enables applications that traditionally run on a private server to run in a shared server infrastructure. Currently, there are two main approaches to virtualization, a container and a virtual machine. Both provide cloud customers the illusion that their applications are running on dedicated servers. In this thesis, we focus on containers.

Containers typically have four options for communication: bridge mode, host mode, macvlan mode, and overlay mode (similar to how network virtualization works for virtual machines). Table 2.1 compares the different modes in terms of the IP addresses used by containerized applications and routing in the host network. *Bridge mode* is used exclusively for containers communicating on the same host: each container has an independent IP address, and the OS kernel routes traffic between different containers.

How can the cloud provider enable communication between containers on different hosts? With *host mode*, containers directly use the IP address of their host network interface. The network performance of host mode is close to the performance of any process that directly uses the host OS's network stack. However, host mode creates many management and deployment challenges. First, containers cannot be configured with their own IP addresses; they must use the IP address of the host network interface. This complicates porting: distributed applications must be re-written to discover and use host IP addresses, and, if containers can migrate (e.g., after a checkpoint), the application must

Mode	Applications use	Routing uses
Bridge	Container IP	–
Host	Host IP	Host IP
Macvlan	Container IP	Container IP
Overlay	Container IP	Host IP

Table 2.1: Comparison of container networking implementation options. The first column is the container communication mode. The second column is the IP addresses used in the application, and the third column is the IP address used for routing in the host network.

be able to adapt to dynamic changes in their IP address. Worse, because all containers on the same host share the same host IP address, only one container can bind to a given port (e.g., port 80), resulting in complex coordination between different applications running on the same host. In fact, container orchestrators, such as Kubernetes, do not allow usage of host mode [96] due to these issues.

Macvlan mode or similar hardware mechanisms (e.g., SR-IOV) let containers have their own IP addresses different from their hosts. Macvlan or SR-IOV let the physical NIC emulate multiple NICs each with a different MAC address and IP address. Macvlan[1] extends the host network into the containers by making the container IP routable on the host network. However, this approach complicates data center network routing. Assume that a distributed application with IP addresses IP 1.2.3.[1-10] is not co-located on the same rack, or that it is initially co-located but some containers are then migrated. In this case, the host IP addresses will not be contiguous, e.g., one might be on host 5.6.7.8 and another on host 9.10.11.12. Macvlan requires the cloud provider to change its core network routing to redirect traffic with destination IP 1.2.3.[1-10] to 5.6.7.8 and 9.10.11.12, potentially

[1]There are software approaches (e.g., Calico [45]) to extend the host network into containers. They have the same problem as macvlan.

requiring a separate routing table entry for each of the millions of containers running in the data center. Further, containers must choose IP addresses that do not overlap with the IP addresses of any other container (or host). Because of these complications, most cloud providers currently block macvlan mode [103].

To avoid interference with routing on the host network, the popular choice is to use *overlay mode*. This is the analog of a virtual machine but for a group of containers—each container has its own network namespace with no impact or visibility into the choices made by other containers or the host network. A virtual network interface (assigned an IP address chosen by the application) is created for each container. It is connected to the outside world via a virtual switch (e.g., Open vSwitch [124]) inside the OS kernel. Overlay packets are encapsulated with host network headers when routed on the host network. This lets the container overlay network have its own IP address space and network configuration that is disjoint from that of the host network; each can be managed completely independently. Many container overlay network solutions are available today—such as Weave [144], Flannel [65], and Docker Overlay [54]—all of which share similar internal architectures.

Figure 2.3 presents a high-level system diagram of a container overlay network that uses packet transformation to implement network virtualization. It shows an OS kernel and a container built with namespaces and cgroups. *Namespace isolation* prevents a containerized application from accessing the host network interface. *Cgroups* allow fine-grained control over the total amount of resources (e.g., CPU, memory, and network) that the application inside the container can consume.

The key component of a container overlay network is a *virtual switch* inside the kernel. This switch has two main functions: (1) *network bridging*, allowing containers on the same host to communicate, and (2) *network tunneling*, enabling overlay traffic to travel across the physical network. The virtual switch is typically configured using the Open vSwitch kernel module [124] with VXLAN as the tunneling protocol.

To enforce various network policies (e.g., access control, rate limiting, and quality of

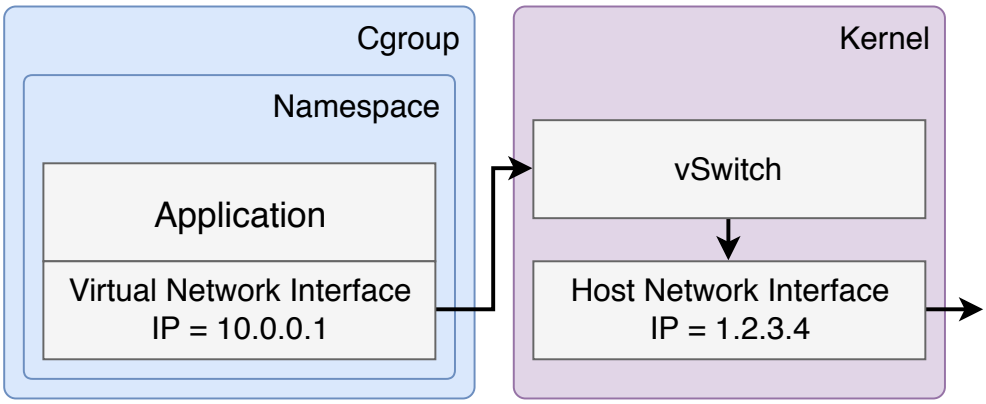


Figure 2.3: Architecture of a container overlay network. A container is constructed by cgroups and namespaces. Cgroups and namespaces are functionalities in Linux to isolate resources. An application runs inside a container with a virtual network interface. The OS kernel controls a virtual switch and a host network interface. Outgoing packets go through the virtual network interface, the virtual switch, and the host network interface.

service), a network operator or a container orchestrator [96, 55, 80] issues policy updates to the virtual network interface or the virtual switch. For example, firewall rules are typically implemented via *iptables* [83], and rate limiting and quality of service (QoS) can also be configured inside the Open vSwitch kernel module. These rules are typically specified in terms of the application's virtual IP addresses rather than the host's IP addresses, which can change depending on where the container is assigned.

The hosts running a set of containers in an overlay network must maintain a consistent global network view (e.g., virtual to physical IP mappings) across hosts. They typically do this using an external, fault-tolerant distributed datastore [61] or gossiping protocol.

Chapter 3

CORROPT: UNDERSTANDING AND MITIGATING PACKET CORRUPTIONS IN DATA CENTER NETWORK

This chapter focuses on understanding packet corruption in data centers. Packet corruption occurs when the receiver cannot correctly decode transmitted bits. Such decoding errors are detected by the cyclic redundancy check in the Ethernet frame to fail, causing the receiver to drop the packet. Packets may also be lost due to other reasons, such as congestion. While recent studies categorize different sources of packet loss and acknowledge packet corruption as a contributor [149, 35, 153], not much is known today about the extent and characteristics of corruption-induced packet loss. To improve mitigation techniques for packet corruption, we need a thorough understanding of its characteristics.

We present the first large-scale study of packet corruption in data center networks. We monitor 350K switch-to-switch, optical links within 15 data centers of Microsoft, over seven months. We find that the number of packets lost due to corruption is significant. We uncover several relevant characteristics of corruption losses and contrast them with those of congestion. For instance, while the loss rate due to congestion varies with link utilization, that due to corruption is relatively stable over time and is independent of the link's utilization. This observation implies that reducing the load on the link, as in congestion control, will not reduce the packet corruption rate. We also find that, compared to congestion, corruption plagues fewer links but imposes higher loss rates on those links. Finally, we find that corruption exhibits weak locality, i.e., the chances of multiple corrupting links being on the same switch or being topologically close are noticeable but low, while congestion exhibits strong locality.

We also analyze hundreds of trouble ticket logs to find the common root causes of

corruption. These range from faulty transceivers (i.e., devices that convert between optical and electrical signals) and switches, to poorly installed hardware, to damaged optical fiber, to dirty optical connectors. By monitoring the optical layer contemporaneously with the tickets, we uncover the common symptoms for each root cause.

The prevalent method to mitigate corruption is to disable links with corruption loss rate above a certain level (e.g., 10^{-6}), provided that the switches to which they attach have at least a threshold number of active uplinks toward the spine of the data center network [104]. This threshold ensures that the hosts using the switch have enough leftover capacity—otherwise, we might replace corruption losses with heavy congestion losses. Links are disabled automatically using software that monitors the corruption loss rate of each link. Though it does not *repair* corrupting links, this software is important because it reduces the chances of application traffic experiencing corruption losses. For each disabled link, a maintenance ticket is issued for operators to manually repair the link. The operators attempt to repair the link via a sequence of steps (e.g., clean the optical fiber and connectors; replace the transceiver; replace the cable), based on their expertise and largely independent of the root cause. The link is enabled after each step, and the next step is taken if the previous one did not succeed at eliminating corruption.

The method above has two limitations. First, the criterion for disabling links is greedy and local. While such decisions can be made quickly, they miss better opportunities that can reduce the level of corruption losses, i.e., disable links with higher corruption rates or disable those with more alternative routes. We show that such opportunities exist while meeting the same capacity constraint. Second, since the strategy to repair corruption is agnostic of the root cause, it often takes multiple tries before the real root cause is addresses. In fact, with the current strategy, the link is fixed in the first step only 50% of the time.

Based on the observations above, we develop CorrOpt, a system to mitigate corruption in data center networks. Because the problem of identifying the optimal set of corrupting links to disable, which minimizes corruption losses while meeting capacity constraints,

is NP-hard, CorrOpt uses a two-phase approach. First, when a link starts corrupting packets, a fast decision is made on whether the link can be safely turned off. Even this fast decision allows us to reduce corruption losses relative to the current method because it considers the entire set paths from top-of-rack switches to the spine, instead of just the switches adjacent to the link. But this fast decision is not optimal. To approximate optimality, we use a second phase that does a global optimization to determine the set of links that can be safely disabled. The combination of the two phases allows us to react quickly and optimize later.

CorrOpt also has a recommendation engine that uses a root cause-aware approach to propose the right repair for corrupting links. Based on the link's characteristics (i.e., corruption rate, optical transmit power, optical receive power) and history of actions taken thus far (if any), it generates concrete recommendations for operators on what corrective action is needed. This recommendation engine has been deployed in over 70 Microsoft data centers.

We evaluate CorrOpt using the deployment of the recommendation engine and a trace-based analysis using data from production data center networks. We find that CorrOpt responds to packet corruption quickly and lowers the amount of corruption losses by up to three to six orders of magnitude, while meeting the desired capacity constraints. We also find that our recommendation engine has improved the accuracy of repairing the link at the first attempt from 50% to 80%.

3.1 Extent of Packet Corruption

We demonstrate the need to understand and mitigate packet corruption by quantifying its extent in today's data center networks. We show that the number of packets lost due to corruption is startlingly high. Our analysis considers corruption- and congestion-induced losses in 15 production data centers. We focus on switch-to-switch links. Corruption mitigation, by disabling or routing around corrupting links, is relevant only for such links, not server-to-ToR links. Further, as we discuss later, the complexity of repair is a concern

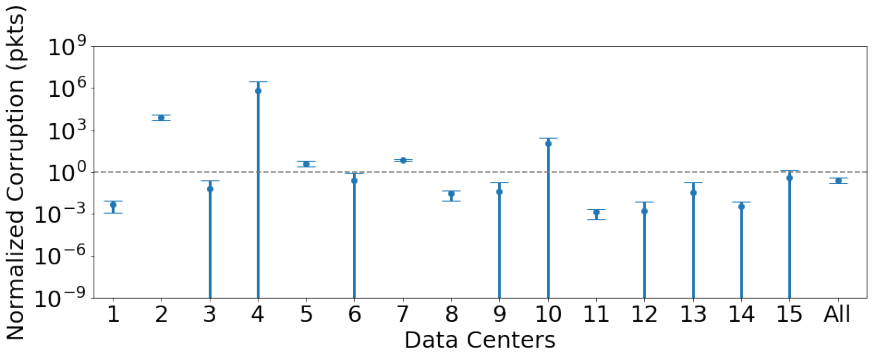


Figure 3.1: Mean and standard deviation of packets lost per day due to corruption across 15 data center networks, sorted by their size. Values are normalized by the mean number of congestion losses in each data center network. Horizontal dashed line denotes the threshold when packets loss due to corruption and congestion are the same.

only for switch-to-switch links, which are optical and can go long distances; server-to-ToR links, which are electrical and short, simply get replaced. As the data centers that we study use standard designs and physical layer technologies (e.g., transceivers, fibers), we expect our findings to be applicable to other data centers as well.[1]

The data center networks in our study have 4–50K links, and the total across all of them is 350K. For each link, we use SNMP [47] to query its packet drop, packet error, and total packet counts, as well as its optical power levels from its switch every 15 minutes. Our network operators found SNMP to be a reliable and lightweight mechanism for monitoring these counters. The data cover a period of seven months. The results in this section are based on three weeks of data, and the next section looks deeply into one representative week. The rest of the chapter uses the entire data set.

Figure 3.1 shows the number of packets lost per day due to packet corruption and congestion. The data center networks are sorted by size. The number of packet losses

[1]The data centers in our data set have diverse ages and have been through different rounds of equipment replacement cycles. We do not have data of when each device is installed and thus cannot study the effect of packet corruption over device lifetime.

on the Y-axis is normalized with respect to mean congestion per data center network. The error bars represent the standard variation around the mean for packets lost due to corruption on different days.

While results vary across data centers and days, in aggregate, corruption losses is a substantial fraction of packet losses. Although this graph does not show corruption *rate*, the next section shows that it can be quite high for some links. This high level of corruption loss happens even though there is already a system to discover and turn off links with corruption. While this system has limitations, which we explain in Chapter 3.4, we estimate that without it, corruption-induced losses would be two orders of magnitude higher.

Our results clearly demonstrate the need for an effective strategy to mitigate corruption in data center networks. Our proposed system, CorrOpt, provides such a strategy. To explain the rationale underlying its design, in the next two sections, we dig more deeply into the nature of corruption and its root causes.

3.2 Corruption Characteristics

To develop a thorough understanding of packet corruption, in this section we identify the characteristics of corruption and compare them to congestion. Though not our focus, our observations can also help load balancing and congestion control systems appropriately handle congestion versus corruption losses when switch counters are not available to distinguish the two.

Corruption impacts fewer links but can be more severe than congestion. Our data reveal that while congestion is a more widespread phenomenon in terms of the links it impacts, packet corruption affects fewer links. We compute the percentage of links with congestion and corruption loss rate above 10^{-8} and find that the total number of links

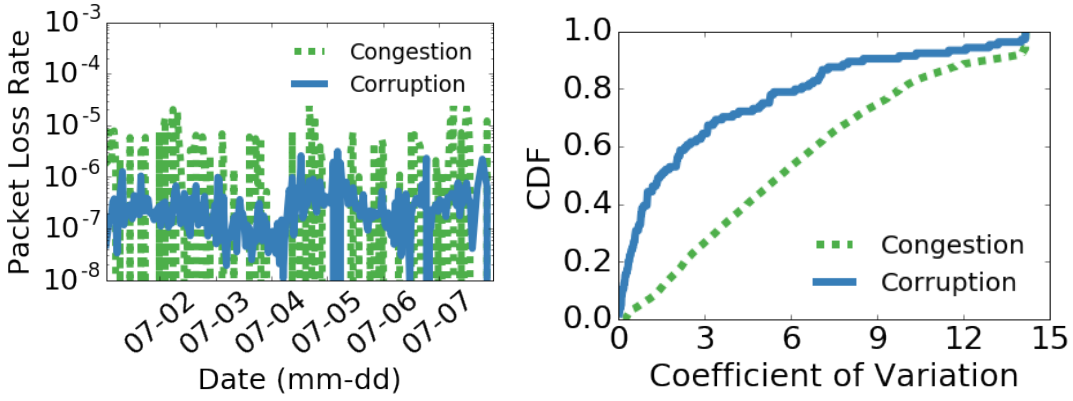
Loss bucket	links w. corruption	links w. congestion
$[10^{-8}$ - $10^{-5})$	47.23%	92.44%
$[10^{-5}$ - $10^{-4})$	18.43%	6.35%
$[10^{-4}$ - $10^{-3})$	21.66%	0.99%
$[10^{-3}+)$	12.67%	0.22%
total	100%	100%

Table 3.1: Comparison of the normalized distribution of links with congestion and corruption loss for different loss buckets. For example, 12.67% of total links that experience corruption, have a corruption loss rate greater than or equal to 10^{-3} (0.1% loss rate) whereas only 0.22% of links with congestion, have congestion loss rate of 10^{-3}. The numbers in each column are normalized so that the table does not reveal the overall percentage of links with congestion or corruption losses for confidentiality reasons.

with corruption is about 2-4% of those with congestion.[2]

This difference suggests that a small set of links have a high corruption loss rate, given that the aggregate number of corruption and congestion losses is similar. Table 3.1 shows the distribution of links with corruption and congestion in different loss buckets, normalized such that the total in each column adds to 100%. Overall, only a small percentage of links in the data center network have any corruption or congestion; we exclude these percentages for confidentiality. We see that over 90% of links with congestion have a loss rate between 10^{-8} and 10^{-5}. As the loss rate increases, the percentage of links with congestion in each category decreases. This is expected, as congestion control reduces flows' sending rate which lowers the loss rate. However, the same trend does not hold for corruption: 12.67% of links appear in the last loss bucket with loss rate $\geq 10^{-3}$ (or 0.1%) whereas

[2]IEEE 802.3 standard requires each link to have corruption loss rate under 10^{-8}, but operators today tend to worry only when packet loss rates start approaching 10^{-6}. In this chapter, we conservatively use 10^{-8} as the threshold to deem a link as lossy or non-lossy.



(a) Loss rate of one link

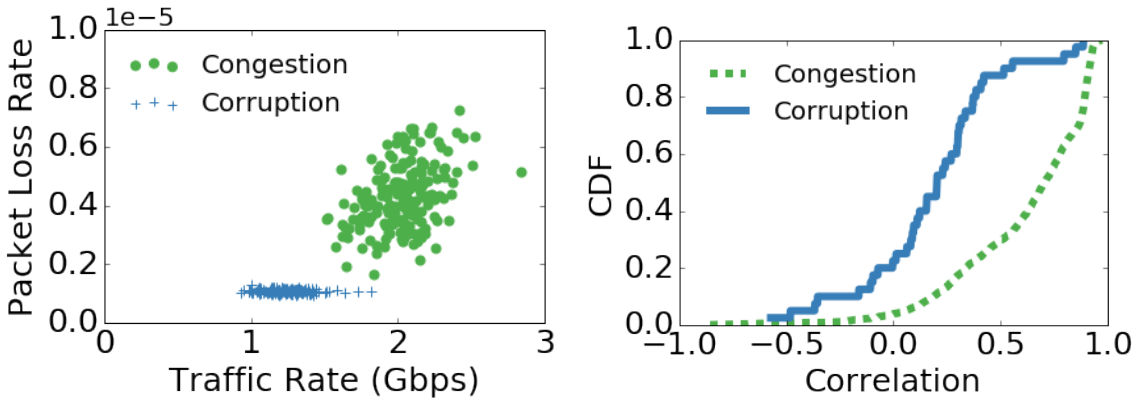
(b) CV of loss rate for all links

Figure 3.2: Packet loss rate over time for two types of loss events. Figure 3.2a compares the corruption loss rate with congestion loss rate of a link for one week and shows corruption has little variation than congestion. Figure 3.2b plots the cumulative distribution function of the coefficient of variance of packet loss rate across 15 data centers for one week and shows corruption loss rate has less variation compared to congestion loss rate.

congestion appears in 0.22% of links.

Corruption rate is stable over time. Figure 3.2a shows the corruption and congestion loss rate of example links for one week. We can see that the corruption rate has less variation compared to congestion which frequently varies by three orders of magnitude in a short amount of time. A similar observation holds for the entire data set. We quantify the level of variation of loss rate on a link using the coefficient of variation (CV), which is standard deviation divided by the mean.

Figure 3.2b shows the cumulative distribution function (CDF) of the CV between congestion and corruption loss rates for all links in one week. For 80% of the links, the CV for corrupting links is smaller than four; for congestion, it is about twice that amount. Thus, corruption loss rate is more stable than congestion loss rate on average.



(a) Utilization vs. loss rate

(b) Correlation of utilization and loss rate

Figure 3.3: Correlation of packet loss rate and traffic rate for two types of loss events. Figure 3.3a shows a scatter plot of utilization versus loss rates of a link for one week. Unlike congestion, the corruption loss rate does not change as the link's utilization changes. Figure 3.3b plots the cumulative distribution function of Pearson correlation between utilization and logarithm of loss rate. The average Pearson correlation between utilization and congestion loss rate is 0.62. The average Pearson correlation between utilization and corruption loss rate is 0.19.

Corruption rate is uncorrelated with utilization. Inherently, corruption loss rate is stable over time because it does not vary with link utilization. Figure 3.3a shows a scatter plot of utilization versus loss rate of a link for one week. Congestion loss rate has a positive correlation with the outgoing traffic rate. Unlike congestion, corruption loss rate does not change as the link's incoming traffic rate changes. To depict the lack of correlation in all links, we compute the Pearson correlation between utilization of a link and the logarithm of its loss rates. Figure 3.3b shows the cumulative distribution function of Pearson correlation across all links. The mean correlation between incoming link utilization and corruption loss rate is only 0.19, and 85% of the links have a correlation between -0.5 and +0.5. Hence, for most links, the correlation is low between their utilization and corruption

loss rate. In contrast, the mean correlation between the outgoing link utilization and the congestion loss rate is 0.62, which indicates a strong positive correlation. This behavior is expected: more traffic under randomized or bursty arrivals should lead to a higher congestion loss rate.

Because the correlation between corruption loss rate and utilization is low, application or transport layer reactions will not resolve it. Thus, unlike congestion, packet corruption is a pernicious fault that is is not mitigated when senders slowdown. Instead, it persists—to stop it, we must disable the link and then have technicians fix it. A related, unfortunate aspect of corruption losses is that they lead transport protocols such as TCP to unnecessarily slow down, interpreting corruption loss as a signal of congestion. This does nothing to resolve corruption but does hurt application performance.

Corruption has weak spatial locality. We investigated if corrupting links tend to be spatially correlated (e.g., on the same switch or topologically close) or scattered across the data center network (uniformly and) randomly. We found weak spatial locality. To demonstrate this finding, we first compute the fraction of switches in the data center network that have at least one link with a high corruption loss rate, i.e., in the set of the worst 10% of corrupting links. We then simulate a hypothetical setting in which the same number of corrupting links are randomly spread through the network, and again compute the fraction of switches to which they belong. We then calculate the ratio of the two switch fractions. For example, let's say $x\%$ of switches contain the worst 10% of corrupting links. If those corrupting links are uniformly distributed, $y\%$ of switches will contain them. Then the ratio is $\frac{x}{y}$. If the ratio is 1, it suggests that corrupting links are scattered randomly across the switches. Lower ratios indicate more co-location with switches.

We repeat this analysis for 100 different values, between 0 and 100%, for the set of corrupting links chosen, and we also repeat the analysis for congested links. As Figure 3.4 shows, for congestion, the number of affected switches is only 20% of what the random

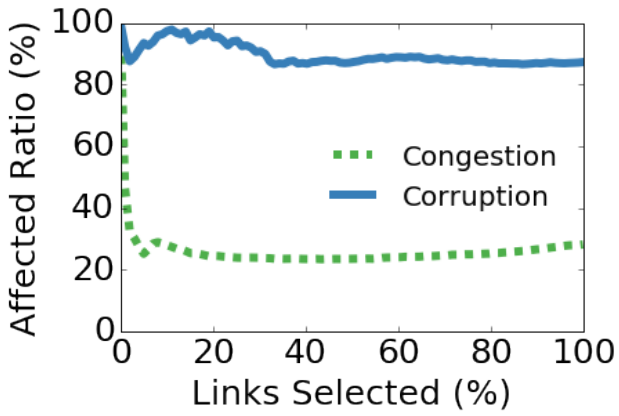


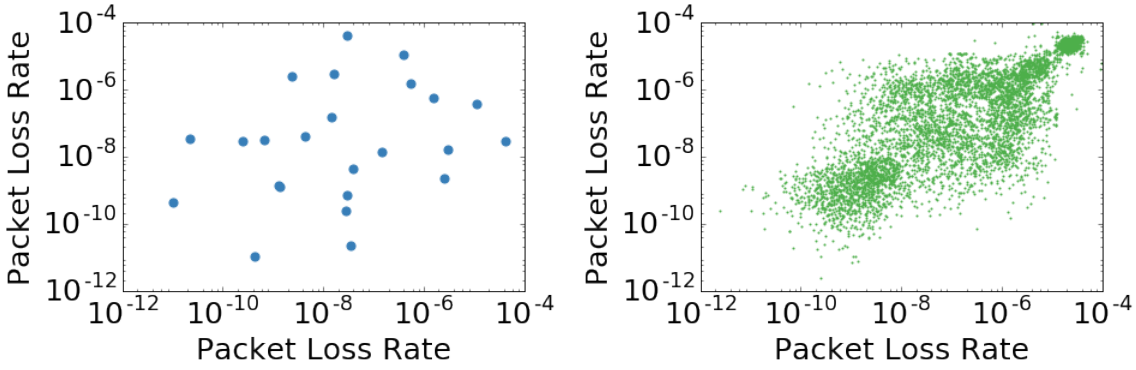
Figure 3.4: Spacial locality for two types of loss events. The top X% loss rate links belong to Y% of switches. Y is normalized to the fraction of switches if the lossy links are uniformly randomly distributed.

distribution suggests. This means congested links exhibit a high degree of spatial locality. For corruption, this ratio is around 80%, which indicates weak spatial locality. We can also see that when we focus on the worst corrupting links (e.g., the top 10%), the locality is weaker. Thus, the worst offenders are more likely to be randomly spread in the network.

While we expected congestion to exhibit locality, the locality (albeit weak) of corruption surprised us. As we show in Chapter 3.3, it occurs because of shared root causes (e.g., bad switch backplane or poorly-routed fiber bundle). We also found that spatially related links start corrupting packets at roughly the same time and have similar corruption loss rates.

Corruption is asymmetric. Corruption in one direction of the link does not imply corruption in the reverse direction. With a week's worth of data, we observed that only 8.2% of the links with packet corruption had bidirectional corruption.[3] For congestion, 72.7%

[3]This asymmetry implies that a more efficient way (in terms of network capacity) to mitigate corruption would be to disable only one direction of the link, but since current hardware and software does not allow unidirectional links, we disable both directions in CorrOpt.



(a) Corruption (b) Congestion

Figure 3.5: Symmetry of packet loss rates for two types of loss events. Figure 3.5a shows corruption loss rate at x-axis and the corruption loss rate on the opposite direction at y-axis. Figure 3.5b is a similar figure for congestion loss rate.

of links with congestion losses had bidirectional losses. For those links with bidirectional losses, Figure 3.5 shows average packet loss rate on both directions of the link. For congestion, we see a cluster of links for which the congestion loss rates in both directions are large. We speculate that high, bidirectional congestion is caused by link failures that temporarily reduce network capacity for both upstream and downstream traffic.

Corruption is uncorrelated with link location. Corruption happens at every stage of the data center network topology. We computed the probability that a link is corrupting for each stage of the network (e.g., ToR-to-fabric, fabric-to-ToR, fabric-to-spine), and we did not observe any bias. This observation also implies that corruption does not depend on cable length, since cable lengths at higher stages tend to be longer, or the type of switch. In contrast, we find (in agreement with earlier work [139]) that certain stages of the data center network have significantly fewer congestion losses than the rest. We also find congestion losses to be inverse correlated with the use of deep buffer switches. For switches at the same stage, shallow buffer switches experience more congestion losses

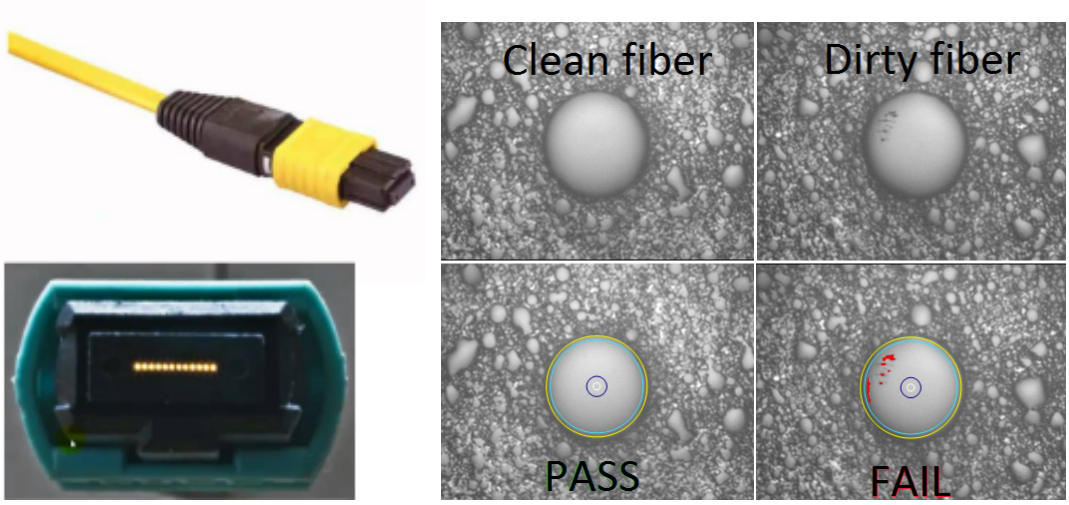
than deep buffer switches.

3.3 Root Causes of Corruption

To successfully repair a corrupting link, operators need to address the root cause of corruption. We analyzed over 300 trouble tickets while monitoring all links' optical receive power (RxPower) and transmit power (TxPower), as well as their corruption statistics. This tandem monitoring of tickets and link statistics turns up a set of symptoms that are the most common signature of each root cause. In Chapter 3.4.2 we describe how we use these symptoms to recommend repair actions to on-site technicians to help them eliminate corruption faster.

Root cause 1: Connector contamination An optical link consists of fiber optics cable and a transceiver on each end. Transceivers convert the signal between electrical and optical domains, and the fiber carries the optical signal. In fiber optics, the tolerance of dirt or contamination on a connector is near zero [87]. Airborne dirt particles may even scratch the connectors permanently if not removed. Fiber tips or connectors can become contaminated during installation or maintenance. Patch panels can become contaminated if they are left open to the air or scrape off foreign particles under repeated usage. Figure 3.6a shows a fiber connection with 12 fiber cores. Prior to installation, technicians should inspect each fiber core manually using a fiber microscope. Figure 3.6b shows a magnified image of two cores we inspected using a P5000i Fiber Microscope [84]. In this case, the device found more than five defects larger than 2 μm in diameter on the right-hand side fiber core and failed the test on it. Common types of contamination and defects include dirt, oil, pits, chips, and scratches [63]. Fiber cleaning can remove dirt and contamination on the connector.

Contamination reduces RxPower which increases the probability of packet corruption by making it more likely that the transceiver is unable to decode the signal correctly [155]. Since fiber optics cables and connectors are unidirectional, we find that the most probable

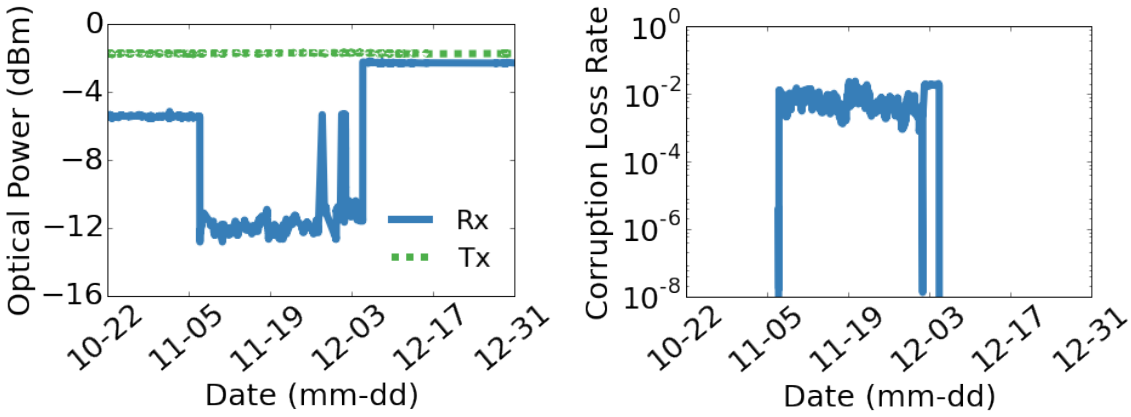


(a) Fiber connector (b) Magnified image of 2 fiber cores

Figure 3.6: Illustration of dirty optical connections. Figure 3.6a shows an fiber connection consisting of 12 fiber cores. Figure 3.6b shows a magnified image of two cores. These images were taken using a P5000i Fiber Microscope to inspect and certify fiber end face quality in production data centers [84]. In this case, the tool found more than five defects larger than 2 μm in diameter on the right-hand side fiber core. Technicians are supposed to inspect each fiber core manually prior to installation–a task that is largely ignored because it is manual and cumbersome. Similar observations are reported for other fiber types [66].

indicator of contamination is high TxPower on both sides of the link, with low RxPower along only one direction of the link (i.e., the receiving side of corruption). As Figure 3.7 shows, the packet corruption of a link jumps at the same time as its RxPower drops, but the TxPower on the opposite side remains stable. In this case, cleaning the both sides of the link mitigates the corruption.

Not all forms of contamination cause low RxPower; some cause back reflections, where the RxPower remains high but the reflections interfere with signal decoding. Transceivers do not report on reflections, and thus we are not able to correctly identify this root cause



(a) Optical power

(b) Corruption loss rate

Figure 3.7: An example of a dirty connection causing packet corruption. RxPower suddenly drops on November 5, causing packet corruption loss to increase to 10^{-2} (1%). TxPower on the transmit side shows no changes. Fiber cleaning takes place on November 27 which restores RxPower level and eliminates the corruption.

all the time. Such limitations of accurate identification exist for other root causes as well, which is why the accuracy of our repair recommendations is not 100%.

Root cause 2: Damaged or bent fiber. Cable management is a tedious task in large fiber plants. A bent or damaged fiber causes the optical signal to leak out of the fiber, reducing the signal strength in the fiber. Cables should be laid such that they are not bent beyond their specification, especially for fibers at the bottom layer of a fiber housing mount. Figure 3.8 shows a case in our data center; the fibers at the bottom row are too bent, causing corruption.

When we study the RxPower of damaged or bent cables, we find both sides of the link are likely to have low RxPower coupled with high TxPower. Figure 3.9 shows an example of a damaged fiber causing packet corruption. Another indicator of cable damage is that switches on both sides experience packet corruption, which is otherwise rare (Chapter

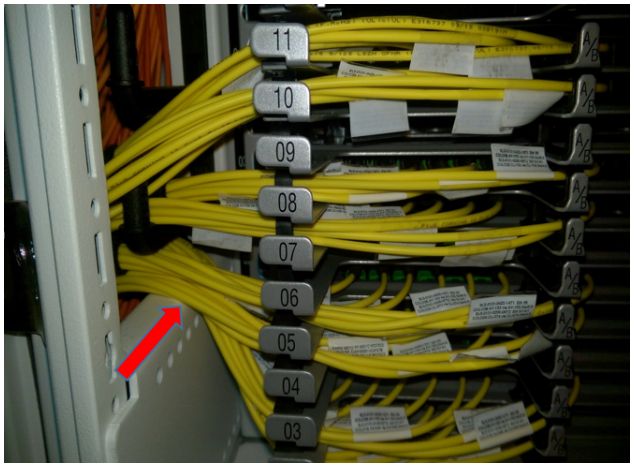


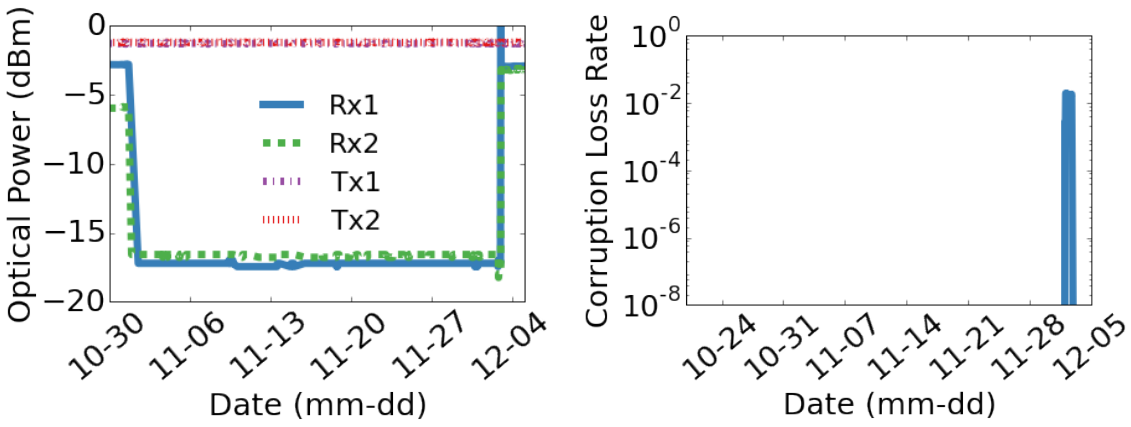
Figure 3.8: Illustration of bent fiber cables. Bent fiber (red arrow) can cause packet corruption when the fiber cable is bent tighter than its maximum bend tolerance.

3.2).

Root cause 3: Decaying transmitters. Transceivers are built using semiconductor laser technology. While lasers tend to have a long life expectancy, old lasers can suffer deterioration in TxPower, leading to low RxPower and corruption on the receive side of the link. Replacing the dying transceiver can resolve the problem. The most probable symptom of decaying transmitters is that the TxPower on the send side of the link and RxPower on the receive side of the link are both low or are gradually decreasing.

Root cause 4: Bad or loose transceivers. Bad transceivers or loosely-seated ones (i.e., not properly plugged in) also cause corruption. When this happens, technicians should take out the transceiver and plug it back in (a.k.a., reseating the transceiver). If the issue is not resolved, the transceiver is likely bad and needs to be replaced.

When bad or loosely-seated transceivers cause corruption, optical TxPower and RxPower on both sides of the link are high, but the link still corrupts packets. This symptom is shared by root cause 5 as well, but a distinguishing characteristic of this root cause is



(a) Optical power

(b) Corruption loss rate

Figure 3.9: An example of a damaged fiber causing packet corruption. Fiber damage happens on October 30 causing both sides' RxPower to suddenly drop at the same time. TxPower on both sides are not affected. When traffic is put on the link at the beginning of December, the corruption loss rate is around 1%. Fiber replacement restores both sides' RxPower back to normal level.

that only one of the links on the switch is bad. It is uncommon (but still possible) that multiple transceivers on the same switch are bad or loose.

Root cause 5: Shared-component failure. Some infrastructure components in a data center network, such as breakout cables[4] and switches, are shared by multiple links. A faulty breakout cable causes four links on the same switch to have packet corruption at the same time. Breakout cable replacement can resolve the issue. Faults in the switch backplane can also cause multiple links to experience corruption. In such cases, several links on the shared infrastructure suffer packet corruption, despite good optical power levels on all of them. In addition, the corruption loss rate on these links is similar. When

[4]A breakout cable splits a high-speed port (e.g, 40Gbps, 100Gbps) into several low-speed ports (e.g, 10Gbps, 25Gbps). It is typically used between switches with different port speed.

switches have unused ports that are not affected by the failure, rewiring can resolve the issue. Otherwise, the switch has to be replaced. This root cause is primarily responsible for the spatial locality of packet corruption (Chapter 3.2). We find that links experiencing packet corruption because of other root causes, which usually accompany low RxPower, exhibit no locality.

Table 3.2 summarizes the root causes mentioned above, their symptoms and their relative contribution in our data centers. We use the notation of $TxPower \rightarrow RxPower$ to indicate the power levels of each side of optical links. H and L indicate if the power level is above or below the acceptable threshold (determined by the transceiver technology and loss budget of links). The percentage of contribution of each root cause is presented as a range because our ticket diaries show that technicians often take multiple actions (e.g., clear the connectors and reseat the transceiver) without logging which action resulted in the repair. When a root cause is present in such a bundle, we assume that it was not the culprit to compute the low end of the reported range and we assume that it was the culprit to compute the high end of the reported range.

3.4 Mitigating Corruption

There are two aspects to CorrOpt, our system to mitigate corruption. First, to protect applications from corruption, we disable corrupting links, while meeting configured capacity constraints. Meeting capacity constraints is important because otherwise we may trade off corruption losses for heavy congestion losses. We consider disabling links in this chapter; it requires minimal changes to our existing infrastructure. We consider other strategies, such as error coding and traffic engineering to move sensitive traffic away from corrupting links, in Chapter 4.

If we rely solely on disabling links for corruption mitigation, the data center network will have fewer and fewer links as time progresses. Instead, we must also fix the root cause of corruption, so links can be enabled again. Thus, the second aspect of CorrOpt is generating repair recommendations for disabled links based on root causes and symp-

Root cause	Most likely symptom $TxPower \rightarrow RxPower$ $RxPower \leftarrow TxPower$	Contribution
Connector contamination	$H \rightarrow H$, $L \leftarrow H$	17-57%
Bent or damaged fiber	$H \rightarrow L$, $L \leftarrow H$	14-48%
Decaying transmitter	$* \rightarrow *$, $L \leftarrow L$	$< 1\%$
Bad or loose transceiver	$H \rightarrow H$, $H \leftarrow H$, single link	6-45%
Shared component failure	$H \rightarrow H$, $H \leftarrow H$, co-located links	10-26%

Table 3.2: Summary of root causes of corruption, their symptoms and their relative contribution in our data centers. H stands for high power; L for low power; * for either. Contribution is range to indicate potential error in assigning root causes from repair logs.

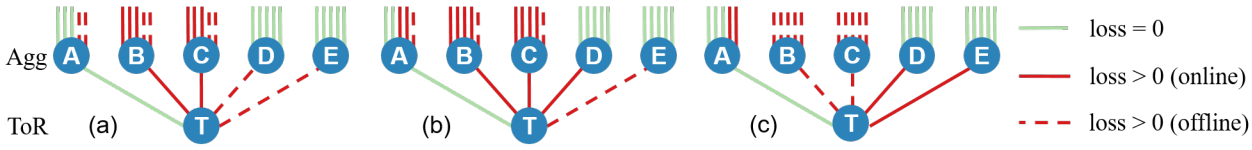


Figure 3.10: Example of problems with switch-local checking. With ToR capacity constraints of c=60%: (a) Every switch keeps s_c=c=60% of its uplinks alive, resulting in 8 disabled links, but only 9 out of 25 paths to the spine are still available for T, far below the constraint of 60%. (b) When s_c=$\sqrt{c} = 0.77$ of the links are kept online, the ToR capacity constraint is met, but only 4 links can be disabled. (c) The optimal solution, which has 12 disabled links offline and meets the capacity constraints.

toms described in Chapter 3.3. Our recommendations reduce both repair time and packet loss (Chapter 3.6.2).

3.4.1 Disabling Corrupting Links

While disabling corrupting links reduces corruption losses, it also reduces network capacity. In extreme cases, blindly disabling links can create congestion or even partition the network.

To lower corruption losses without creating congestion, we consider a common capacity metric [99, 140, 149]: the fraction of available valley-free paths from a top-of-rack switch (ToR) to the highest stage of the network (i.e., the spine). This metric quantifies available capacity and redundancy for a ToR after links are disabled. Because traffic demand can differ across ToRs [73], we allow per-ToR thresholds. Our data show up to 15% of corrupting links cannot be disabled due to capacity constraints under realistic configurations (e.g., when every ToR has threshold between 50–75%).

CorrOpt determines the subset of links to disable based on the impact of corrupting links that remain active. Each link l with packet corruption rate of f_l has impact $I(f_l)$, where $I(.)$ is a monotonically increasing penalty function that reflects the relationship between application performance and loss rate [120, 152]. CorrOpt aims to minimize the total penalty of packet corruption, i.e., $\sum_{l \in \text{links}}(1 - d_l) \times I(f_l)$, where d_l is 1 if the link is disabled and 0 otherwise. Our goal is to determine the value of d_l for each link l, while meeting capacity constraints.

However, as we prove in Appendix A (via reduction to 3-SAT) this problem is computationally difficult.[5]

Theorem 1. *Deciding which links to disable in a Clos topology, s.t. the total penalty of packet corruption is minimized under capacity constraints, is NP-complete.*

[5]The NP-hard problem stated in [42] is orthogonal to our formulation, as it moves logical machines between physical machines.

Because of the complexity, we cannot quickly determine the optimal set of links to disable. Speed is desirable to protect applications from corruption, but it is not possible to be both fast and optimal.

State-of-the-art: switch-local checking. Current data center network operators opt for speed [104]. When a new corrupting link is found, a controller decides whether it can be disabled based on the number of available uplinks at the switch to which it is attached. For a threshold of s_c and a switch with m uplinks, $\lfloor m \times (1 - s_c) \rfloor$ of the uplinks can be disabled. For example, with $m = 5$ uplinks and $s_c = 60\%$, at most two uplinks can be disabled. When a link is enabled, after repairing corruption or other problems, the same check is run for all active corrupting links to see if additional links, which could not be disabled before, can be disabled now.

Unfortunately, switch-local checks can be sub-optimal. Figure 3.10 shows an example, where T is a ToR with five uplinks, to switches (A through E) that also have five uplinks each. Corrupting links are in red, and dashed lines represent disabled links. Suppose we want each ToR to have capacity $\geq c$=60%. If we directly map c to a switch-local constraint, i.e., s_c=c, Figure 3.10a shows the network state that will emerge. Switch-local checking leads to disabling eight links. However, ToR T now has only nine of 25 (36%) possible paths to the spine, far below the desired limit of 60%.

This problem can be fixed by enforcing a switch-local capacity constraint of $s_c = \sqrt{c} = 0.77$ because this forces c fraction of paths to the spine switches to be available. But now, as shown in Figure 3.10b, each switch can disable only one corrupting uplink, for a total of four disabled links (out of a total of 16 corrupting links). The optimal solution, however, shown in Figure 3.10c, can disable as many as 12 corrupting links, for a much lower total penalty due to active corrupting links.

Generalizing the example above, in a simple ToR-fabric-spine-topology, a capacity constraint of c requires every switch to keep $\sqrt{c}$ of its uplinks. Otherwise, the capacity constraint can be violated. The gap widens when the data center network has more tiers:

with r tiers above the ToR-level, a switch-local algorithm needs to keep $\sqrt[r]{c}$ fraction of uplinks active.

Another limitation of a switch-local checker is that it cannot handle different ToR requirements well. If one ToR has a high capacity requirement c', all upstream switches need to keep $\sqrt[r]{c'}$ uplinks active. A switch-local checker may not be able to disable a single link in extreme cases.

CorrOpt's approach. CorrOpt achieves both speed and optimality using a two-pronged approach. First, when a new corrupting link is found, it runs a *fast checker* for a quick response that exploits global network state to bypass the sub-optimality of switch-local checking. Second, when links become active, CorrOpt runs an *optimizer* that globally optimizes over all active corrupting links in the network. Link activations allow other remaining corrupting links to be turned off. Those links tend to have lower loss rates than newly arrived corrupting link due to the fast checker disabling lossy links, which gives us more time to compute a better solution. By analyzing the failure structures in our data set, we are able to efficiently solve the practical instances of this NP-complete problem. We now provide more detail on the two components.

Fast checker. Conceptually, when a new corrupting link l arrives, CorrOpt counts the remaining paths for each ToR to the spine assuming l is removed from the topology. If no ToR's constraint is violated, CorrOpt disables l and creates a maintenance ticket for it with a recommended repair. This process is repeated for each new corrupting link. As long as no link is activated since its last run, the network state after the fast checker runs is maximal, i.e., no more links can be disabled. Thus, we never need to re-run the fast checker on old corrupting links.

Because of its exact counting of paths, our fast checker can disable more links than switch-local checking. A naive implementation of the fast checker is to iterate over all the paths from ToR switches to the spine switches in order to count the number of available

paths for each ToR. That would be slow because a large data center network can possibly have millions of paths. Using information about the links E in the data center network, we efficiently implement CorrOpt's fast checker as follows. First, for each switch v_2 in the second-highest stage, we count the active (one-hop) paths $p_1(v_2)$ to the spine (i.e., the highest stage). Then, each switch v_3 in the third-highest stage adds $p_1(v_2)$ to each of its active uplinks, obtaining the number of two-hop paths $p_2(v_3)$ to the spine. This process is iterated until the ToR-stage is reached.

With this information, to see if l can be safely disabled, we check the downstream of l, updating the path counts with the same method, beginning with the switch directly downstream of l. If all downstream ToRs of l meet the capacity constraints with l offline, l is disabled. Conceptually, we perform $O(1)$ operations per link, resulting in a linear runtime of $O(|E|)$. In our experiments, the fast checker takes only 100-300 ms for the largest data center network in our data set, effectively providing instantaneous decisions.

Optimizer. When a link is enabled, one option is to rerun the fast checker on all active corrupting links, as is done today in switch-local checks. However, we can afford to run a potentially-slower computation to determine the optimal subset of links to disable, given the newly available capacity. The optimization problem is what we defined earlier, operating over the set of active corrupting links.[6]

Even though the problem is NP-complete, we can provide a fast exact algorithm in practice. First, we find that, under realistic capacity constraints, 99% of the ToRs can be ignored because their capacity constraints will not be violated even if all corrupting links are disabled. Only the links that are in danger of capacity constraint violation need to be considered. To identify such links, we run fast checker's path counting procedure on

[6]For practical reasons, CorrOpt does not enable corrupting links before they have been repaired. In theory, we can further reduce corruption losses by doing so; e.g., pre-maturely enabling a link with a lower corruption rate may allow a link with a higher corruption rate to be disabled. On the small to medium data center networks in our data set, the performance of this optimal version was close to that of CorrOpt. We could not evaluate this version on our large data center networks because of its computational complexity, which is worse than CorrOpt's optimizer since it considers a bigger set of links at each step.

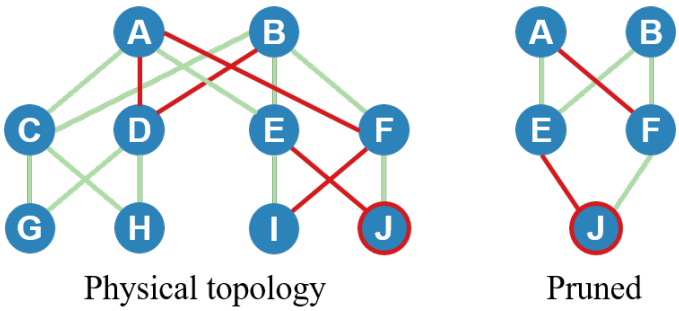


Figure 3.11: Example of topology pruning. If capacity constraint is 50%, only ToR J will violate the constraint if all corrupting links (in red) are disabled. Hence, we only need to consider the pruned topology and can safely disable the other three links.

the network with all corrupting links considered disabled, identifying all ToRs V whose capacity constraints are violated. Only disabling links upstream of the ToRs in V can violate capacity constraints. All corrupting links not upstream of V can hence be safely disabled, thus pruning the topology.

For instance, in Figure 3.11, assume the capacity constraint is 50%. The corrupting links are shown in red, and if we were to disable all of them, ToRs G, H, and I will have at least two out of four paths to the spines, but ToR J will have only one. Thus, we can remove all links and switches except those upstream of J, and the three removed corrupting links can be safely disabled.

Next, we need to decide which remaining corrupting links in the pruned topology can be disabled. CorrOpt iterates through all possible subsets to measure (1) whether the capacity constraint is met if the entire subset is turned off (2) and the total penalty if the subset is turned off. To improve performance, CorrOpt uses a "reject cache" to memorize subsets that can fail capacity constraint. This allows CorrOpt to ignore any super set of set in the reject cache. For example, in Figure 3.11, {AF, EJ} can be kept in the cache because turning both AF and EJ off means ToR J has at most one path (25%) to the core network. Any subset S such that S is a superset of {AF,EJ} is guaranteed to fail the

capacity constraint and thus can be ignored.

The output of CorrOpt's optimizer is an exact solution to the optimization problem. Pruning only removes links that can be safely disabled and the "reject cache" only skips infeasible link sets. In our experiments, the combination of techniques allows the optimizer to run in less than one minute on a 1.3 GHz computer with 2 cores, for the topologies in our data set.

3.4.2 Corruption Repair

Simply stated, repairing corruption in today's data center networks is cumbersome. Unlike switch configuration errors or congestion, corruption cannot be remedied via a software-based reaction. For example, as mentioned in Chapter 3.3, dirt on connectors can cause corruption, and the only repair is to manually clean the connections. If the root cause of the corruption is not correctly diagnosed, on-site technicians must rely on guesswork when deciding what action to take.

Network technicians currently use manual diagnosis. When assigned to a ticket, they manually inspect the transceiver and the fiber to find tight bends or damage. If equipment is not connected firmly, they reconnect it. If tight bends or damage are found on the fiber, the technicians replace the fiber. If they cannot find any problem visually, they may choose to clean the connector with an optical cleaning kit [141].

If the repair does not address the actual cause of packet corruption, the link will continue to corrupt packets as soon as the link is enabled. Figure 3.7 and Figure 3.9 show examples of successful repair. In contrast, Figure 3.12 shows a series of two unsuccessful repair attempts. Both include cleaning the fiber and reseating the transceiver. Finally, on the third try, the technician replaces the fiber and fixes the corruption.

This whole process takes several days. In between repair attempts, the link is enabled and a new ticket is generated when it is disabled again. Generated tickets are placed in a FIFO queue; thus, the exact time needed for a fix depends on the number of tickets

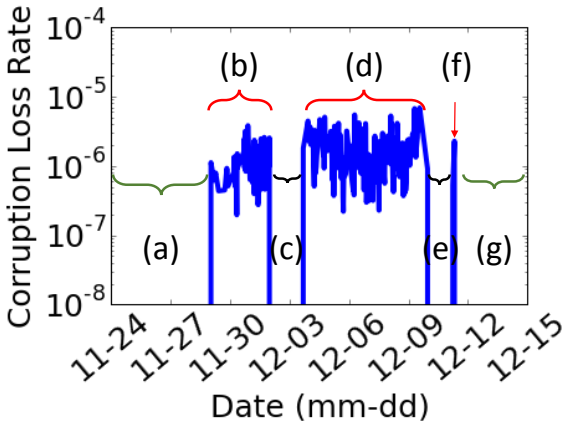


Figure 3.12: An example of unsuccessful repair actions on a link. (a) A healthy state with corruption loss rate below 10^{-8}. (b) Starts corrupting packets. (c) Disabled for repair. (d) Enabled after the repair but starts corrupting packets again. (e) Disabled again. (f) Enabled after but the repair failed again. (g) Disabled again for repair, and the repair is finally successful, more than a week after the problem started.

in the queue. Our analysis of 3400 tickets shows that, on average, it takes two days for technicians to resolve a ticket; this means, each failed repair attempt adds two more days during which the link must be disabled.

Unsuccessful repairs also increase the likelihood of collateral damage because technicians need to enter the facility more often. Each entry poses a risk of them affecting something unrelated (e.g., tripping over cables, replacing the wrong cable or transceiver, or accidentally powering off equipment).

In CorrOpt, we seek to improve the accuracy of repair by leveraging our observations of the most likely symptoms of corruption root causes (Chapter 3.3) in terms of optical power levels and the link's history. Our strategy is listed in Algorithm 1. It first uses packet corruption rate on neighboring links to identify shared component failures. Then it uses TxPower on the opposite side to detect decaying transmitters. CorrOpt uses RxPower to separate optical and non-optical issues. With non-optical issues, the only

Algorithm 1 CorrOpt's recommendation engine

```
1:  function RECOMMEND REPAIR(link)
2:      neighbors ← links sharing same component (i.e., switch)
3:      if has_corruption(neighbors) then
4:          return Replace shared component
5:      if has_corruption(opposite_side) then
6:          return Replace cable/fiber
7:      Rx1 ← RxPower of link
8:      Rx2 ← RxPower of opposite side of link
9:      Tx2 ← TxPower of opposite side of link
10:     if Tx2 ≤ Power_ThreshTx then
11:         return Replace transceiver on the opposite side
12:     if Rx1 < Power_ThreshRx and Rx2 < Power_ThreshRx then
13:         return Replace cable/fiber
14:     if Rx1 < Power_ThreshRx then
15:         return Clean fiber
16:     else
17:         if Transceiver is not reseated recently then
18:             return Reseat transceiver
19:         else
20:             return Replace transceiver
```

solution is to try reseating the transceiver, and then to replace it.

CorrOpt uses $Power_{ThreshRx}$ ($Power_{ThreshTx}$) per optical technology as the minimal RxPower (TxPower) threshold. When both ends of a link have RxPower below $Power_{ThreshRx}$, this suggests bent or damaged fiber. Connector contamination tends to cause RxPower to be low in one direction. Cleaning connectors with fiber cleaning kits can often fix corruption.

CorrOpt's recommendation engine has been deployed in Microsoft data centers since October 2016. Chapter 3.6.2 evaluates its effectiveness.

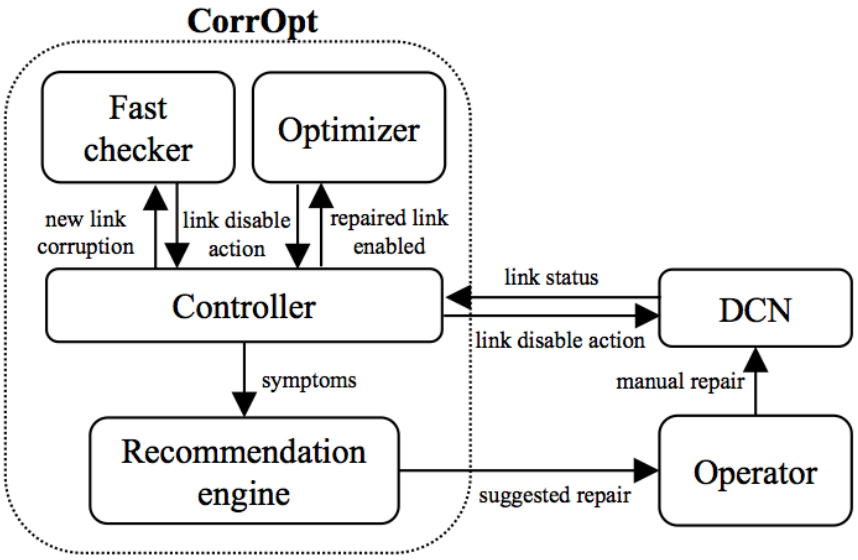


Figure 3.13: CorrOpt's system components and workflow.

3.5 Implementation

Figure 3.13 shows the workflow and system components of CorrOpt. When a switch detects packet corruption, it reports to the CorrOpt controller. The controller uses the fast checker logic to quickly determine if the link can be safely disabled. If the link is disabled, the recommendation engine (Chapter 3.4.2) generates a ticket with a suggested repair procedure, based off the monitoring data (collected by another system). When a link is activated, CorrOpt uses the optimizer logic to check if any active corrupting links can be disabled.

We implemented fast checker and optimizer with around 500 lines of python code. We integrated CorrOpt's recommendation engine into the cloud provider's infrastructure with around 50 lines of C# code.

3.6 Evaluation

We now evaluate CorrOpt for $i)$ its ability to protect applications by safely disabling corrupting links, while meeting capacity constraints; and $ii)$ its ability to speed repairs by correctly identifying the root cause. The first evaluation uses simulations based on data from our data center networks, and the second uses our deployment of CorrOpt's repair recommendation engine for three months. We study these two factors individually in Chapter 3.6.1 and Chapter 3.6.2, and we quantify their combined impact in Chapter 3.6.3.

3.6.1 Disabling Links

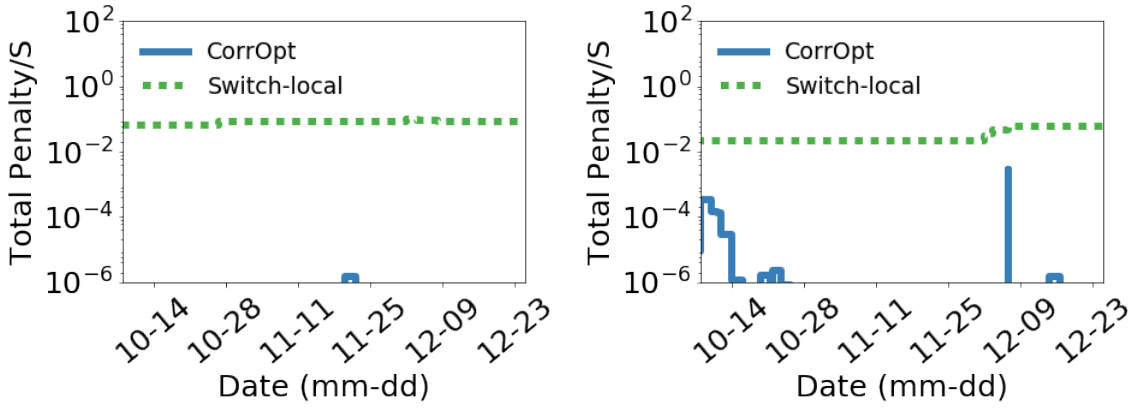
We simulate the impact of CorrOpt using the topologies and link corruption traces from two production data centers, a large data center network with O(35K) links and a medium-sized data center network with O(15K) links. The trace period is from Oct to Dec 2016.

We quantify the effectiveness of CorrOpt at disabling links using "total penalty." Each corrupting link l with corruption rate f_l incurs a penalty of $I(f_l)$ per second (Chapter 3.4), and the total penalty per second is $\sum_{l \in \text{links}}(1 - d_l) \times I(f_l)$, where d_l is 1 if the link is disabled and 0 otherwise. For simplicity, results in this chapter use $I(f_l) = f_l$. Thus, the total penalty is proportional to corruption losses (assuming equal utilization on all links).

We compare CorrOpt with "switch-local," the link disabling technique used today. As we discussed earlier, for this method to guarantee a capacity constraint of c, it should be configured with $s_c = \sqrt{c}$ for three-stage data center networks (which is what we study).

To isolate the impact of link disabling strategy, we couple both methods with the same repair effectiveness (as CorrOpt's). When a link is disabled, it is put into a queue of links that are waiting to be fixed. Links stay in that queue for two days, the average service time in our data center networks (Chapter 3.4.2). Based on our observed repair accuracy (Chapter 3.6.2), 80% of the links are repaired correctly after this time. The remaining take an additional round to fix, so the overall it takes them four days to be enabled again.

Figure 3.14 shows the performance of both methods for the two data center networks,



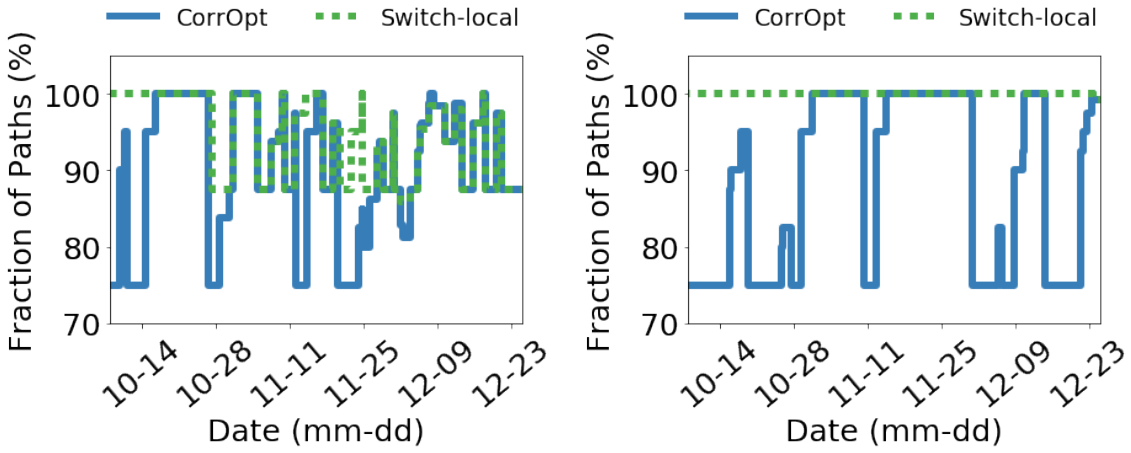
(a) Medium DCN

(b) Large DCN

Figure 3.14: Total penalty per second of switch-local and CorrOpt when the capacity constraint is 75% for every ToR, for traces taken from two data centers.

when the capacity constraint is c=75% for every ToR. The x-axis is time, and the y-axis is total penalty per second. We see that the penalty of the switch-local checker is much higher, because of its sub-optimality that we illustrated earlier. It is piecewise flat for switch-local approach because there is a set of corrupting links that approach is not able to disable and in our model, they corrupt packets at constant rates. It is not completely flat due to the arrivals of corrupting links. In contrast, CorrOpt can disable the vast majority of the corrupting links, leading to a much lower penalty. The penalty varies with time based on the number and relative locations of corrupting links in the data.

The inability of the switch-local checker to disable links is visible in Figure 3.15 and Figure 3.16, which show the worst ToR's fraction of available paths to the spine when the capacity constraint is 75% and 50%. When lines overlap, it means for some period of time, performance of CorrOpt is the same of switch-local check. When capacity constraints are low (50%), CorrOpt and switch-local lines overlap. Overall, we see that CorrOpt can hit the capacity limit as needed, but switch-local often leaves links enabled even though it is not limited by the capacity constraint.



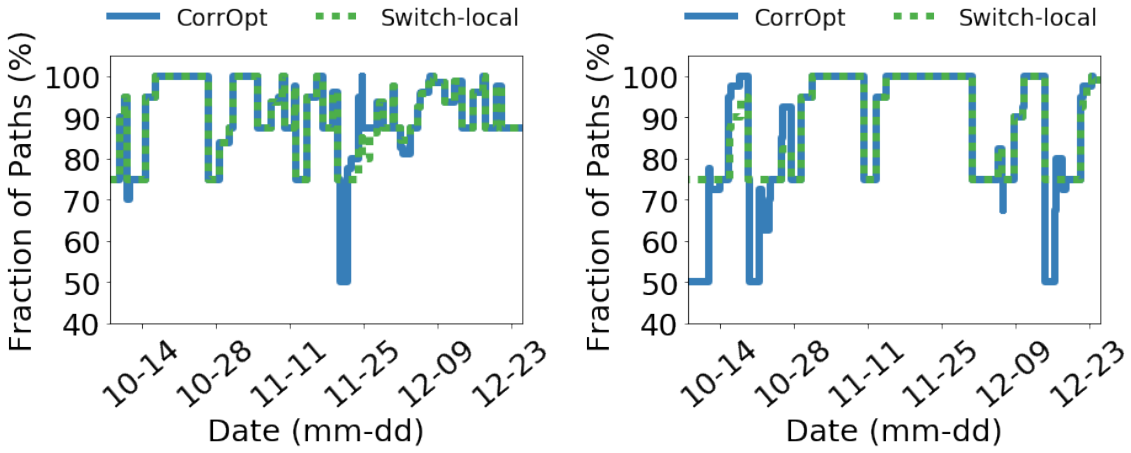
(a) Medium DCN

(b) Large DCN

Figure 3.15: Fraction of available paths to the spine for the worst ToRs when the capacity constraint is 75%, for traces taken from two data centers.

Impact of the capacity constraint The advantage of CorrOpt over today's switch-local checks depends on the capacity constraint. If the constraint is lax, both methods are expected to perform similarly, as both can turn off almost all corrupting links. However, when the constraint is more demanding, the intelligent decision making of CorrOpt begins to shine. For different capacity constraints, Figure 3.17 shows the total penalty, integrated over time, of CorrOpt divided by that of the switch-local checker. Since our penalty function is linear in corruption losses, this ratio represents the reduction in the amount of corruption losses.

We see that when the capacity constraint is lax (c=25%), as expected, there is no difference between the two methods. However, when the capacity constraint is 50% or higher, a more realistic regime, CorrOpt outperforms the switch-local checker. On the medium size data center, with a capacity constraint of 50%, CorrOpt can eliminate almost all corruption while the switch-local check keeps some corrupting links active. Thus, the total penalty ratio drops to 0. When the capacity constraint is 75%, CorrOpt's total penalty is



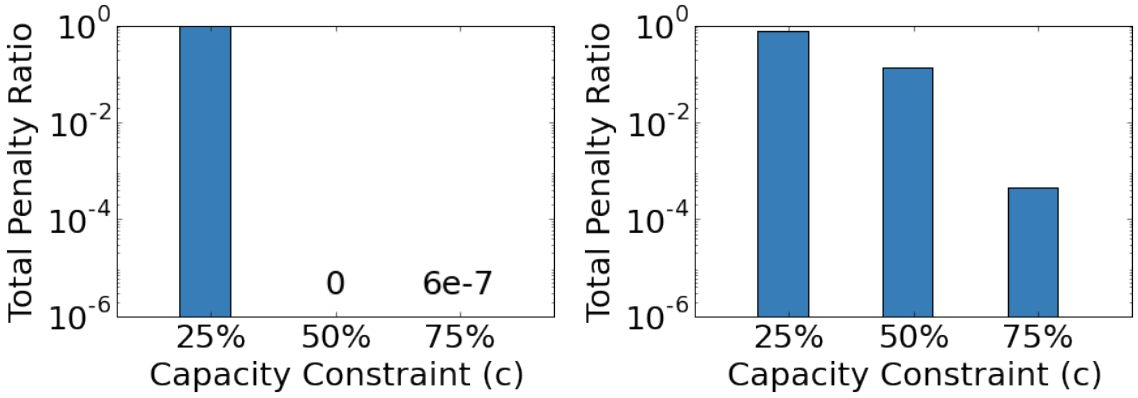
(a) Medium DCN

(b) Large DCN

Figure 3.16: Fraction of available paths to the spine for the worst ToRs when the capacity constraint is 50%, for traces taken from two data centers.

three to six orders of magnitude lower across both data centers.

Fast checker vs. optimizer To isolate the performance gain of fast checker and optimizer, we simulate the large data center network using fast checker alone, which is run both when new corrupting links appear and disabled links are activated. We bin time into one-hour chunks and estimate the total penalty incurred using fast checker alone compared to the full CorrOpt logic. Figure 3.18a shows the total penalty ratio of using CorrOpt versus using fast checker only for a month-long period. We see most of the time, using the optimizer has little benefit. However, during certain periods, it can significantly reduce corruption losses compared to using fast checker alone. Figure 3.18b shows the cumulative distribution function of the ratio of the penalty of CorrOpt over that with using fast checker alone with one-hour bins. The optimizer yields no benefit for 90% of the time. For 7% of the time, using the optimizer reduces the total penalty per second by at least one order of magnitude.



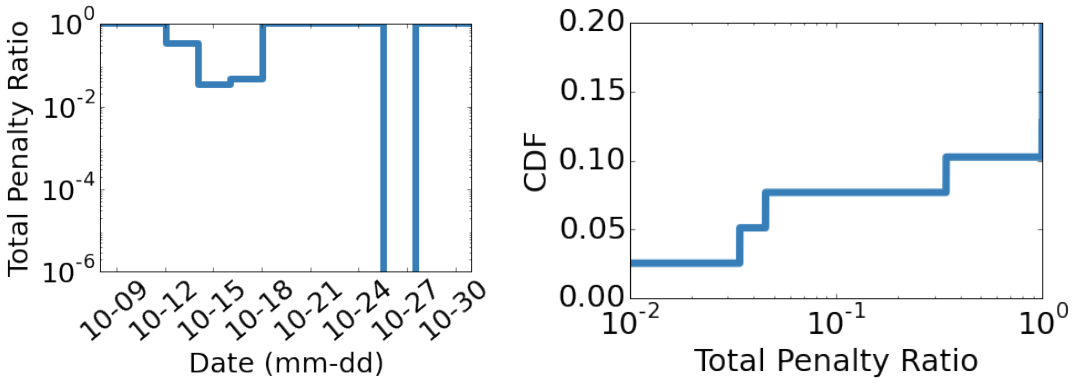
(a) Medium DCN (b) Large DCN

Figure 3.17: Total penalty of CorrOpt divided by switch-local for different capacity constraints, for traces taken from two data centers.

3.6.2 Accuracy of Repair Recommendations

CorrOpt's repair recommendation engine has been deployed across 70 data center networks of different sizes since Oct 2016. Because of certain limitations of the current infrastructure, the deployed version is simpler than the version outlined in Chapter 3.4.2. It uses a single RxPower threshold rather than customizing it to the links' optical technology (information about which was not readily available), and it does not consider historical repairs or space locality. As a result of these simplifications, the results are a conservative estimate of the efficacy of the full system.

To evaluate CorrOpt, we analyze tickets generated between Oct 22 and Dec 31 2016. In this period, it generated close to two thousand tickets with a repair recommendation. Not all generated tickets have a repair recommendation because we cannot get optical power information from all types of switches. We deem repair successful if we do not see another ticket for the same link within a week. Because corruption faults are infrequent, if a link experiences corruption soon after a repaired, it is likely that the repair was not successful.



(a) Penalty Ratio over Time

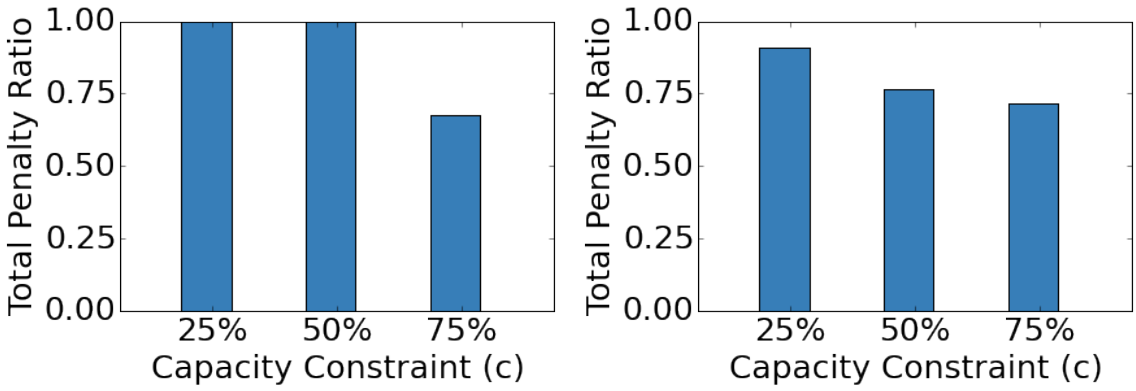
(b) Penalty Ratio CDF

Figure 3.18: Total penalty ratio of using CorrOpt versus using fast checker alone for the large data center network in our trace. (a) The penalty ratio over time. (b) The cumulative distribution function of the ratio over the entire simulation period, for each hour in the trace.

Based on this analysis, the success rate of repair was 58.0%, which is much lower than our expectation. To investigate, we read diaries of 322 tickets. We found that 30% of the time, technicians were ignoring the recommendations! Since CorrOpt is newly-deployed, not all operators have been informed or trained to leverage the information it provides.

When the technicians followed our recommendation, the success rate was 80%. In contrast, our analysis of tickets before CorrOpt's deployment revealed that the previous repair success rate was 50%. The higher success rate of CorrOpt implies the links can be put back into service sooner; at the same time, it reduces the risk of collateral damage that occurs with each manual intervention.

CorrOpt's higher accuracy of repair also lowers corruption losses because it leads to faster repairs. Thi means more healthy links in the data center network, which allows more corrupting links to be disabled while meeting capacity constraints. To quantify this effect, we ran simulations similar to those in the previous section and considered two



(a) Medium DCN (b) Large DCN

Figure 3.19: Penalty ratio comparing impact of CorrOpt repair recommendations on corruption loss. For both traces, CorrOpt's repair recommendations help lower corruption loss.

different repair processes. With CorrOpt, 80% the links are repaired in two days and the rest in four days (i.e., requiring two attempts). Without CorrOpt, 50% of the links are repaired in two days and the rest in four days. In both cases, CorrOpt's link disabling algorithm was in used.

Figure 3.19 shows the results for different capacity constraint for the medium and large data center networks from the earlier traces. The penalty is normalized to that of the setting without CorrOpt repair recommendations. We see that, in addition to their other benefits, CorrOpt repair recommendations reduce corruption losses by 30% when the capacity constraint is 75%, for this simulated scenario for the data centers in our traces.

3.6.3 Combined Impact

We conclude by evaluating the combined impact of CorrOpt's strategy of disabling links and repair recommendations. (Previous sections studied their impact individually.) We compare it to the current practice of using switch-local checks to disable links and 50%

repair accuracy.

In terms of reducing packet losses, the results are similar to those in Figure 3.17. That is because most of the gain stem from its strategy for disabling links, though its higher repair accuracy has other benefits noted above. Overall, in the realistic capacity constraint regime of 75%, CorrOpt reduces corruption losses by three to six orders of magnitude.

Finally, we also find that the massive reduction in corruption losses with CorrOpt does not come at the expense of significantly reduced network capacity. We measure the average fraction of paths to the spine available for each ToR when the capacity constraint is 75%. We find that, compared to the current practice, CorrOpt reduces this average by at most 0.2% across all one-second time intervals.

3.7 Future Extensions

We discuss a few directions for extending CorrOpt.

Accounting for the impact of repair. While disabling a link has limited local effect, we found in our deployment that repairing it can sometimes cause collateral damage. For example, when one link in a breakout cable has packet corruption, to repair the breakout cable, an additional three healthy links have to be turned off. A future extension of CorrOpt would be to account for such collateral impact, e.g., by enabling other (lossy) links while the repair is occuring.

Improving the performance of the optimizer. Our optimizer suffices for today's data center networks, but it may need to be extended for larger networks or for those with more corrupting links. One approach is to divide corrupting links into non-overlapping segments such that the decision to disable them is independent of other segments. Figure 3.20 shows an example. Such segmentation significantly reduces the search space.

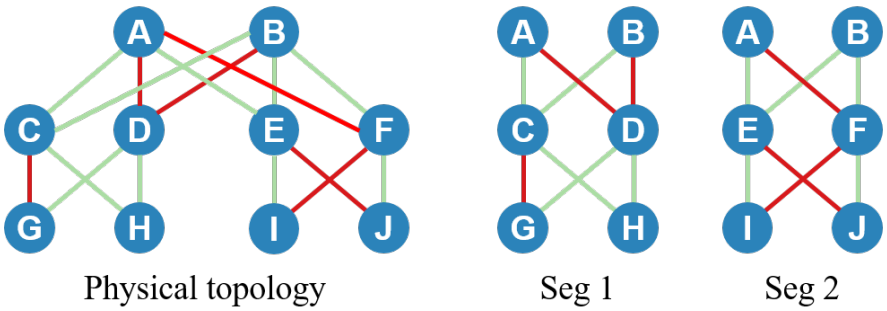


Figure 3.20: Example of topology segmentation. The corrupting links to D affect ToRs G,H, and the leftmost link affects G, resulting in Segment 1. The corrupting link from A to F affects ToRs I,J, with the uplinks of I,J only affecting themselves respectively, resulting in Seg 2. We can thus optimize Seg 1 and Seg 2 independently. Pruning shrinks Seg 1,2 further, depending on capacity constraints.

3.8 Related Work

Our work follows a rich body of work on understanding, diagnosing, and mitigating faults in large, complex networks. While it is not possible to cover everything here, we place our work in the context of the most relevant, recent work.

Packet corruption in data center networks. Some work has acknowledged corruption as a problem. For instance, NetPilot [149] observes that corruption hurts application performance and develops a method to locate corrupting links without access to switch counters.

Faults in data center networks. Many prior studies have considered other types of faults in data center networks [36, 41, 75]. For instance, Gill et al. [74] study equipment failures in data center networks, characterizing different elements' downtime and failure numbers, combined with an impact estimation and redundancy analysis. Our focus is on a different type of fault—packet corruption—and its mitigation.

Fault mitigation. Most work on fault mitigation focuses on congestion or fail-stop faults, using techniques such as load balancing and fast rerouting [146, 28, 40, 101, 150]; however, these are less relevant for corrupting links (e.g., reducing traffic on the link will not alleviate corruption). We view corruption as an anomaly and mitigate it by disabling corrupting links, so they can be repaired. zUpdate [99] and NetPilot [149] depend on knowledge of future traffic demand to further reduce congestion loss when network topology changes. Those techniques are complementary to CorrOpt's link disabling techniques. CorrOpt can work in settings where future traffic demand is not available.

Root cause diagnosis. Using optical-layer characteristics to diagnose network faults was previously proposed by Kompella et al. [93]. Ghobadi et al. [72] use optical-layer statistics to help predict failures in backbone networks. CorrOpt's diagnosis of corruption failures is based on our large-scale study of optical links in data center networks.

3.9 Summary

Our analysis of packet corruption across many data center networks showed that the extent of corruption losses is significant. It also showed that, compared to congestion, corruption impacts fewer links but imposes heavier loss rates, and the corruption rate of a link is temporally stable and uncorrelated to its utilization. CorrOpt, our system to mitigate corruption, lowers corruption losses by three to six orders of magnitude by intelligently selecting which corrupting links to disable while meeting configured capacity constraints. It also generates repair recommendations that are guided by common symptoms of different root causes. This recommendation engine is deployed in Microsoft data centers, where it improved the accuracy of repair, when its recommendations were used, from 50% to 80%.

Chapter 4

RAIL: A CASE FOR REDUNDANT ARRAYS OF INEXPENSIVE LINKS IN DATA CENTER NETWORKS

In this chapter, we take a look at how to reduce the cost of the data center network from a physical-layer perspective. In modern data center networks, all inter-switch links tend to be optical, and the design objective is that *every* link should have a bit error rate (BER) lower than 10^{-12}.[1] To meet this objective, when manufacturers rate transceivers—devices that convert signals between electrical and optical domains—for a certain link length (e.g., 300m), they assume worst case operating conditions (e.g., temperature, signal attenuation due to connectors). These worst-case conditions are rare, and consequently, the vast majority of the links are significantly over-engineered—the optical signal quality is much higher than what is needed to support 10^{-12} BER.

To confirm and quantify physical layer over-engineering in today's data center networks, we conduct what to our knowledge is the first large-scale study of operational optical links. We analyze over 300K links across more than 20 data centers of a large cloud provider over a period of 10 months. We find a remarkably conservative state of affairs—99.9% of the links have incoming optical signal quality higher than the minimum threshold for 10^{-12} BER, with the median 6 times higher.

This over-engineering is expensive—transceivers account for 48–72% of total data center network cost depending on link speed and link length distribution (Chapter 4.6)—and reducing it can lower network costs. While there are multiple ways to do so, we explore

[1]The BER requirement of 10^{-12} was standardized by the IEEE in 2000 [16] and is likely a holdover from the telecom world. In reality, few data center applications today need that level of BER. RDMA is perhaps the most BER-sensitive application today, and a BER of 10^{-10} suffices for it [152]. A lower threshold would likely make our techniques more effective. However, for the purposes of this chapter, we make the conservative assumption that some (future) applications will need 10^{-12} BER.

an approach that does not require any changes to existing hardware. Inspired by the approach of running data centers at higher temperatures than manufacturers' specifications for hardware [59], we suggest data center network operators use transceivers for link lengths that are greater than manufacturers' specifications. This approach can lower costs because transceivers with shorter length specification tend to be cheaper; thus, cheaper transceivers can be used for many links that ostensibly need more expensive ones today.

However, because of transceiver "stretching," some links in the data center network may have BER higher than 10^{-12}. Traffic on such *gray* links will experience higher loss rates because more packets will have bit errors. With the right routing software, the overall impact on application performance is likely to be negligible. Many applications do not need BER of 10^{-12} and can be carried over gray links without hurting their performance. Moreover, since the network has many paths between pairs of top-of-rack switches, applications requiring high reliability can be well supported by being routed through low-loss paths.

Two questions remain about our approach: $i)$ how much can cost be reduced by "stretching" transceivers beyond their specifications? And $ii)$ how can we ensure applications only use paths that meet their loss tolerance? To answer the first question, we perform extensive simulations and experiments with transceivers under realistic conditions. We show that depending on the technology (10 or 40 Gbps) and the desired upper bound on the fraction of gray paths (1% or 5%), current transceivers can be stretched from 1.6 to 4 times their specified length. At these design points, for a fat tree topology, and depending on the link length distribution, the cost of the network can be lowered by up to 10% for 10 Gbps and 44% for 40 Gbps networks.

We answer the second question by designing RAIL. It is a practical system that builds multiple virtual topologies on the same physical topology, where the class of the topology offers a bound on the maximum packet error rate (i.e., grayness) on any path in it. The first-class topology does not have any gray paths; hence, it offers the same path packet error rate guarantee as current data center network designs. Other classes increasingly

use grayer links. Each application uses the virtual topology that meets its needs. Thus, loss-tolerant applications use virtual topologies that have more gray paths. To support applications such as large transfers that are otherwise loss-tolerant but use a loss-sensitive transport protocol (e.g., TCP), RAIL includes a transparent coding-based error correction scheme. We develop an efficient algorithm to compute virtual topologies that leverages the hierarchical topological structure of data center networks (e.g., Clos). RAIL is easily deployable as it requires no changes to the switch or transceiver hardware.

We evaluate RAIL using simulations-based analysis and a testbed. Even at the maximum stretched reach level we consider, we find 95% of all paths are as reliable as today. Furthermore, RAIL protects loss-sensitive applications from gray paths.

4.1 Optical Links' Performance in the Wild

4.1.1 Data Set

To shed light on the performance of the optical layer in today's data center networks, we build a monitoring system that uses SNMP optical MIB [19] to poll optical performance metrics from transceivers every five minutes. Our system runs in multiple Microsoft data centers and collects transceivers' transmit power (TxPower), receive power (RxPower), temperature, transmit bias current, and supply voltage. The TxPower and RxPower values reported by the transceivers are average (across bits 0 and 1) signal power values.

The data we report in this chapter were collected over ten months from Nov 2015 to Aug 2016. They cover over 300K links (600K transceivers), with a mix of multi- and single-mode transceivers with 10, 40 and 100Gbps speeds.

4.1.2 Optical Power Levels Are Too Good

We begin our analysis by studying optical transmit and receive power levels. RxPower is a key indicator of optical layer performance. Modulo significant dispersion, it directly

determines BER.[2] To keep BER under 10^{-12}, RxPower should be above the receiver sensitivity required by the IEEE standard.

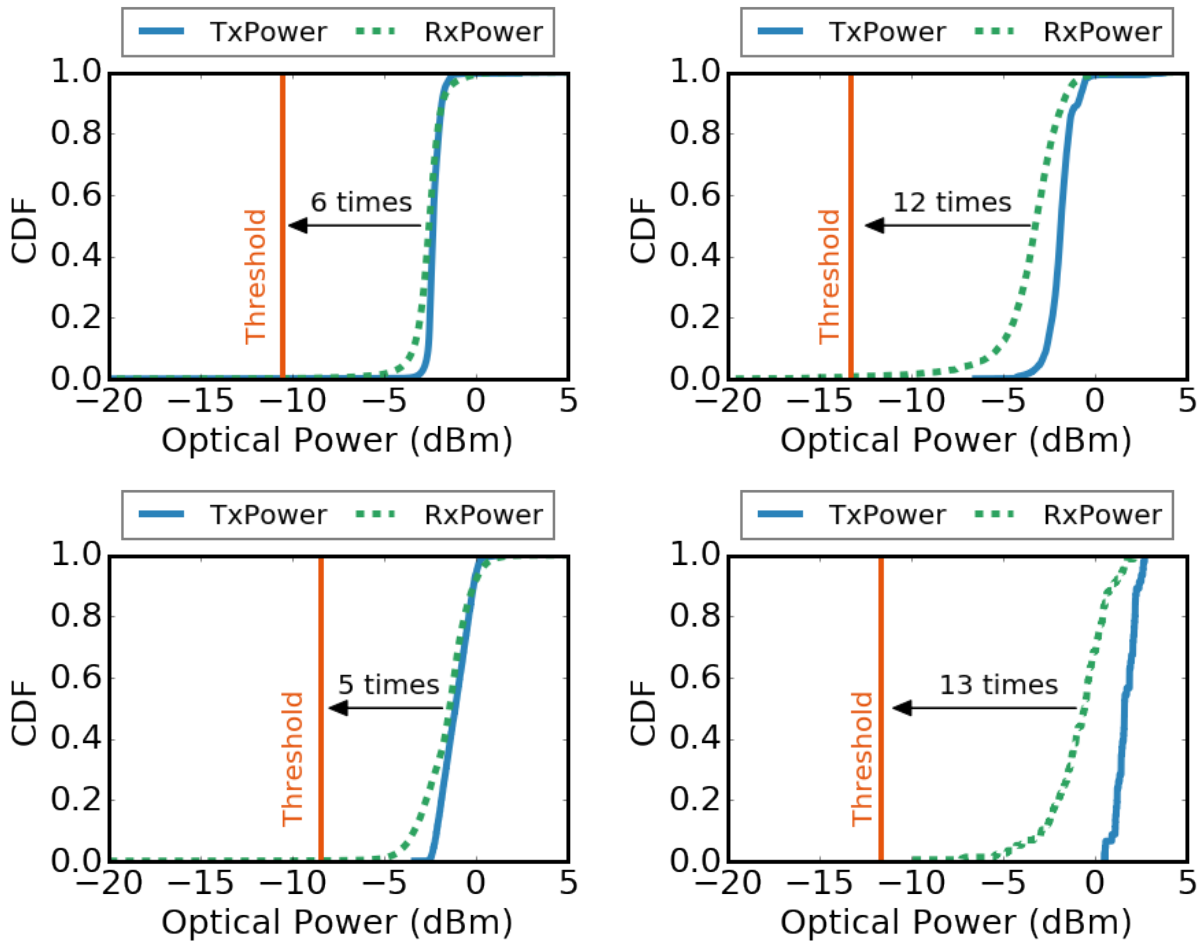
Degree of power over-engineering. Figure 4.1 shows the cumulative distribution function of average TxPower and RxPower of all transceivers separated by their speed and mode (10 Gbps multi- and single- mode, 40 Gbps multi-mode and 100 Gbps single-mode). The vertical line in each plot is the receive power threshold to keep BER under 10^{-12}. We see that across the board nearly all links (99.9%) have RxPower above the threshold. In fact, 99% of transceivers are at least 5 dB (i.e., 3 times) above it. Thus, when we use the IEEE standard as our basis of comparison, we see the vast majority of links are greatly over-engineered from an optical power perspective.

Over-engineering across transceiver models. We find this degree of over-engineering is consistent across transceiver manufacturers. Our data include over 50 transceiver models across various vendors. Figure 4.2a plots the cumulative distribution function of RxPower of 10 Gbps multi-mode transceivers across five transceiver manufacturers each with more than 1000 transceivers inside one data center. For confidentiality, we do not report the name of the manufacturers. All five manufacturers exhibit similar RxPower distributions, and over-engineering is not tied to one manufacturer.

Over-engineering across data centers. We also study whether over-engineering varies by data center. Figure 4.2b plots power levels of the ten largest data centers in our dataset. Each color represents a data center. We see that RxPower distributions and thus over-engineering levels are similar for every data center.

Optical power variation over time. Over the course of our study, we found power levels of a link vary little over time. Figure 4.3a illustrates RxPower over time for two sample

[2]Most transceivers today do not report BER. Switches report packet error rate, but this measures the combined impact of errors from all sources, including electrical components.



(a) 10 Gbps multi-mode

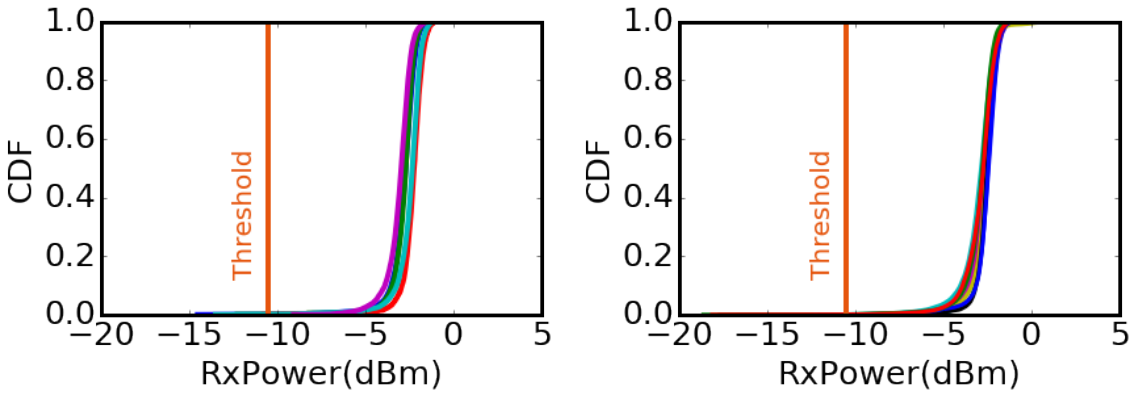
(b) 10 Gbps single-mode

(c) 40 Gbps multi-mode

(d) 100 Gbps single-mode

Figure 4.1: Cumulative distribution function of TxPower and RxPower for different speeds and modes across links in our trace. The vertical lines are the RxPower threshold needed to keep BER under 10^{-12}. Almost all links across all technologies have RxPower above the threshold.

transceivers. As shown, for each transceiver, RxPower remains mostly stationary over time. Figure 4.3b plots the CDF of maximum minus minimum RxPower value for all transceivers. The figure shows 78% of the links have a variation below 0.2 dB, and over



(a) Different manufacturers

(b) Different data centers

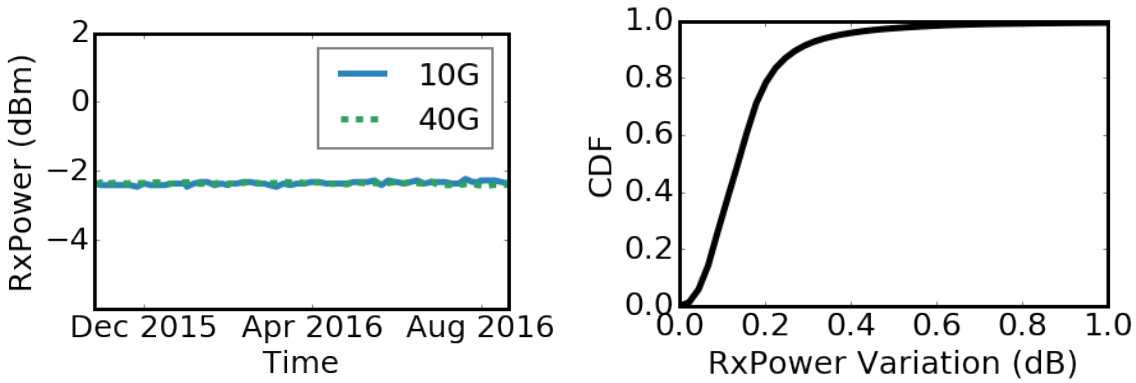
Figure 4.2: Cumulative distribution function of RxPower of 10Gbps multi-mode transceivers across five transceiver manufacturers and ten different data centers. Over-engineering is not limited to one manufacturer or data center.

99% have a variation below 0.8 dB. Thus, links with high RxPower have consistently high RxPower, instead of power levels dropping intermittently (which would cause high BER during those times if we were to reduce their over-engineering).

4.1.3 Understanding Low Optical Power

In Figure 4.1, unlike TxPower, RxPower has a long tail, suggesting that a small fraction of links experience high attenuation and thus low RxPower. Figure 4.4a shows this effect directly by plotting the cumulative distribution function of attenuation for all multi-mode links; single-mode results are similar. We compute attenuation as TxPower at the sender minus RxPower at the receiver. The figure shows most links have attenuation close to zero—the median is 0.26 dB—but 0.44% have attenuation higher than 5 dB.

Initially, we thought high attenuation would correspond to links that are long or have many optical connectors. To confirm this supposition, we studied *attenuation symmetry* between the two directions of a bi-directional optical link. If link length or connector



(a) RxPower vs. time.

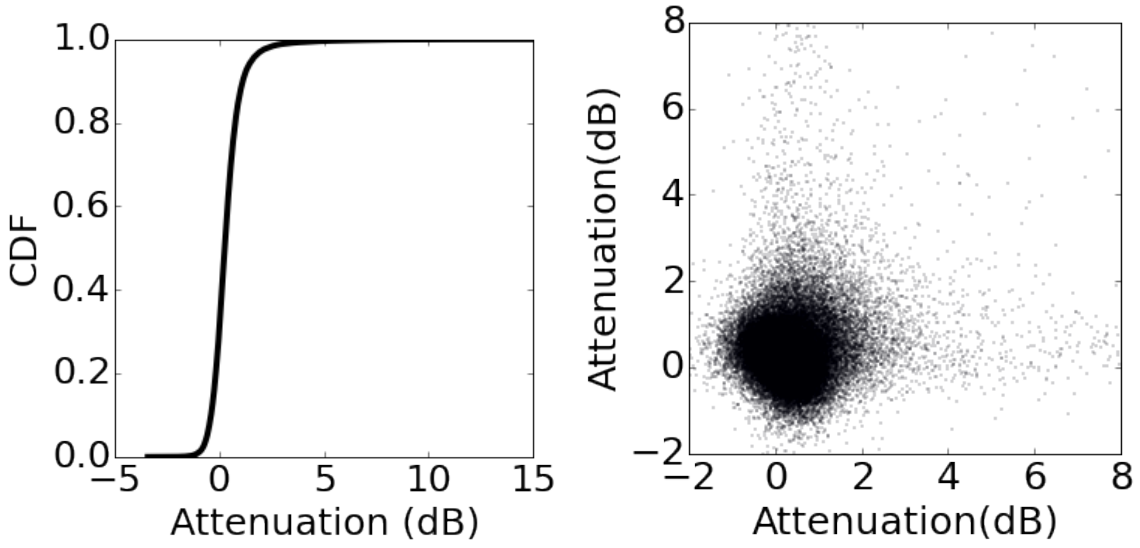
(b) (max - min) RxPower

Figure 4.3: RxPower variation of individual transceivers over time. 10 Gbps and 40 Gbps lines are overlapping in the left graph. The right graph is the cumulative distribution function of maximum RxPower variation over time across all transceivers in our data set.

count are to blame, because these factors are identical in both directions, high attenuation should be symmetric. Figure 4.4b shows a scatter plot where the two axes represent attenuation levels in different directions. We see that attenuation is low and symmetric for most links.[3] But it is asymmetric when it is high. When one side has high attenuation, the other side does not.

This behavior suggests that high attenuation does not stem from link length or connector count but due to poor or dirty connectors or fiber damage—connectors and fibers are different in the two directions. To confirm this hypothesis, we analyzed hundreds of repair records for poor optical links. We found two main root causes for low RxPower: (i) dirty fiber connectors (which must be cleaned to fix the problem); (ii) damage to the fiber's cladding, the outside layer of the fiber that protects the actual core. The finding that high attenuation rarely stems from long links is consistent with our later experiments showing

[3] Attenuation levels below 0 are due to (unbiased) calibration error in TxPower and RxPower sensors. The reported power is within $\pm$ 2 dB of the actual value.



(a) CDF of Attenuation

(b) Bi-directional Attenuation

Figure 4.4: Attenuation for 10 Gbps multi-mode optical links. The left graph is the cumulative distribution function of average attenuation over time. In the right figure, x-axis shows the average attenuation in one direction of an optical link, and y-axis shows the avergage attenuation in the opposite direction.

current transceivers have low BER even on links that are longer than their specification.

Dependence on link location. We also find that low RxPower links are scattered across the data center uniformly and randomly, and they are not correlated with a specific switch brand or topology tier. To confirm, we compute the percentage of switches having links with RxPower at the bottom 0.01% of Figure 4.1. We then uniformly and randomly select 0.01% of links with any value for RxPower and again compute the percentage of switches to which they belong. If the two numbers match, it suggests links with RxPower at the tail are scattered uniformly and randomly across switches. We repeat this analysis for 100 values between 0.01%-tile and 1%-tile tail and observe the same result. Figure 4.5 shows

that the numbers based on this independence assumption closely match the data. If some switches were more likely to have links with low power levels, the "Low RxPower data" curve would be consistently lower than the "Uniform Random" curve. This observation, together with our observation on the root causes of high attenuation, suggests that dirty connectors and damaged fiber can show up anywhere in the data center.

4.2 Reducing Over-engineering and Cost

Our measurements above reveal that the optical layer of data center networks today is heavily over-engineered. This over-engineering does not come for free; data center networks are expensive to build, with transceivers accounting for 48-72% of the network cost, depending on link speed and link length distribution (**??**). We shared our measurement results with experts in transceiver manufacturing companies and learned of many possibilities to lower the price of transceivers (e.g., changing transceiver hardware, relaxing test requirements, reducing labor and packaging cost).

However, we also learned that immediate realization is difficult because of two intertwined challenges. First, transceiver manufacturers cannot reliably estimate cost savings without carefully crafting and optimizing the entire manufacturing chain, and they are reluctant to do so without standardization or firm commitments from data center network operators. Second, any method of reducing over-engineering can cause some gray links (with BER higher than 10^{-12}). Thus, network operators are reluctant to commit unless the cost savings are known to be high, and there is a way to protect sensitive applications against gray links.

In this chapter, we address both challenges. First, we demonstrate immediate cost savings are possible by allowing commodity transceivers to be "stretched," that is, used for distances longer than their current specifications. Second, we devise a system for routing and forwarding packets when some links may be persistently gray. Leveraging application and path diversity in data center networks enables applications to use paths according to their loss-tolerance (Chapter 4.3).

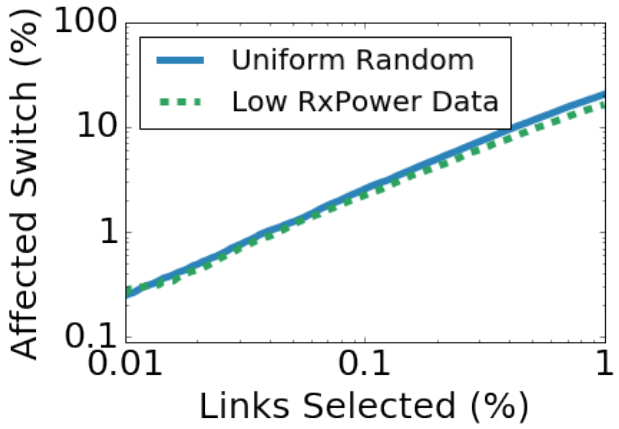


Figure 4.5: Links with low RxPower are uniformly and randomly scattered across switches.

Below, we explain how stretching reduces cost and show how to engineer a stretched network.

Cost reduction through stretching. Figure 4.6 shows the price and specified reach of different standard transceiver technologies available in the market for 10, 40, and 100 Gbps. The price of 10 and 40 Gbps transceivers represents the average across three volume retailers [21, 20, 9] and the price of 100 Gbps transceivers is from one retailer [15]. As shown, the standard includes discrete reaches, with the price of transceivers increasing with read. Longer reach requires more expensive components (e.g., narrow spectrum laser, cooling module) and manufacturing processes.

Figure 4.7 illustrates how stretching transceivers reduces cost. The solid line is price for standard reach, and the dashed line shows the stretched reach. When the reach is stretched from R_1 to R_1', we don't have to pay to use an R_2-rated transceiver for distances between R_1 and R_1' and, hence, we can save cost. The exact savings depends on how much transceivers can be stretched and the relative pricing of different options. We quantify the savings for available standard technologies in Chapter 4.6.

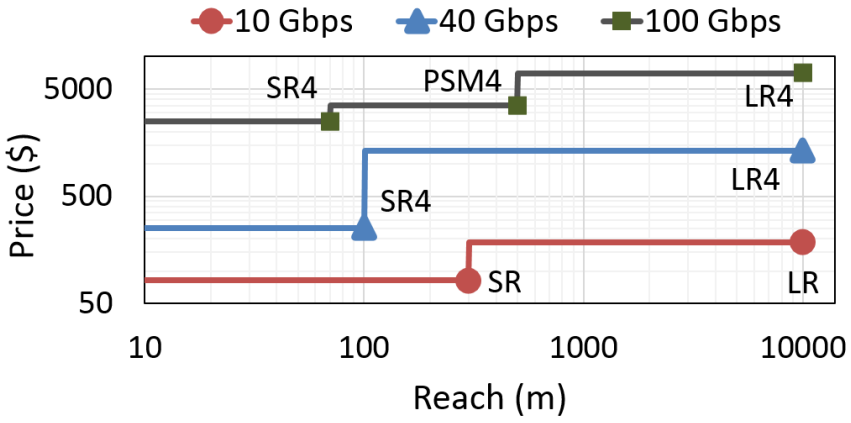


Figure 4.6: Price and reach of standard transceivers for 10 Gbps, 40 Gbps and 100 Gbps technologies. Both X and Y axes are in log scale. The label beside each data point is the name of the technology. PSM4 [25] is not yet an IEEE standard but is an agreement between 12 companies.

We don't expect transceivers' lifetimes to be shortened by stretching because lasers are designed for a long life-time, and TxPower does not decrease over time (96.7% of transceivers' TxPower does not change more than 0.2 dB over our 10-month measurement period).

Determining how much to stretch. To determine how much transceivers in a data center network can be safely stretched, we use a metric called *network reliability bound* (NRB). NRB is a lower bound on the expected fraction of paths that the network administrator desires to be good (i.e., those where all links have BER below 10^{-12}). We compute the maximum stretch for a transceiver based on NRB and i) the maximum number of links in a data center network path, and ii) the BER distribution expected for different stretch levels. We show in Chapter 4.6 how to compute this distribution using expected attenuation distribution.

This computation is best illustrated in an example. Suppose our goal is for NRB to be

95%, which can be translated to each path being good with a probability of at least 95%. In a 3-stage Clos network, where the maximum number of hops is four, this goal can be met if each link is good with probability of $\sqrt[4]{95\%} = 99.7\%$ (assuming that links being good or bad is independent of location, as we showed in Chapter 4.1.3). From the BER distribution for different stretch levels, we can now determine the stretch at which 99.7% of the links will be good. We use this value as the maximum stretch for a transceiver. In reality, the network will have more than 99.7% good links because many links where "stretched" transceivers will be used are shorter than the maximum stretch.

Need for software layer protection. NRB enforces a lower bound on the fraction of good paths, allowing a small fraction of gray paths. To preserve application performance, a naive solution is to simply turn gray links off and thus remove all gray paths. But even if the fraction of gray links is small, turning them off can result in 20 to 50% capacity loss for certain ToR-pairs (Chapter 4.6). Instead, we propose that a software system be used to effectively utilize such gray links, while protecting loss-sensitive applications. The design of RAIL, described next, accomplishes this goal.

4.3 Overview of RAIL

RAIL is a system for routing and forwarding in a data center network, where $i)$ links have a range of link packet error rate (LPER); and $ii)$ applications have different requirements for path packet error rate (PPER). In such a network, RAIL ensures an application is routed only through *paths* with its desired reliability level (or better). Thus, sensitive applications (i.e., RDMA-based ones) are routed only along the most reliable paths while tolerant ones (i.e., UDP-based ones) can be routed through any path. RAIL is a general solution that is agnostic to why gray links occur; they could be caused by any method of reducing over-engineering, including stretched transceivers, cheaper hardware, or cheaper fiber.

We want our solution to be: $i)$ readily deployable, i.e., requiring minimal changes to

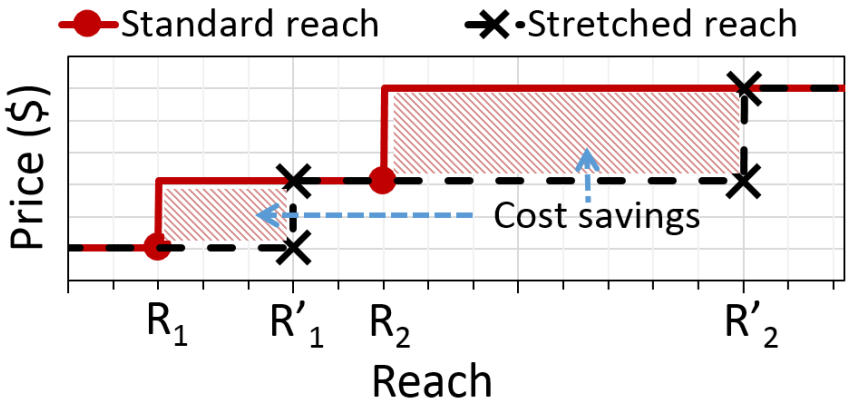


Figure 4.7: Illustration of how stretching transceivers reduces cost. When R1-rated transceiver is stretched from R_1 to R'_1, we do not have to use (the more expensive) R_2-rated transceiver for link lengths between R_1 and R'_1.

current infrastructure; and ii) practical, i.e., having low overhead and complexity. To appreciate these constraints, we consider two extreme design possibilities with respect to how much hosts need to know about the network. The first is source routing, in which hosts are responsible for selecting paths (i.e., sequence of links to the destination) through the network that meet application needs. The challenge with this approach is that hosts will need an up-to-date view of network topology and LPER. With hundreds of thousands of links and frequent link failures [74], maintaining such a view at every host is highly complex.

In the second design, hosts are not told anything about the network (as is the case today). To deliver reliable communication, they use network coding transparently; i.e., application traffic to a given destination is coded such that it can withstand some fraction of loss. The problem with this approach is that it sets up an unwanted trade-off between coding overhead and the loss rate experienced by traffic. Hosts do not know in advance which path a given flow might take (due to ECMP routing). If they encode every flow to a level that guarantees the most sensitive applications do not suffer (i.e., encoding

based on the least reliable path), the coding overhead will be onerous for most paths. If they encode based on the average path loss rate, sensitive applications that traverse worse paths will suffer. It is possible to adapt by learning path loss rate based on what is actually experienced by a flow, but this is complicated by the fact that flows are randomly assigned to paths and most flows are short and thus do not generate enough packets for robust learning.

To be deployable and practical, RAIL explores a design point in the middle. On the same physical topology, it builds k *virtual topologies*. Each topology falls in a different *class* representing a bound on the worst Path Packet Error Rate (PPER) in the topology. PPER represents the packet corruption probability for a complete ToR-to-ToR path and is derived from each hop's LPER. The first-class topology is made exclusively of the most reliable paths in the physical topology, the second-class topology permits all paths with PPER less than a bound, and the k-th class topology may include all paths in the physical topology. Figure 4.8 illustrates this concept with k=3, that is, three virtual topologies. Within a virtual topology, routing and forwarding will follow the same protocol as DC operators prefer today for their physical topology (e.g., ECMP over equal hop paths). Figure 4.8a shows the physical topology with two types of links: some with LPER 10^{-8} and three gray links with LPER 10^{-5}. RAIL creates three virtual topologies shown in Figure 4.8b, Figure 4.8c, and Figure 4.8d. Virtual topology (b) guarantees PPER of 2×10^{-8}. Virtual topology (c) guarantees PPER of 1×10^{-5} since its worst paths are A-D and C-E. Virtual topology (d) guarantees PPER of 2×10^{-5} since its worst path is D-A-F. Thresholds of these virtual topologies are configurable.

Virtual topologies are exposed to end hosts as different (virtual) interfaces. Applications (or the hypervisor on their behalf) bind to the interface that reflects their reliability criteria. Thus, the most-sensitive applications (e.g., RDMA-based ones) may bind to the interface for the first-class topology, TCP flows may bind to the second-class topology, and bulk transfers may bind to the k-class topology. Beyond making this choice, hosts are not aware of RAIL.

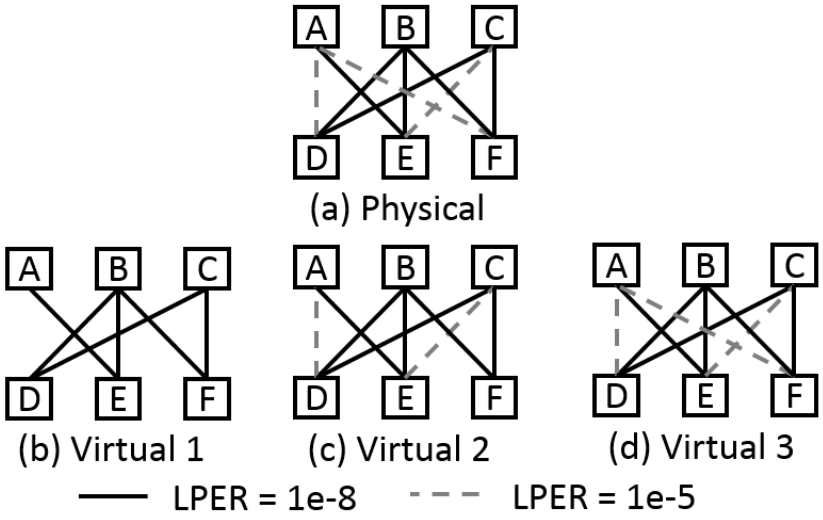


Figure 4.8: Example of RAIL's virtual topologies. Solid lines are links having 10^{-8} LPER. Dashed gray lines are links having 10^{-5} LPER. RAIL maintains three virtual topologies (b, c, d) depending on Path Packet Error Rate (PPER).

Applications that bind to less reliable topologies may be already robust to small amounts of loss they may experience, or they can be made that way by using a version of TCP that is robust to corruption-based losses [37, 107]. If this is not the case, however, RAIL includes a module that uses coding to transparently enhance traffic reliability.[4] Designing this module is an easier task than the coding-based option outlined above; flows going over less reliable topologies are likely longer, giving our module a chance to learn the path being used and select an appropriate coding level.

In the following section, we describe RAIL's design in more detail. As will become clear, it requires no changes to existing switches and only small change to the host software (i.e., picking among virtual topologies, error correction).

[4]Retransmitting lost packets is another (more efficient in terms of byte overhead) way to improve traffic reliability. We explore coding because it offers a faster, proactive way to recover lost data.

4.4 RAIL in Detail

This section describes virtual topologies, routing and forwarding, and error correction mechanisms in RAIL. It also provides guidelines on configuring RAIL.

4.4.1 Virtual Topology Construction

Virtual topologies in RAIL share the same physical fabric but offer different guarantees for the maximum packet corruption rate along any of their paths. Our task is to build K virtual topologies, where the k-th topology offers the worst-case PPER of L_k. K and $\{L_k\}$ are selected by network operators based on their applications (see Chapter 4.4.4).

These topologies are built by the RAIL controller, which maintains an up-to-date view of each link's LPER by polling the switches. Since link qualities are relatively static (Chapter 4.1), maintaining such a view is straightforward. In addition to providing a worst-case PPER guarantee, we want each virtual topology to include as many links as possible to maximize its capacity. Because optimally finding such a topology is computationally expensive, we resort to a fast, greedy algorithm. Speed is important because, while link qualities are relatively static, links do fail frequently [74], and we need to recompute virtual topologies when links fail or recover.

Our algorithm has K rounds, one per topology. In each round, it starts with a set of candidate links for the topology and iteratively removes low reliability links until the required guarantee can be met. We start with the K-th class (least reliable) topology. For it, all links (that are currently alive) are initial candidates. We find the ToR-to-ToR path with the worst PPER, and if that PPER is higher than the required bound, we remove the link (on the path) with the worst LPER. Such link-removal iterations are repeated until the worst-case path meets the required bound. We then begin the next round, for the topology that is one class lower, starting with links not removed in the previous round.

The speed of the above algorithm depends on how quickly we can check whether a topology meets a PPER requirement. Tracking every path's PER takes at least $O(n^3)$ in a

n-ToR Clos topology (as there are $O(n^2)$ ToR pairs with $O(n)$ paths between each pair). Such running time is computationally infeasible on realistic network controllers. Our algorithm speeds up this running time to $O(n \log n)$ and is linear in the number of links in the topology. The intuition is that because of a data center network's highly structured topology and its simple routing scheme, we can track the worst path without tracking the PPER on every path.

Our algorithm goes through each switch exactly once and computes three values on each switch, from the bottom stage all the way to the top stage. The three values computed on switch s are (1) up_s: PPER for the worst monotonic upward path from any ToR to s, (2) $down_s$: PPER for the worst monotonic downward path from s to any ToR, and (3) top_s: PPER for the worst up-down path for which s is the highest stage switch.

Definition 1. *Children of switch s are the set of switches on a lower stage than s, with direct links to s.*

Algorithm 2 Finding the worst up-down path for which s is the highest stage switch.

$up_s \leftarrow 0$
$down_s \leftarrow 0$
$top_s \leftarrow 0$
for $c \in Children(s)$ **do**
    $up_s \leftarrow max(up_s, 1 - (1 - up_c)(1 - LPER_{c \to s})$
    $down_s \leftarrow max(down_s, 1 - (1 - LPER_{s \to c})(1 - down_c))$
for $c, d \in Children(s), c \neq d$ **do**
    $tops_s \leftarrow max(top_s, 1 - (1 - up_c)(1 - LPER_{c \to s})(1 - LPER_{s \to d})(1 - down_d))$

After those three numbers are calculated, our algorithm outputs the worst path by picking the worst top_s among all switches in the network. The worst PPER is simply $max_{s \in \textit{all switches}}(top_s)$. The correctness proof and running-time analysis appear in Appendix C.

4.4.2 *Routing and Forwarding*

To simplify routing and forwarding, we use a non-overlapping IP address space within each virtual topology. We configure switches such that, when routing or forwarding for a topology, they ignore links that are not part of that topology. The exact mechanism will depend on the routing paradigm used by the data center. If the data center uses a distributed protocol such as BGP to compute paths [97], we configure BGP to not announce prefixes for a virtual topology over links that are not part of it. No RAIL-specific changes are made to switch software, and they will forward packets as they do today (e.g., ECMP).

If the data center uses an SDN controller to centrally compute forwarding paths, we can either instantiate one controller per virtual topology or use one network-wide controller programmed to not use certain links for given prefixes.

Our approach may create uneven load on links because different links are part of different topologies. Load can be uneven even without our modifications, however, with traffic engineering is required to balance it [151, 28, 133]. We leave the task of extending traffic engineering to account for virtual topologies for future work.

4.4.3 *Error Correction*

When the application or transport protocol is not robust to small amounts of PPER (corruption-based loss), RAIL's error correction module can be used to guarantee high performance in exchange for slight bandwidth overhead. This module is completely transparent to applications.

As argued earlier, given the diversity of PPERs across paths, it is important that error correction be based on the PPER of each path, rather than being guided by the worst-case or average PPER in the virtual topology. Recall that because of ECMP-hashing, hosts are not aware of the path taken by a flow.

RAIL's error correction module learns the PPER in two steps. First, as soon as a new flow starts, the source host sends a `traceroute` probe with a header (5-tuple) identical

to that of the flow. This probe reveals the path taken by the flow. We use special DSCP bits in the IP header of the probe packet to indicate to the destination host module that it should not deliver the packet to the application. Second, the error correction module queries RAIL's controller for PPER of the path.

Ideally, the error correction module should be able to use bit-level forward error correction (FEC) for each packet. However, this approach does not work in practice because today's switches drop packets when the CRC checksum fails. As we are seeking an immediately deployable solution, we do not consider this option. The main cost is that end-to-end packet-level FEC has higher coding overheads. As we show in Chapter 4.6, the bandwidth overhead of our error correct module is low enough despite this inefficiency.

RAIL sends "parity" packets after every n data packets, where n is based on the path packet loss rate (see below). We use XOR encoding because it is lightweight and known to be effective [135]. That is, after every n data packets, the sender sends a packet whose content is the XOR of the previous n packets. In this coding scheme, as long as n out of the $n+1$ packets are successfully delivered, the receiver can recover the original n packets. Losing two or more packets within a group of $n+1$ packets results in data loss.

If PPER is p, the probability of having two or more losses among $n+1$ packets is $1-(1-p)^{n+1}-(n+1)p(1-p)^n$. We pick n such that this probability is lower than the desired post-recovery loss probability t (experienced by applications). Any path with $p < t$ does not use any error correction. To show an example of computing n, we first quantify t for a particular transport. Suppose TCP's performance degrades when the loss rate is above 0.1%; therefore, we can pick t=0.1%. For a path loss rate of $p = 0.3\%$, we would choose n to be 14 so that the post-recovery loss rate is again $0.092\% < 0.1\%$. The bandwidth overhead in this case is 7.1%.

For a given virtual topology, we include all paths meeting the loss criteria; thus, most paths are as reliable as the IEEE standard requires. Even if coding overhead is high for a particular flow, the average overhead will be small. We show later (Chapter 4.6) that the number of good paths for which error correction is unnecessary dominates the total

number of paths.

4.4.4 Configuring RAIL

While it is up to individual operators to configure the number and loss rate guarantee of virtual topologies in RAIL, based on their applications, we offer a simple recommendation. It is based on the double observation that many applications use TCP, and the performance of some TCP variants degrade noticeably only when loss rate exceeds 0.1% [120, 46] for today's data center networks. We see this behavior in our experiments, and it is consistent with what others have reported. That is why some operators completely switch off links with error rates higher than 0.1% [149].

We recommend that data centers be configured with three virtual topologies. The first-class topology should provide paths with the same reliability as today, equivalent to where each link has BER less than 10^{-12}. This can be used to carry the most sensitive applications, such as RDMA. The second-class topology should provide paths with PPER below 0.1%, and it should carry applications like short TCP flows. Finally, the third-class topology should provide paths with PPER below 10%, and unless the application uses loss-tolerant transport, it should be error corrected. When the application is a long TCP flow, it should be error corrected to a reliability of t=0.1%. RAIL always put RDMA traffic on first-class topology and never uses error-correction code with RDMA. A corollary of our recommendation is that any link with PPER above 10% will be turned off entirely for repair. Such links are rare (Chapter 4.6).

4.5 Implementation

Our implementation of the RAIL controller and the error correction module includes the following features. The controller learns link PERs from CRC error counters. It does not distinguish between optical- versus electrical-related corruption and provides protection from both. It constructs three virtual topologies with worst-case PPER of 4.8×10^{-8}, 0.1%

and 10%. The controller computes routing tables globally based on the virtual topologies and pushes rules to switches accordingly.

Our implementation of error correction sits below the kernel TCP/IP stack so that it is oblivious to the transport protocol. We implement it as a driver for tun/tap device in the Linux kernel. (On Windows, a WinSock kernel device driver may be used.) The driver keeps a buffer of size n so that it can decode the coded packet if needed and deliver packets to higher layers in order. It keeps a fine-grained timer such that if a missing packet is not recovered within a short time window, the next packet is delivered to the transport protocol (e.g., TCP). This delivery may trigger a recovery at the transport layer. Such transport-layer retransmissions are new packets for our error module.

To identify packets for coding and decoding, we insert a 4-byte header after the IP header containing a sequence number. Once the encoding rate is negotiated, every packet is geven a sequence number. The parity packet has the last sequence number in each group of n. As the error correction module knows the exact path to the destination, it performs cross flow error correction among all the flows with the same path.

4.6 Evaluation

Our evaluations are divided into three categories. First, we use a testbed and an optical simulator to quantify the stretch of two widely used short reach transceivers and compute the resulting BER and potential cost savings (Chapter 4.6.1). Next, we evaluate the impact of stretching these transceivers on the overall network path quality (Chapter 4.6.2). Finally, we show RAIL preserves applications' performance in a network with gray links (Chapter 4.6.3). Our results demonstrate that RAIL has minimal impact on overall network quality and application performance, while reducing total network cost by up to 10% for 10 Gbps networks and up to 44% for 40 Gbps networks.

4.6.1 *Stretching Existing Technologies*

We experiment with eight IEEE-standard short-reach 10 Gbps and 40 Gbps transceivers (two brands for each speed and two units of each brand). Some manufacturers provide non-standard technologies with reach values that are not supported by IEEE. These transceivers can likely be stretched as well if they follow similar specification practices, but evaluating individual non-standard transceivers is beyond the scope of this chapter. Our goal rather is to demonstrate that commodity transceivers are over-engineered and can be stretched beyond their specification. We also experimented with a standard 100 Gbps transceiver. Those results are preliminary and appear in Appendix D.

Experimental methodology. Our stretch experiments are based on a testbed and an optical simulator. Our testbed has one 10 Gbps [2] and one 40/100 Gbps switch [3], four 10GBASE-SR transceivers [10, 4], four 40GBASE-SR4 transceivers [11, 4] and a set of OM3 fibers [5] of lengths between 10m and 1000m. We focus on SR (short reach) technologies as they are viable candidates for stretching; the reach of LR (long reach) technologies is longer than typical maximum data center link lengths.

To emulate long fibers of different lengths, as shown in Figure 4.9, we concatenate multiple short cables with fiber connectors. To emulate additional attenuation in real environments, due to more/dirty connectors or damaged fiber, we insert a variable attenuator which adds additional attenuation from the distribution shown in Figure 4.4a.

Since our transceivers do not directly report BER, to infer their BER, we measure packet corruption rate (LPER) with full-sized, line rate traffic for five minutes. We then calculate BER from LPER using a simplified model:[5] $LPER = 1 - (1 - BER)^{PACKET\ SIZE}$.

Testbed experiments are useful to provide coarse data on how link BER changes with different link lengths and attenuation levels. To explore the parameter space in fine gran-

[5]This model may overestimate BER, and this would underestimate potential for stretch because of two factors. First, some packet corruption events may be due to non-optical issues. Second, Ethernet uses line code (e.g., 64b/66b for 10 Gbps network), and any bit flip in the code causes an extra 2 bits to be corrupted in the future, possibly in subsequent packets.

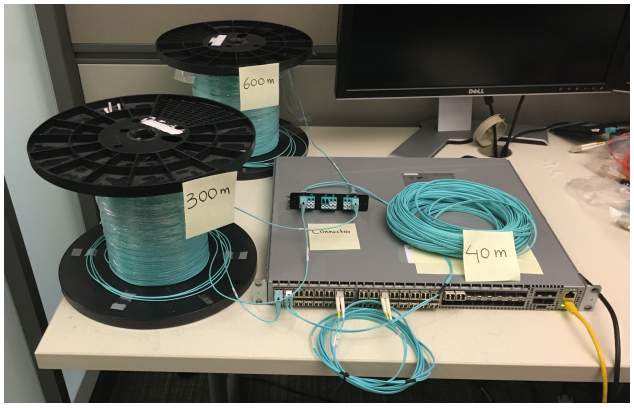
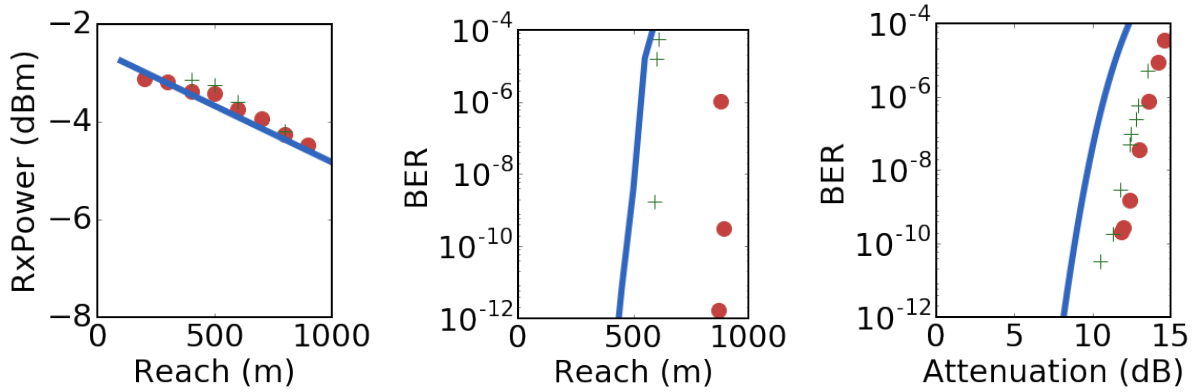


Figure 4.9: Our 10GBASE-SR testbed. We stitch short fibers together with fiber connectors to emulate long fibers. We concatenate 600m, 300m, 40m fibers together to emulate a 940m fiber. Two 300m 10GBASE-SR Finisar transceivers [10] are attached to the fiber on both ends. Switch ports are on and we can send traffic through with BER of around 10^{-8}.

ularity and to eliminate hardware quality differences between manufacturers, we use VPI [23], a standard optical simulator for data transmission system. (We cannot use a closed-form formula to compute BER based on fiber length because of complex dispersion effects.) Our simulations model laser characteristics and a laser driver in the sender, modal and chromatic dispersion in the fiber, loss on the connector, and receiver sensitivity and dispersion equalization chips in the receiver. We configure these parameters based on the transceivers' specification sheet.[6]

For both 10 Gbps and 40 Gbps, we validate our simulator along three dimensions: RxPower, BER, and attenuation. Figure 4.10 shows our validation results for 40GBASE-SR4; the results are similar for 10GBASE-SR4. Figure 4.10a shows RxPower as fiber length increases. The scatter dots are testbed results, and the solid line is the result of our simulator. The figure shows that the simulator is able to closely match the RxPower of both transceiver brands for all fiber lengths. Figure 4.10b shows BER as fiber length increases.

[6]Our simulation files are available online [26]; other researchers can use these to simulate optical links in data center networks.



(a) RxPower vs. reach (b) BER vs. reach (c) Attenuation vs. BER

Figure 4.10: Validation for 40GBASE-SR4 on OM3 fiber. Blue line corresponds to simulations. Circles and crosses are testbed results for transceivers from different manufacturers. (a) RxPower as a function of fiber length (no added attenuation). (b) BER as a function of fiber length (no added attenuation). (c) BER as a function of added attenuation when fiber is at transceiver's specified reach.

Since BER depends on a transceiver's sensitivity to modal dispersion, the two transceivers do not necessarily have to match. Hence, we configure our simulator to be the most sensitive one of the two brands and use the more conservative results in our evaluations for the rest of this chapter. Similarly, Figure 4.10c shows that our simulations capture BER vs. additional attenuation conservatively. In this experiment, we use a 100m fiber, the design limit of our 40GBASE-SR4 transceivers, and introduce additional attenuation using a variable attenuator.

BER distribution versus link length. We can now derive the distribution of BER that a link will observe as a function of its length. This distribution is a function of both link length and additional attenuation caused by dirty connectors or damaged fiber in real deployments. To simulate how stretched links impact BER in real world, we add the

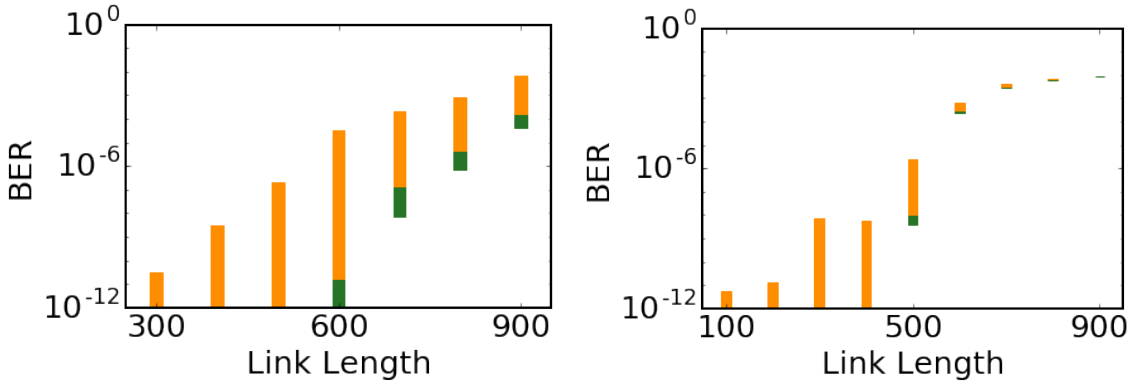
attenuation distribution seen in our measurement data in Chapter 4.1. This method is conservative because it assumes that all the attenuation measured in the wild is caused by factors other than link length. In practice, link length contributes as well, and our simulator already includes link length.

Figure 4.11 shows the BER distribution that will occur for various link lengths. The BER is represented as a bar for each link length. The top of the bar represents when extra attenuation is 99.9%-tile value (from the attenuation data) and the middle, which separates the two colors, when it is 99%-tile. As the figure shows, when we use 10GBASE-SR, which is rated for 300m, on 500m links, at least 99% of these links will have BER less than 10^{-12}. At the same performance level, we can stretch 40GBASE-SR4, which is rated for 100m, to 400m.

Cost savings. The level of stretch that a network should use depends on the trade-off between cost savings and performance, which we quantify using the NRB metric defined in Chapter 4.2. More stretch means more cost savings, but it also means that a larger fraction of paths will have losses.

We illustrate this trade-off using a standard 3-stage 10 Gbps Clos network. A data center network's total cost includes the equipment costs (i.e., transceivers, fibers, switches) and switch power consumption. Switches are \$90/port [127], multi-mode and single-mode fiber cost \$0.44 and \$0.21 per meter, respectively [7]. Each switch consumes around 150 Watts (this includes energy consumed by transceivers). With 32-port switches, we can build a full bisection 3-stage fat tree network with 512 ToR, 512 fabric, and 256 spine switches. There are 8192 ToR-fabric links and 8192 fabric-spine links. The link length distribution depends on the physical layout of the data center network. For links under 300m, 10G-SR is used with multi-mode fiber. We draw link length from 0-500m uniform random distribution. Results with link lengths drawn from Microsoft's network were similar.

For links above 300m, 10GBASE-LR is used with single-mode fiber. Table 4.1 provides



(a) 10G-SR

(b) 40G-SR4

Figure 4.11: BER distribution for different link lengths. The top of the orange bar is 99.9 percentile BER, and the bottom of the green bar is 0 percentile BER. The two colors meet at 99% BER. The green portion is not visible when BER is below 10^{-12}.

a breakdown of the cost of the data center network. As the table shows, transceivers represent 48% of the total cost of the network. If the same network were using a 300m 10GBASE-SR transceiver stretched to 500m (for example), all the 300–500m links could use this transceiver instead of the more expensive 10GBASE-LR one.

Figure 4.12 shows the cost reduction versus NRB for 0-500m uniform and 0-1000m uniform link length distributions. For these plots, we compute the stretch level from NRB as outlined in Chapter 4.2 and then the cost savings from the stretch level. NRB in the data centers networks we measured is 99.9%. We see that cost savings are significant when NRB drops to 99%, and the incremental gain is small beyond 95%. Thus, in later evaluations, we only consider two degrees of stretch, low stretch for NRB=99% and high stretch for NRB=95%, which correspond to stretch levels in Table 4.2. We get the stretch levels from the optical simulator by stretching until the link has a probablity to be a gray link to violate the NRB requirement given the attenuation distribution measured in Chapter 4.1.

Next, we study how link length distribution affects the amount of cost savings. Fig-

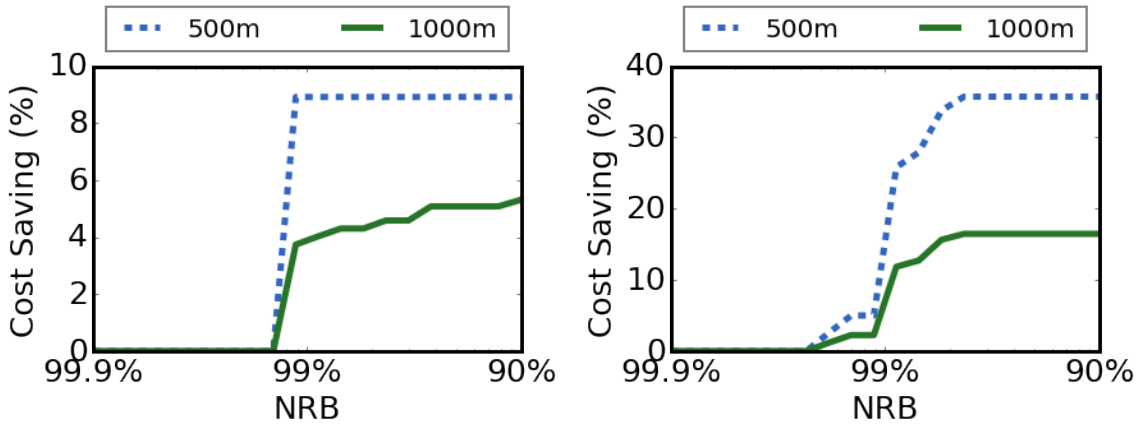
Device	Unit Cost	Count	Total Cost
300m 10G-SR Transceiver	$82.3	19660	$1.6M
10km 10G-LR Transceiver	$187.2	13108	$2.5M
Multi-mode Fiber (10 Gbps)	$0.44/m	1475 km	$0.6M
Single-mode Fiber (10 Gbps)	$0.21/m	2621 km	$0.6M
Switch (10 Gbps)	$90/port	32768 ports	$2.9M
Power(150W/Switch, 3 years)	$0.07/KWh	5 GWh	$0.4M

Table 4.1: Total data center network cost breakdown for 512 ToR, 512 fabric and 256 spine switches Clos network with 10 Gbps technology. Link length is drawn from uniform distribution between 0-500m. With 10 Gbps technology, transceivers account for 48% of the total data center network cost.

ure 4.13a shows the resulting cost savings for a 10 Gbps network. When no link is longer than 300m, stretch does not result in any savings because we never use stretched technologies. The cost saving peaks at 480m for low stretch and 580m for high stretch, the lengths at which most fractions of links can use shorter reach technologies beyond their design reach. When link length distribution concentrates on long links, the amount of cost savings decreases because most of the links need to use long reach technologies anyway, and the cost on long reach technologies dominates the total cost of the network.[7] Overall, stretching short reach technology reduces the total data center network cost up to 10% depending on the link length distribution. The savings are lower for low stretch, because fewer 10GBASE-LR transceivers can be changed to 10GBASE-SR transceivers.

Figure 4.13b shows the cost results for 40 Gbps networks. Except for transceiver costs, which are listed in Figure 4.6, we assume the cost and energy consumption of 40 Gbps

[7] If the network uses non-standard, intermediate reach (e.g., 500m) transceivers, we could have stretched them to cover longer distances (e.g., 500m to 1km) as well.



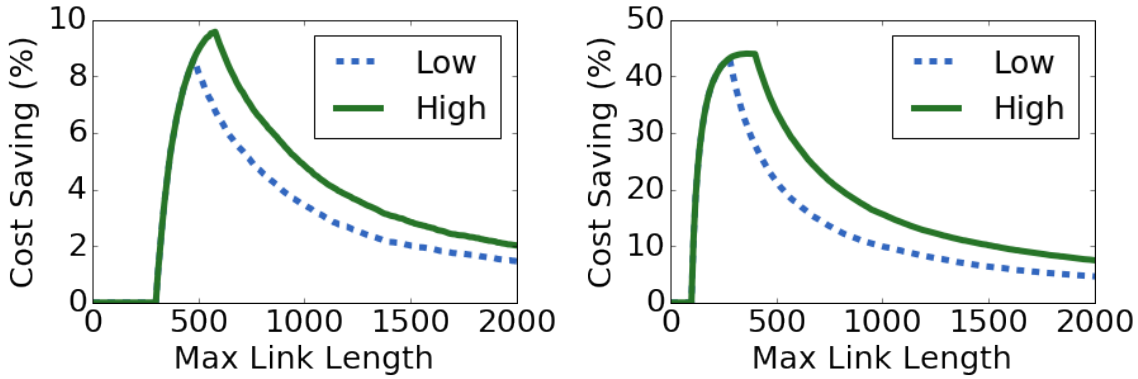
(a) 10 Gbps (b) 40 Gbps

Figure 4.12: Total data center network cost reduction for transceiver stretch on a 3-stage fat tree network (10 Gbps, 40 Gbps) with 512 ToRs assuming 0-500m and 0-1000m uniform link length distribution. The cost reduction depends on the amount of stretch and the tolerance for gray paths.

components is $4\times$ higher than their 10 Gbps counterparts—40 Gbps components often bundle four 10 Gbps components. While we assume multi-mode fiber is $1.32 per meter, we keep the cost of single mode fiber the same because 40 Gbps single-mode transceivers use wavelength division multiplexing. Transceivers account for 72% of the total data center network cost for 0-500m uniform link length distribution. The overall trend of cost savings is much higher thant that in the 10 Gbps network. The total cost savings on 40 Gbps networks can be up to 44% depending on the link length distribution. The cost reductions are higher for faster networks because $i)$ there is a higher cost difference between short and long reach technologies, $ii)$ a larger fraction of the links can make use of shorter reach technology because the short reach 40 Gbps technology can reach up to only 100m, and $iii)$ faster transceivers account for a larger fraction of total data center network cost based on our cost assumptions.

Technology	No Stretch	Low (NRB=99%)	High (NRB=95%)
10GBASE-SR (OM3)	300m	480m	580m
40GBASE-SR4 (OM3)	100m	280m	400m

Table 4.2: Optical technologies' maximum reach under different degrees of stretch.



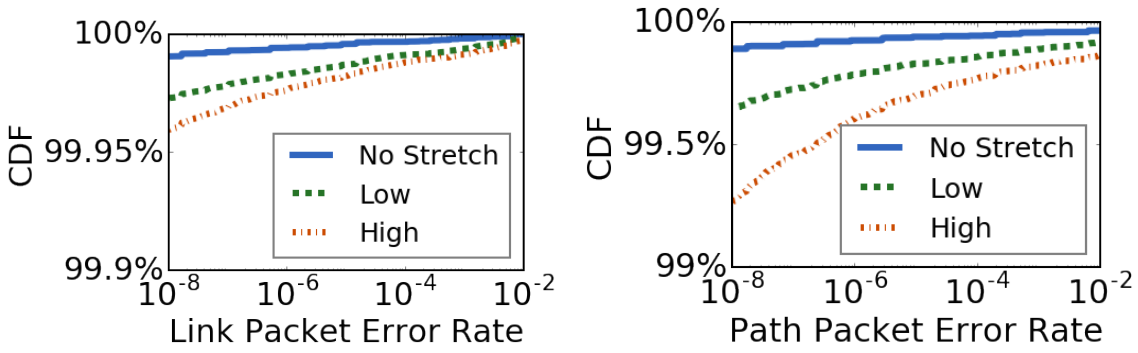
(a) 10 Gbps

(b) 40 Gbps

Figure 4.13: Total data center network cost reduction for transceiver stretch on a 3-stage fat tree network (10 Gbps, 40 Gbps) with 512 ToRs assuming uniform link length distribution. Maximum cost savings for 10/40 Gbps is achieved when max link length is 580/400m.

4.6.2 Characterizing a Stretched Network

NRB ensures by design that the network has a certain minimum fraction of good paths. We now provide a more detailed characterization of a stretched network in terms of the distribution of its link and path qualities. These distributions depend on the exact link length distribution in the network. So that our results can be reproduced, we use 0-500m uniform link length distribution on a 512-ToR Clos network (as in Chapter 4.6.1). We study 40 Gbps below; 10 Gbps networks behave similarly.



(a) LPER

(b) PPER

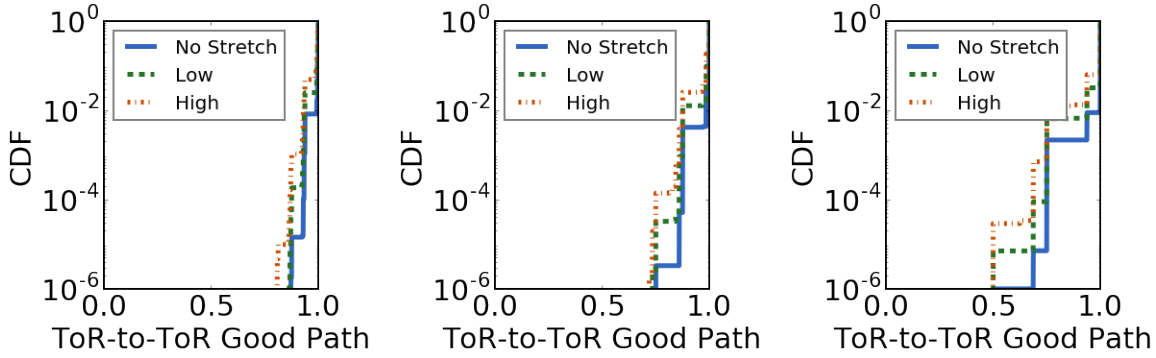
Figure 4.14: Link and path packet error rates when oversubscription ratio is 4: (a) cumulative distribution function of LPER. (b) cumulative distribution function of PPER.

Link qualities. Figure 4.14a shows the cumulative distribution function of LPERs with and without stretch. We see that even with high stretch, only 0.05% of the links have LPER worse than 10^{-8} (which corresponds to 10^{-12} BER). Only 0.01% of the links have LPER higher than 10%, our guideline for switching off links. As reference, we note that at any given time, roughly 0.08% of the links are down for other reasons in our data centers.

Path qualities. To study path characteristics in a stretched network, we simulate three different oversubscription levels. When the oversubscription level is 1 (4, 16), we have 512 (256, 128) fabric switches and 256 (128, 64) spine switches. Each switch has 32 ports.

We assume the links in the topology have LPERs as per the distribution above. We showed earlier that low RxPower (and thus high LPER) levels are not correlated with switches and appear independent across links. For each oversubscription level, we generate multiple topologies with different randomized mappings from the LPER distribution and present results aggregated across them.

Figure 4.14b shows the cumulative distribution function of PPER for a 512-ToR topology with an oversubscription of four. The results are similar for other oversubscription



(a) Oversub = 1 (b) Oversub = 4 (c) Oversub = 16

Figure 4.15: CDFs of the fraction of good paths for different ToR-ToR pairs across 50 different topologies. The Y axis is log scale to capture the tail behavior.

levels; 99% of paths in the topology have PPER below 10^{-8}.

Worst-case experience of loss-sensitive applications. An overall high fraction of good paths is not sufficient to ensure good performance for loss-sensitive applications. For every pair of ToRs that exchange traffic for such applications, there must be enough good paths. Figure 4.15 shows the CDF of the ratio of good paths to total paths across ToR pairs. A good path means PPER less than 4.8×10^{-8}, equivalent to 10^{-12} BER on each link of a path in a 3-stage fat tree.

We see that when the oversubscription is equal to one, the tail 0.01% of the ToR-to-ToR pairs still has 83% of the good paths remaining. This fraction decreases as oversubscription increases because ToR switches now have fewer uplinks to fabric switches. If one such uplink has high LPER, it impacts a higher fraction of paths from this ToR to other ToRs. However, even when the oversubscription is 16, the tail 0.01% of the ToR-to-ToR paths still has 70% of good paths left. On the flip side, these data also demonstrate that simply turning gray links off can halve the capacity between some ToR pairs.

4.6.3 *Application Performance with RAIL*

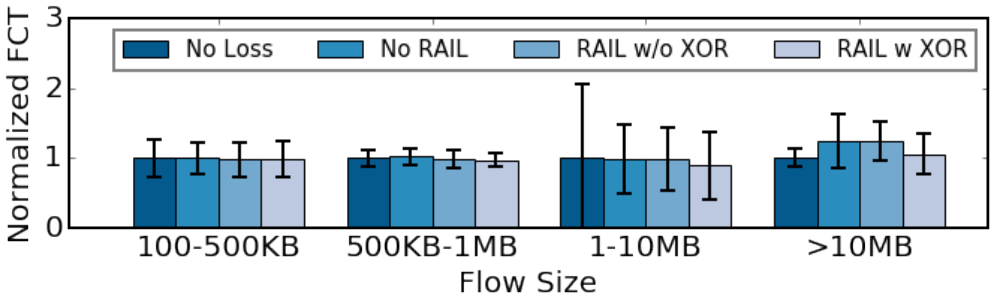
We study the effectiveness of RAIL in preserving application performance using experiments on a small but realistic testbed. Our testbed emulates a 3-stage fat tree network with four ToRs, four fabric switches, and two spine switches. The ToR and fabric switches are spread across two pods. Each port in the topology is a 10Gbps SPF+ port, and each link has two Finisar FTLX8571D3BCL [10] multi-mode transceivers connected via OM3 fiber [5]. The switches implement ECMP routing. One host is attached to each ToR switch, runs Ubuntu 14.04 with Linux kernel version 3.19, and uses TCP CUBIC [78].

We emulate a gray, high-BER link using an optical attenuator and change the location of the attenuator in different experiments. We use the virtual topology configuration guideline above, but because it is difficult to finely control the BER using an attenuator, we obtain configurations in which the first- and second-class topologies are identical and contain highly-reliable paths. The third-class topology contains the high-BER link(s). We use iperf to send TCP flows between arbitrary end hosts, with a flow size and inter-arrival time distribution from prior work [31]. TCP flows smaller than 1MB bind to the second-class topology; longer flows bind to the third-class topology.[8]

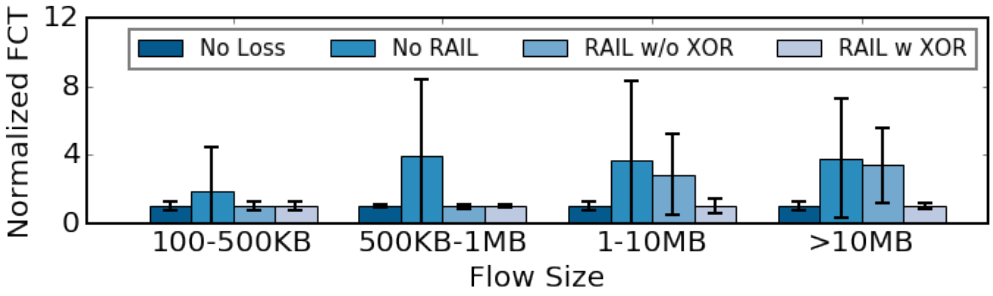
Figure 4.16 shows flow completion times (FCT) binned by flow size and normalized to the case of a completely lossless network ("No Loss"). When the network is lossy, without RAIL ("No RAIL"), we see that FCTs are higher for all flow sizes, especially when the loss rate is high. RAIL without error coding ("RAIL w/o XOR") is able to protect only high-priority (short) flows because it routes them over the reliable, second-class topology. With error coding, RAIL ("RAIL w XOR") is able to protect low-priority flows as well. The performance experienced by all flows matches that of the lossless network.

Error correction overhead. Finally, we study the bandwidth overhead of RAIL's error correction. Recall that we set the target post-recovery loss rate to 0.1%. Thus, error cor-

[8]This mapping between flow size and topology is only for our experiments. In reality, we expect applications to bind to the desired virtual interface. RAIL does not try to guess flow sizes.



(a) LPER = 0.7%



(b) LPER = 5%

Figure 4.16: Normalized flow completion time. Error bars are standard deviations of the normalized flow completion time. Flows shorter than 1MB are high priority flows. High priority flows are not affected by packet corruption with RAIL. Low priority flows are protected by XOR coding.

rection only kicks in for paths with PER above 0.1%. For paths with PER between 0.1% and 3%, we use standard XOR code with n computed based on error rate. For PPER from 3–10%, we simply replicate every packet three times to reduce the post-recovery loss rate to 0.1%.

The average bandwidth overhead of our coding scheme is below 0.1% for the 512-ToR topology with 0-500m and 0-1000m uniform link length distribution. This is because almost all paths have loss rate below 0.1% and thus no FEC is needed. This low overhead is the reason we use a simple coding method in RAIL instead of more efficient methods based on retransmissions or bit-level FEC (forward error correction).

4.7 Discussion

Masking bit-level corruptions. Today, when a bit in a packet is corrupted, the entire packet is dropped at the point of detection via a frame sequence check (FCS) at the Ethernet layer. This means the network capacity of transmitting the remaining bits in the corrupted packet is wasted even when those bits are not corrupted. Currently, RAIL implements packet-level forward error correction and thus wastes network capacity in this way. A more principled direction is to implement bit-level forward error correction on end-to-end network paths.

4.8 Related Work

Our work draws on several themes of previous work.

Measuring optical links. Many researchers have studied optical wide area networks for properties such as dispersion [62, 88, 44, 147], temperature variations [86, 98], and packet loss and inter-arrival times [67, 106]. Ghobadi et al. study optical signal quality in the wide area network setting and, like us, find that optical layers are overprovisioned [71]. In contrast, however, our focus is on data center networks, where the environment and technology are different. We believe we are the first to study this optical layer.

Reducing cost of optics in data center networks. We are inspired by other efforts in the industry to lower the cost of optics in data center networks. Facebook [48] is pushing for a new standard for cost-efficient 100 Gbps transceivers. Their initial observation is similar to ours: the data center network is a milder operating environment than traditional telecom networks. Corning [49] also observed, using stochastic attenuation models, that data center network link qualities can be disparate and it is possible to extend the reach on some fraction of links. We complement these efforts with a detailed characterization of optical links in operational data center networks, proposing a way to reduce cost without hardware changes and developing a system to preserve application performance if some links turn gray due to reduced over-engineering.

Virtual topologies. The concept of virtual topologies over the same physical infrastructure has been leveraged in other contexts, such as detour routing [136], virtual local area networks, overlay networks, simplifying the specification of network policies in software-defined networking (SDN) [94, 132, 113], or "slicing" the network to isolate users [38, 138, 114]. We use this concept to build topologies with different reliability guarantees.

Reliable systems atop unreliable components. There is a long-standing tradition of building reliable systems using (cheaper) unreliable components and masking unreliability from applications using intelligent software techniques. A classic example is to build reliable storage systems using disks that are individually less reliable [121, 70]. Our work follows this tradition, though the set of techniques it uses are specific to its domain.

4.9 Summary

We present the first study of optical links in data center networks. We show that optical links in data center networks are over-engineered. This over-engineering is not only expensive but also unnecessary because of application and path diversity in data center

networks. Many applications can tolerate small amounts of loss, and loss-sensitive applications can be supported as long as some (not all) paths between ToR pairs are reliable. We find that reducing optical over-engineering simply by using transceivers beyond their specified length can reduce network cost by up to 10% for 10 Gbps networks and 44% for 40 Gbps networks. Moreover, when coupled with the traffic routing and error correction mechanisms of RAIL, there is negligible loss in application performance.

Chapter 5

SLIM: OS KERNEL SUPPORT FOR A LOW-OVERHEAD CONTAINER OVERLAY NETWORK.

The past two chapters have focused on the efficiency and reliability of the physical network infrasctructure. In this chapter, we move to look at how to most efficiently construct a virtual network in a data center for containers.

Containers [52] have quickly become the de facto method to manage and deploy large-scale distributed applications, including in-memory key-value stores [109], web servers [115], databases [128], and data processing frameworks [33, 95]. Containers are attractive because they are lightweight and portable. A single physical machine can easily host more than ten times as many containers as standard virtual machines [105], resulting in substantial cost savings.

Container overlay networks—a key component in providing portability for distributed containerized applications—allow a set of containers to communicate using their own independent IP addresses and port numbers, no matter where they are assigned or which other containers reside on the same physical machines. The overlay network removes the burden of coordinating ports and IP addresses between application developers, and vastly simplifies migrating legacy enterprise applications to the cloud [64]. Today, container orchestrators, such as Docker Swarm [55], require the usage of overlay network for hosting containerized applications.

However, container overlay networks impose significant overhead. Our benchmarks show that, compared to a host network connection, the throughput of an overlay network connection is 23-48% less, the packet-level latency is 34-85% higher, and the CPU utilization is 93% more. (See Chapter 5.1.1.) Known optimization techniques (e.g., packet

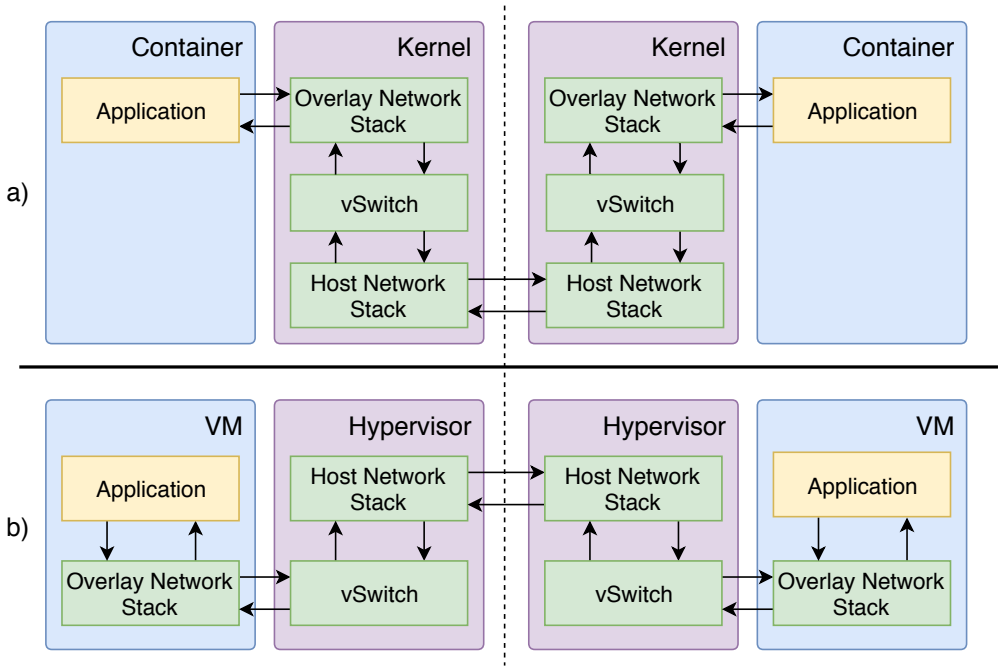


Figure 5.1: Packet flow in: (a) today's container overlay networks, (b) overlay networks for virtual machines.

steering [119] and hardware support for virtualization [89, 64]) only partly address these issues.

The key problem is that today's container overlay networks depend on multiple packet transformations within the OS for network virtualization (Figure 5.1a). This means each packet has to traverse network stack **twice** and also a virtual switch on both the sender and the receiver side. Take sending a packet as an example. A packet sent by a container application first traverses the overlay network stack on the virtual network interface. The packet then traverses a virtual switch for packet transformation (e.g, adding host network headers). Finally, the packet traverses the host network stack, and is sent out on the host network interface. On the receiving server, these layers are repeated in the opposite order.

This design largely resembles the overlay network for virtual machines (Figure 5.1b). Because a virtual machine has its own network stack, the hypervisor has to send/receive

raw overlay packets without the context of network connections. However, for containers, the OS kernel has full knowledge of each network connection.

In this chapter, we ask whether we can design and implement a container overlay network, where packets go through the OS kernel's network stack **only once**. This requires us to remove packet transformation from the overlay network's data-plane. Instead, we implement network virtualization by manipulating connection-level metadata at connection setup time, saving CPU cycles and reducing packet latency.

Realizing such a container overlay network is challenging because: (1) network virtualization has to be compatible with today's unmodified containerized applications; (2) we need to support the same networking policies currently enforced by today's container overlay network on the data-plane; and (3) we need to enforce the same security model as in today's container overlay networks.

We design and implement Slim, a low-overhead container overlay network that provides network virtualization by manipulating connection-level metadata. Our evaluations show that Slim improves the throughput of an in-memory key-value store, Memcached [109], by 71% and reduces its latency by 42%, compared to a well-tuned container overlay network based on packet transformation. Slim reduces the CPU utilization of Memcached by 56%. Slim also reduces the CPU utilization of a web server, Nginx [115], by 22%-24%; a database server, PostgreSQL [128], by 22%; and a stream processing framework, Apache Kafka [33, 95], by 10%. However, Slim adds complexity to connection setup, resulting in 106% longer connection setup time. Other limitations of Slim: Slim supports quiescent container migration, but not container live migration; connection-based network policies but not packet-based network policies; and TCP, defaulting to standard processing for UDP sockets. (See Chapter 5.6.)

The chapter makes the following contributions:

- Benchmarking of existing container overlay network with several data-plane optimizations. We identify per-packet processing costs (e.g., packet transformation, extra traversal of network stack) as the main bottleneck in today's container overlay

network. (See Chapter 5.1.1, Chapter 5.1.2.)

- Design and implementation of Slim, a solution that manipulates connection-level metadata to achieve network virtualization. Slim is compatible with today's containerized applications and standard OS kernels. Slim supports various network policies and guarantees the same security model as that of today's container overlay network. (See Chapter 5.3.)
- Demonstration of the benefits of Slim for a wide range of popular containerized applications, including an in-memory key-value store, a web server, a database server, and a stream processing framework. (See Chapter 5.5.)

Fundamentally, Slim integrates efficient virtualization into the OS kernel's networking stack. A modern OS kernel already has efficient native support to virtualize file systems (using *mount* namespace) and other OS components (e.g., process id, user group). The network stack is the remaining performance gap for efficient container virtualization. Slim bridges this gap.

5.1 Overheads in Container Overlay Networks

We quantify the overhead of today's container overlay network solutions in terms of throughput, latency, and CPU utilization. We show that the overhead is significant even after applying known overhead reduction techniques (e.g., packet steering [119]).

5.1.1 Overhead in Container Overlay Networks

The overhead of today's container overlay networks comes from per-packet processing (e.g., packet transformation, extra traversal of the network stack) inside the OS kernel.

Journey of an Overlay Network Packet

In our example (Figure 2.3), assume that a TCP connection has previously been established between 10.0.0.1 and 10.0.0.2. Now, the container sends a packet to 10.0.0.2 through

this connection. The OS kernel's overlay network stack first writes the virtual destination IP address 10.0.0.2 and source IP address 10.0.0.1 on the packet header. The OS kernel also writes the Ethernet header of the packet to make the packet a proper Ethernet frame. The Ethernet frame traverses a virtual Ethernet link to the virtual switch's input buffer.

The virtual switch recognizes the IP address 10.0.0.2 inside the Ethernet frame as that of a container on a remote host. It adds a physical IP header to the Ethernet frame using host source and destination addresses from its routing table. The packet now has both a physical and a virtual header. On the host network, the packet is simply a UDP packet (assuming the tunneling protocol is VXLAN) and its UDP payload is the Ethernet frame. The OS kernel then delivers the encapsulated packet to the wire using the host network stack.

The receiving pipeline is the same except that the virtual switch removes the host network header instead of adding one. The receiving side receives the exact same Ethernet frame from the sending side's virtual network interface.

We can thus see why the overlay network is expensive: delivering a packet on the overlay network requires one extra traversal of the network stack and also packet encapsulation and decapsulation.

Quantifying Overhead

We give a detailed breakdown of the overhead in one popular container overlay network implementation, Weave [144]. Our testbed consists of two machines with Intel Xeon E5-2680 (12 physical cores, 2.5 GHz). The machines use hyper-threading and therefore each has 24 virtual cores. Each machine runs Linux version 4.4 and has a 40 Gbps Intel XL710 NIC. The two machines are directly connected via a 40 Gbps link. The physical NIC is configured to use Receive Side Scaling (RSS). In all of our experiments, we do not change the configuration of the physical NICs.

We create an overlay network with Weave's fast data-plane mode (similar to the archi-

Setup	Throughput (Gbps)	RTT (µs)
Intra, Host	48.4 ± 0.7	5.9 ± 0.2
Intra, Overlay	37.4 ± 0.8 (23%)	7.9 ± 0.2 (34%)
Inter, Host	26.8 ± 0.1	11.3 ± 0.2
Inter, Overlay	14.0 ± 0.4 (48%)	20.9 ± 0.3 (85%)

Table 5.1: Throughput and latency of a single TCP connection on a container overlay network, compared with that using host mode. Intra is a connection on the same physical machine; Inter is a connection between two different physical machines over a 40 Gbps link. The numbers followed by ± show the standard deviations. The numbers in parentheses show the relative slowdown compared with using host mode.

tecture in Figure 2.3). We use *iperf3* [82] to create a single TCP connection and study TCP throughput atop the container overlay network. We use *NPtcp* [118] to measure packet-level latency. For comparison, we also perform the same test using host mode container networking. In all of our experiments, we keep the CPU in maximum clock frequency (using Intel P-State driver [130]).

The overhead of the container overlay network is significant. We compare TCP flow throughput and packet-level latency under four different settings. Table 5.1 shows average TCP flow throughput with maximum ethernet frame size over a 10-second interval and the round trip latency for 32-byte TCP packets for 10 tests. For two containers on the same host, TCP throughput reduces by 23% and latency increases by 34%. For containers across physical machines, TCP throughput reduces by almost half (48%) and latency increases by 85%. Intra-host container overlay network has lower overheads because packet encapsulation is not needed.

To understand the source of the main bottleneck, we measure CPU utilization with a standard Linux kernel CPU profiling tool, *mpstat*. We specifically inspect the overlay

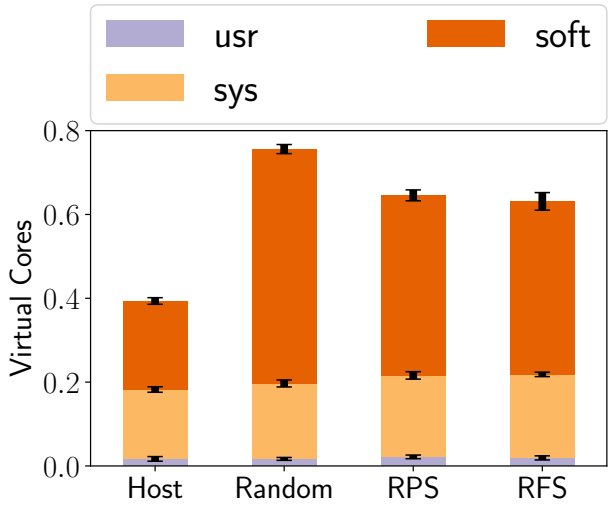


Figure 5.2: CPU utilization under different overlay network setups measured by number of virtual cores used for a single 10 Gbps TCP connection. The CPU cycles are spent: in user-level application (usr), inside kernel but excluding interrupt handling (sys), and serving software interrupts (soft). Error bars denote standard deviations.

network across two different physical machines. We set the speed of the TCP connection to 10 Gbps and then use *mpstat* to identify where CPU cycles are spent for 10 tests where each test lasts 10 seconds. Figure 5.2 shows the overall CPU utilization and the breakdown. Compared with using a direct host connection, in the default mode (Random IRQ load balancing), the overlay network increases CPU utilization (relatively) by 93%. RPS (receive packet steering) and RFS (receive flow steering) are two optimizations we have done to Weave. (See Chapter 5.1.2.)

The main CPU overhead of the overlay network comes from serving software interrupts; in the default overlay setting, it corresponds to 0.56 virtual cores. The reason why the extra CPU utilization is in the software interrupt handling category is that packet transformation and the traversal of the extra network stack is not directly associated with a system call. These tasks are offloaded to per-core dedicated *softirq* thread. For compari-

son, using the host mode, only 0.21 virtual cores are spent on serving software interrupts. This difference in CPU utilization captures the extra CPU cycles wasted on traversing the network stack one extra time and packet transformation. Note here we do not separate the CPU utilization due to the virtual switch and due to the extra network stack traversal. Our solution, Slim, removes both these two components from the container overlay network data-plane at the same time, so understanding how much CPU utilization these two components consume combined is sufficient.

In Chapter 5.1.2, we show that existing techniques (e.g., packet steering) can address some of the performance issues of a container overlay network. However, significant overhead still remains.

5.1.2 Fine-Tuning Data-plane

There are several known techniques to reduce the data-plane overhead. Packet steering creates multiple queues, each per CPU core, for a network interface and uses consistent hashing to map packets to different queues. In this way, packets in the same network connection are processed only on a single core. Different cores therefore do not have to access the same queue, removing the overhead due to multi-core synchronization (e.g., cache-line conflicts, locking). Table 5.2 shows the changes to throughput and latency on a container overlay network using packet steering.

Packet steering improves TCP throughput to within 91% of using a host TCP connection, but it does not reduce packet-level latency. We experimented with two packet steering options, Receive Packet Steering (RPS) and Receive Flow Steering (RFS), for internal virtual network interfaces in the overlay network. RPS[1] ensures that packets in the same flow always hit the same core. RFS, an enhancement of RPS, ensures that software interrupt processing occurs on the same core as the application.

Although packet steering can imporve throughput, it has a more modest impact on

[1]RSS requires hardware NIC support. RPS is a software implementation of RSS that can be used on virtual network interfaces inside the OS kernel.

Setup	Throughput (Gbps)	RTT (µs)
Random LB	14.0 ± 0.4 (48%)	20.9 ± 0.3 (85%)
RPS	24.1 ± 0.8 (10%)	20.8 ± 0.1 (84%)
RFS	24.5 ± 0.3 (9%)	21.2 ± 0.2 (88%)
Host	26.8 ± 0.1	11.3 ± 0.2

Table 5.2: TCP throughput and latency (round-trip time for 32-byte TCP packets) for different packet steering mechanisms atop a container overlay network across two physical hosts. The numbers followed by ± show the standard deviations. The numbers in parentheses show the relative slowdown compared with using the host mode.

CPU utilizaion than throughput and almost no change to latency. Packets still have to go through the same packet transformations and traverse the network stack twice. Our design, Slim, focuses directly on removing this per-packet processing overhead in container overlay networks.

5.2 Overview

Slim provides a low-overhead container overlay network in which packets in the overlay network traverse the network stack exactly once. Like other container overlay network implementations [144, 54, 65], Slim creates a virtual network with a configuration completely decoupled from the host network's. Containers have no visibility of host network interfaces, and they communicate only using virtual network interfaces that the OS kernel creates.

We require Slim to be (1) *readily deployable*, supporting unmodified application binaries; (2) *flexible*, supporting various network policies, such as access control, rate limiting, and quality of service (QoS), at both per-connection and per-container levels; and (3) *se-*

cure, the container cannot learn information about the physical hosts, create connections directly on host network, or increase its traffic priority.

Figure 5.3 shows Slim's architecture. It has three main components: (1) a user-space shim layer, SlimSocket, that is dynamically linked with application binaries; (2) a user-space router, SlimRouter, running in the host namespace; and (3) a small optional kernel module, SlimKernelModule, which augments the OS kernel with advanced Slim features (e.g., dynamically changing access control rules, enforcing security).

Slim virtualizes the network by manipulating connection-level metadata. SlimSocket exposes the POSIX socket interface to application binaries to intercept invocations of socket-related system calls. When SlimSocket detects an application is trying to set up a connection, it sends a request to SlimRouter. After SlimRouter sets up the network connection, it passes access to the connection as a file descriptor to the process inside the container. The application inside the container then uses the host namespace file descriptor to send/receive packets directly to/from the host network. Because SlimSocket has the exact same interface as the POSIX socket, and Slim dynamically links SlimSocket into the application, the application binary need not be modified.

In Slim, packets go directly to the host network, circumventing the virtual network interface and the virtual switch; hence, a separate mechanism is needed to support various flexible control-plane policies (e.g., access control) and data-plane policies (e.g., rate limiting, QoS). Control-plane policies isolate different components of containerized applications. Data-plane policies limit a container's network resource usage and allow prioritization of network traffic. In many current overlay network implementations, both types of policies are actually enforced inside the data-plane. For example, a typical network firewall inspects every packet to determine if it is blocked by an access control list.

SlimRouter stores control-plane policies and enforces them at connection setup time. This approach obviates the need to inspect every packet in the connection. Before creating a connection, SlimRouter checks whether the access control list permits the connection. When the policy changes, SlimRouter scans all existing connections and removes the file

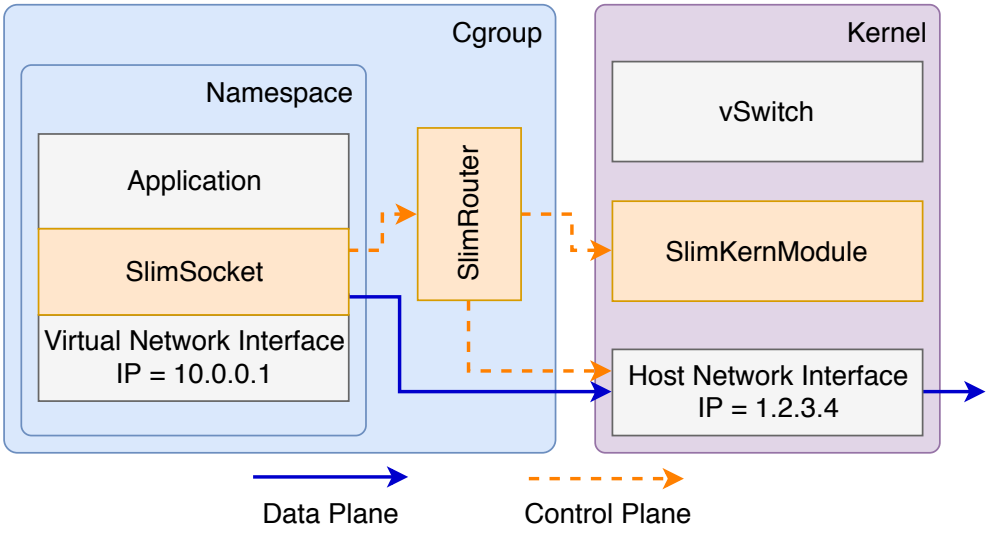


Figure 5.3: Architecture of Slim.

descriptors for any connection that violates the updated access control policy through SlimKernelModule. Slim leverages existing kernel functionalities to enforce data-plane policies.

Sending a host namespace file descriptor directly to a malicious container raises security concerns. For example, if a malicious container circumvents SlimSocket and invokes the *getpeername* call directly on the host namespace file descriptor, it would be able to learn the IP addresses of the host machines. A container could also call *connect* with a host network IP address to create a connection directly on the host network, circumventing the overlay network. Finally, a container could call *setsockopt* to increase its traffic priority.

To enforce the same security model as in today's container overlay network, Slim offers a secure mode. When secure mode is on, Slim leverages a kernel module, SlimKernelModule, to restrict the power of host namespace file descriptors inside containers. SlimKernelModule implements a lightweight capability system for file descriptors. SlimKernelModule has three roles: (1) track file descriptors as they propagate inside the container, (2) revoke file descriptors upon request from SlimRouter, and (3) prohibit a list

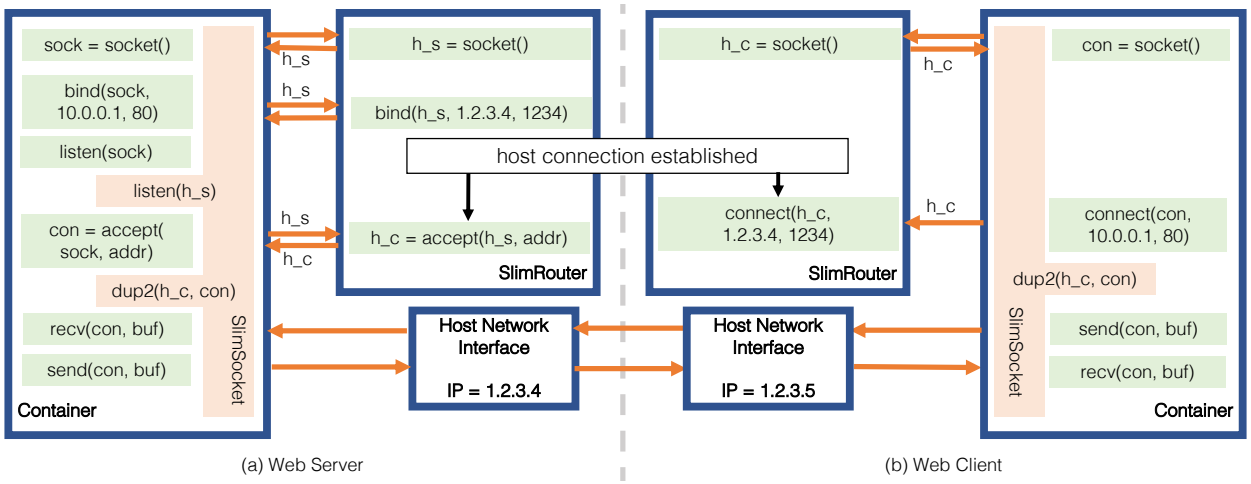


Figure 5.4: TCP connection setup between a web client and a web server atop Slim.

of unsafe system calls using these file descriptors (e.g., *getpeername*, *connect*, *setsockopt*). SlimSocket emulates these system calls for non-malicious applications.

5.3 Design

We first describe how to implement network virtualization without needing packet transformations in the data-plane while maintaining compatibility with current containerized applications. We then describe how to support flexible network polices and enforce security for malicious containers.

Slim does not change how virtual to physical IP mappings are stored. They can still be either stored in external storage or obtained through gossiping. As with today's container overlay network, Slim relies on a consistent and current view of containers' locations in the host network.

5.3.1 Connection-based Network Virtualization

Slim provides a connection-based network virtualization for containers. When a container is initiated on the host, Slim dispatches an instance of SlimRouter in the host names-

pace. Slim links a user-level shim layer, SlimSocket, to the container. When the process inside the container creates a connection, instead of making standard socket calls, SlimSocket sends a request to SlimRouter with the destination IP address and port number. SlimRouter creates a connection on behalf of the container and returns a host namespace file descriptor back to the container. We first present an example that shows how Slim supports traditional blocking I/O. We then describe how to additionally make Slim support non-blocking I/O.

Support for blocking I/O. Figure 5.4 shows how a TCP connection is created between a web client and a web server on Slim. Consider the web server side. The container first creates a socket object with the *socket* function call. This call is intercepted by SlimSocket and forwarded to SlimRouter, which creates a socket object in the host network. When the container calls *bind* on the socket with virtual network interface IP address 10.0.0.1 and port 80, SlimRouter also calls *bind* on the host network interface IP address 1.2.3.5 and with some unused port 1234. The port translation is needed because a host can run multiple web servers binding on port 80, but the host network interface only has a single port 80. SlimRouter updates the port mapping. The web server then uses *accept* to wait for an incoming TCP connection. This function call is also forwarded to SlimRouter, which waits on the host socket.

We move next to the web client side. The client performs similar steps to create the socket object. When the client side connects the overlay socket to the server side at IP address 10.0.0.1 port 80, SlimRouter looks up the virtual IP address 10.0.0.1 and finds its corresponding host IP address 1.2.3.5. SlimRouter then contacts the SlimRouter for the destination container on 1.2.3.5 to locate the corresponding host port, 1234. SlimRouter sets up a direct connection to port 1234 on 1.2.3.5. After the TCP handshake is complete, *accept*/*connect* returns a file descriptor in which socket send/recv is enabled. SlimRouter passes the file descriptor back to the container, and SlimSocket replaces the overlay connection file descriptor with the host namespace file descriptor using system call *dup2*. From this point on, the application directly uses the host namespace file descriptor to

send or receive packets.

To ensure compatibility with current containerized applications, SlimSocket exposes the same POSIX socket interface. Besides forwarding most socket-related system calls (e.g., *socket*, *bind*, *accept*, *connect*) to SlimRouter, SlimSocket also carefully maintains the expected POSIX socket semantics. For example, when a containerized application calls *getpeername* to get an IP address on the other side of the connection, SlimSocket returns the overlay IP address rather than the host IP address, even when the file descriptor for the overlay connection has already been replaced with the host namespace file descriptor.

Support for non-blocking I/O. Most of today's applications [109, 115] use a non-blocking I/O API (e.g., *select*, *epoll*) to achieve high I/O performance. Slim must also intercept these calls because they interact with the socket interface. For example, *epoll* creates a meta file descriptor that denotes a set of file descriptors. An application uses *epoll_wait* to wait any event in the set, eliminating the need to create a separate thread to wait on an event in each file descriptor. On connection setup, we must change the corresponding file descriptor inside the *epoll*'s file descriptor set. SlimSocket keeps track of the mapping between the *epoll* file descriptor and epoll's set of file descriptors by intercepting *epoll_ctl*. For an *accept* or *connect* on a file descriptor that is inside an *epoll* file descriptor set, SlimSocket removes the original overlay network file descriptor from the *epoll* file descriptor set and adds host namespace file descriptor into the set.

Service discovery. Our example in Figure 5.4 assumes that the SlimRouter on the client side knows the server side has bound to physical IP 1.2.3.4 and port 1234. To automatically discover the server's physical IP address and port, we could store a mapping from virtual IP/port to physical IP/port on every node in the virtual network. Unfortunately, this mapping has to change whenever a new connection is listened.

Instead, Slim uses a distributed mechanism for service discovery. Slim keeps a standard container overlay network running in the background. When the client calls *connect*, it actually creates an overlay network connection on the standard container overlay network. When the server receives an incoming connection on the standard overlay net-

work, SlimSocket queries SlimRouter for the physical IP address and port and sends them to the client side inside the overlay connection. In secure mode (Chapter 5.3.3), the result queried from SlimRouter is encrypted. SlimSocket on the client side sends the physical IP address and port (encrypted if in secure mode) to its SlimRouter and the SlimRouter establishes the host connection. This means connection setup time is longer in Slim than that on container overlay networks based on packet transformation. (See Chapter 5.5.1.)

5.3.2 Supporting Flexible Network Policies

This section describes Slim's support for both control- and data-plane policies.

Control-plane policies. Slim supports standard access control over overlay packet header fields, such as the source/destination IP addresses and ports. Access control can also filter specific types of traffic (e.g., SSH, FTP) or traffic from specific IP prefixes.

In the normal case where policies are static, Slim enforces access control at connection creation. SlimRouter maintains a copy of current access control policies from the container orchestrator or network operator. When a connection is created by *accept* or *connect*, SlimRouter checks whether the created connection violates any existing access control policy. If so, SlimRouter rejects the connection by returning -1 to *connect* or by ignoring the connection in *accept*.

Access control policies can change dynamically, and any connection in violation of the updated access control policy must be aborted. SlimRouter keeps per-connection state, including source and destination IP addresses, ports, and the corresponding host namespace file descriptors. When access control policies change, SlimRouter iterates through all current connections to find connections that are forbidden in the updated policies. SlimRouter aborts those connections by removing the corresponding file descriptors from the container. Removing a file descriptor from a running process is not an existing feature in commodity operating systems such as Linux. We build this functionality in SlimKernelModule. (See Chapter 5.3.3 for more details.)

Data-plane policies. Slim supports two types of data-plane policies: rate limiting and quality of service (QoS). Rate limiting limits the amount of resources that a container can use. QoS ensures that the performance of certain applications is favored over other applications.

Slim reuses an OS kernel's existing features to support data-plane policies. A modern OS kernel has support for rate limiting and QoS for a single connection or a set of connections. Slim simply sets up the correct identifier to let the OS kernel recognize the container that generates the traffic.

In Slim, rate limits are enforced both at the per-connection and per-container level. Per-connection rate limits are set in a similar way as in today's overlay network using Linux's traffic control program, *tc*. For per-container rate limits, Slim first configures the *net_cls* cgroups to include the SlimRouter process. The *net_cls* cgroup tags traffic from the container or the corresponding SlimRouter with a unique identifier. SlimRouter then sets the rate limit for traffic with this identifier using *tc* on the host network interface. In this way, the network usage by SlimRouter is also restricted by the rate limit. Correct accounting of network usage is the fundamental reason why each container requires a separate SlimRouter.

Quality of service (QoS) also uses *tc*. SlimRouter uses socket options to set up the type of service (ToS) field (via *setsockopt*). In this way, switches/routers on the physical network are notified of the priority of the container's traffic.

Compatibility with existing IT tools. In general, IT tools[2] need to be modified to interact with SlimRouter in order to function with Slim. IT tools usually use some user-kernel interface (e.g., iptables) to inject firewall and rate limits rules. When working with Slim, they should instead inject these rules to SlimRouter. Because Slim is fundamentally a connection-based virtualization approach, a limitation of our approach is that it cannot support packet-based network policy (e.g., drop an overlay packet if the hash of the

[2]We only consider IT tools that run on the host to manage containers but not those run inside containers. IT tools usually require root privilege to the kernel (e.g., iptables) and are thus disabled inside containers.

packet matches a signature). (See Chapter 5.6.) If packet-based policies are needed, the standard Linux overlay should be used instead.

If static connection-based access control is the only network policy needed, then existing IT tools need not be modified. If an IT tool blocks a connection on a standard container overlay network, it also blocks the metadata for service discovery for that connection on Slim, thus it blocks the host connection from being created on Slim.

5.3.3 Addressing Security Concerns

Slim includes an optional kernel module, SlimKernelModule, to ensure that Slim maintains the same security model as today's container overlay networks. The issue concerns potentially malicious containers that want to circumvent SlimSocket. Slim exposes host namespace file descriptors to containers and therefore needs an extra mechanism inside the OS kernel to track and manage access.

SlimKernelModule implements a lightweight and general capability system based on file descriptors. SlimKernelModule tracks tagged file descriptors in a similar way as taint-tracking tools [60] and filters unsafe system calls on these file descriptors. We envision this kernel module could also be used by other systems to track and control file descriptors. For example, a file server might want to revoke access from a suspicious process if it triggers an alert. Slim cannot use existing kernel features like seccomp [137] because seccomp cannot track tagged file descriptors.

SlimKernelModule monitors how host namespace file descriptors propagate inside containers. It lets SlimRouter or other privileged processes tag a file descriptor. It then interposes on system calls that may copy or remove tagged file descriptors, such as *dup*, *fork* and *close*—to track their propagation. If the container passes the file descriptor to other processes inside the container, the tag is also copied.

Tagged file descriptors have limited powers within a container. SlimKernelModule disallows invocation of certain unsafe system calls using these file descriptors. For exam-

ple, in the case of Slim, a tagged file descriptor cannot be used with the following system calls: *connect*, *bind*, *getsockname*, *getpeername*, *setsockopt*, etc. This prevents containers from learning their host IP addresses or increasing their traffic priority. It also prevents containers from directly creating a host network connection. For a non-malicious container, SlimSocket and SlimRouter emulate the functionalities of these forbidden system calls.

SlimKernelModule revokes tagged file descriptors upon request. To do so, it needs a process identifier (pid) and a file descriptor index. SlimRouter uses this functionality to implement dynamic access control. When the access control list changes for existing connections, SlimRouter removes the file descriptors through SlimKernelModule. SlimKernelModule revokes all the copies of the file descriptors.

Secure versus Non-secure mode. Whether to use Slim in secure mode (with SlimKernelModule) or not depends on the use case. When containers and the physical infrastructure are under the same entity's control, such as for a cloud provider's own use [100], non-secure mode is sufficient. Non-secure mode is easier to deploy because it does not need kernel modification. When containers are potentially malicious to the physical infrastructure or containers of other entities, secure mode is required. Secure mode has slightly (∼25%) longer connection setup time, making the overall connection setup time 106% longer than that of a traditional container overlay network. (See Chapter 5.5.1.)

5.4 Implementation

Our implementation of Slim is based on Linux and Docker. Our prototype includes all features described in Chapter 5.3. SlimSocket, SlimRouter, and SlimKernelModule are implemented in 1184 lines of C, 1196 lines of C++ (excluding standard libraries), and 1438 lines of C, respectively.

Our prototype relies on a standard overlay network, Weave [144], for service discovery and packets that require data-plane handling (e.g., ICMP, UDP).

SlimSocket uses LD_PRELOAD to dynamically link to the application binary. Communication between SlimSocket and SlimRouter is via a Unix Domain Socket. In non-

secure mode, file descriptors are passed between SlimRouter and SlimSocket by *sendmsg*. For secure mode, file descriptors are passed with SlimKernelModule's cross-process file descriptor duplication method.

SlimRouter allows an network operator to express the access control as a list of entries based on source/destination IP address prefixes and ports in a JSON file. SlimRouter has a command-line interface for network operators to issue changes in the access control list via reloading the JSON file. Slim rejects any connection matched in the list. SlimRouter uses *htb* qdisc to implement rate limits and *prio* qdisc for QoS with *tc*.

SlimRouter and SlimKernelModule communicate via a dummy file in *procfs* [129] created by SlimKernelModule. SlimKernelModule treats *write*s to this file as requests. Accessing the dummy file requires host root privilege.

SlimKernelModule interposes on system calls by replacing function pointers in the system call table. SlimKernelModule stores tagged file descriptors in a hash table and a list of unsafe system calls. SlimKernelModule rejects unsafe system calls on tagged file descriptors.

SlimKernelModule also interposes on system calls such as *dup*, *dup2* and *close* to ensure that file descriptor tags are appropriately propagated. For process fork (e.g., *fork*, *vfork*, *clone* in Linux kernel), SlimKernelModule uses the *sched_process_fork* as a callback function. Slim does not change the behavior of process forking. A forked process still has SlimSocket dynamically linked.

5.5 Evaluation

We first microbenchmark Slim's performance and CPU utilization in both secure and non-secure mode and then with four popular containerized applications: an in-memory key-value store, Memcached [109]; a web server, Nginx [115]; a database, PostgreSQL [128]; and a stream processing framework, Apache Kafka [33, 95]. Finally, we show performance results for container migration. Our testbed setup is the same as that for our measurement study (Chapter 5.1.1). In all the experiments, we compare Slim with Weave [144] with its

fast data-plane enabled and with RFS enabled by default. We use Docker [52] to create containers.

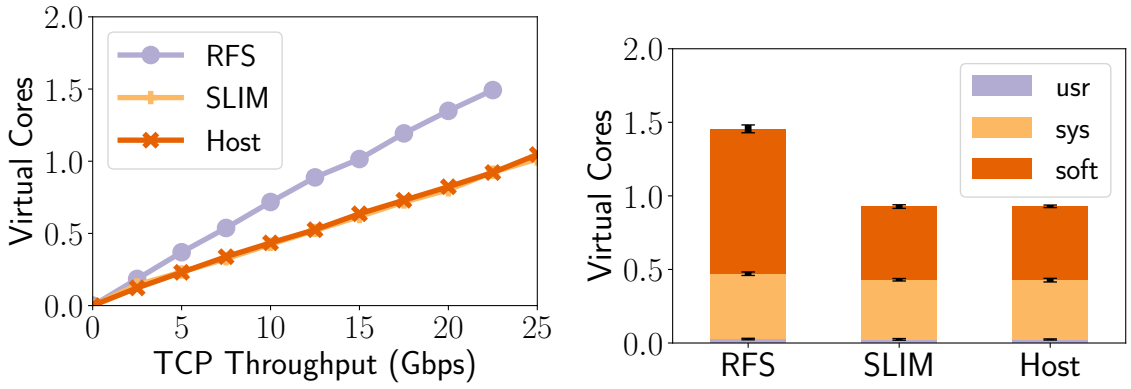
5.5.1 Microbenchmarks

Similar to the performance tests in Chapter 5.1.1, we use *iperf3* [82] and *NPtcp* [118] to measure performance of a TCP flow. We use *mpstat* to measure CPU utilization.

A single TCP flow running on our 40 Gbps testbed reaches 26.8 Gbps with 11.4 µs latency in both secure and non-secure modes. Slim's throughput is the same as the throughput on the host network and is 9% faster than Weave with RFS. Slim's latency is also the same as using the host network, and it is 86% faster than Weave with RFS.

Using Slim, the creation of a TCP connection takes longer because of the need to invoke the user-space router. On our testbed, in a container with Weave, creating a TCP connection takes 270 µs. With the non-secure mode of Slim, it takes 444 µs. With the secure mode, it takes 556 µs. As a reference, creation of a TCP connection on the host network takes 58 µs. This means that Slim is not always better, e.g., if an application has many short-lived TCP connections. We did not observe this effect in the four applications studied because they support persistent connections [112, 116], a common design paradigm.

For long-lived connections, Slim reduces CPU utilization. We measure the CPU utilization using *mpstat* for Slim in secure mode and Weave with RFS when varying TCP throughput from 0 to 25 Gbps. RFS cannot reach 25 Gbps, so we omit that data point. Figure 5.5a shows the total CPU utilization in terms of number of virtual cores consumed. Compared to RFS, CPU overhead declines by 22-41% for Slim; Slim's CPU costs are the same as using the host network directly. To determine the source of this reduction, we break down different components using *mpstat* when TCP throughput is 22.5 Gbps. Figure 5.5b shows the result. As expected, the largest reduction in CPU costs comes from serving software interrupts. These decline 49%: Using Slim, a packet no longer needs



(a) Total CPU Utilization

(b) Breakdown

Figure 5.5: CPU utilization and breakdown for a TCP connection. In Figure 5.5a, the Slim and the host lines overlap. In Figure 5.5b, the usr bar is at the bottom and negligible. Error bars denote standard deviations.

data-plane transformations and traverses the host OS kernel's network stack only once.

Network policies. Slim supports access control, rate limiting and QoS policies, including when applied to existing connections. We examine a set of example scenarios when rate limits and access control are used. We run two parallel containers, each with a TCP connection. The other end of those connections is a container on a different machine. We use *iperf* to report average throughput per half second. Figure 5.6 shows the result.

Without network policy, each container gets around 18-18.5 Gbps. We first set a rate limit of 15 Gbps on one container. The container's throughput immediately drops to around 14 Gbps, and the other container's throughput increases. A slight mismatch occurs between the rate limit we set and our observed throughput, which we suspect is due to *tc* being imperfect. We subsequently set the rate limit to 10 Gbps and 5 Gbps. As expected, the container's throughput declines to 10 and 5 Gbps, respectively, while the other container's throughput increases. Finally, Slim stops rate limiting and sets an ACL to bar the container from communicating with the destination. The affected connection is

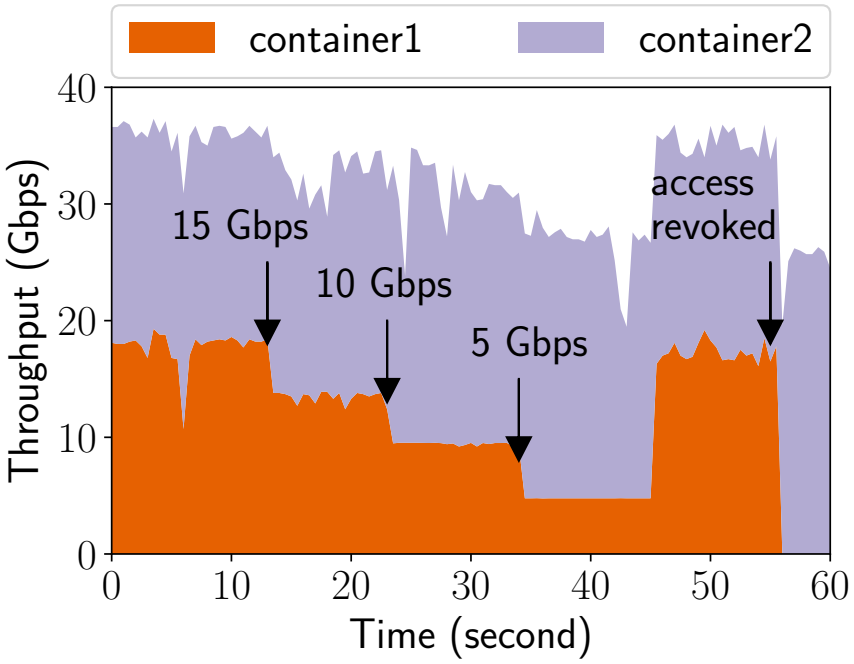


Figure 5.6: A bar graph of the combined throughput of two Slim containers, with rate limit and access control policy updates to one of the containers.

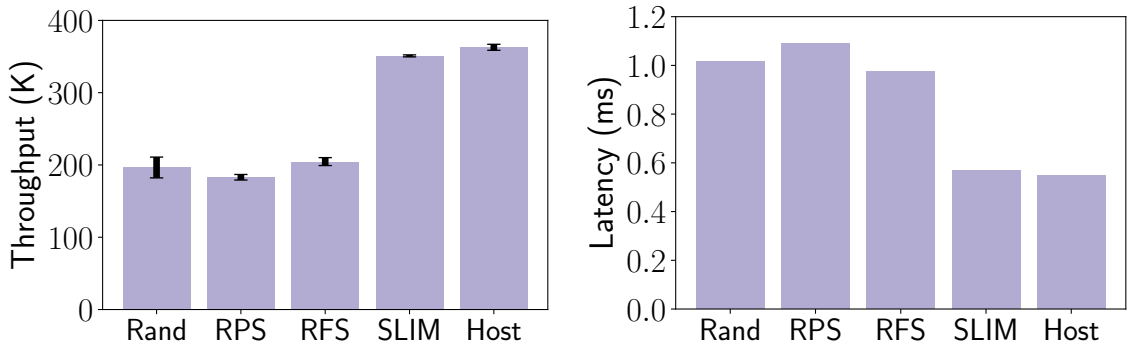
destroyed, and the connection from the other container speeds up to standard connection speed.

5.5.2 Applications

We evaluate Slim with four real world applications: Memcached, Nginx, PostgreSQL, and Apache Kafka. From this point on, our evaluation uses Slim running in secure mode.

Memcached

We measure the performance of Memcached [109] on Slim. We create one container on each of the two physical machines; one runs the Memcached (v1.5.6) server, and the other container runs a standard Memcached benchmark tool, *memtier_benchmark* [110] devel-



(a) Throughput

(b) Latency

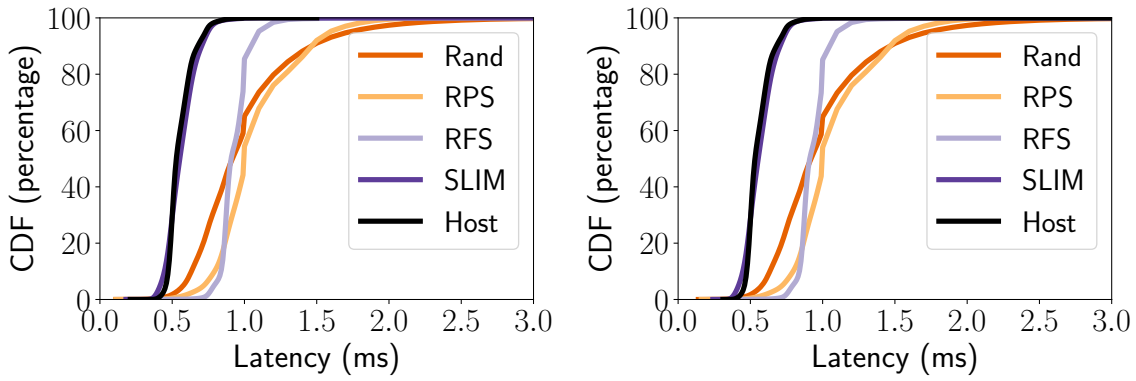
Figure 5.7: Throughput and latency of Memcached with Weave (in various configurations) and with Slim. Error bars in Figure 5.7a shows the standard deviation of completed Memcached operations per-second.

oped by redislab [134]. The benchmark tool spawns 4 threads. Each thread creates 50 TCP connections to the Memcached server and reports the average number of responses per second, the average latency to respond to a memcache command, and the distribution of response latency (SET:GET ratio = 1:10).

Slim improves Memcached throughput (relative to Weave). Figure 5.7a shows the number of total Memcached operations per-second completed on Slim and Weave with different configurations. Receive Flow Steering (RFS) is our best-tuned configuration, yet Slim still outperforms it by 71%. With the default overlay network setting (random IRQ load balancing), Slim outperforms Weave by 79%. Slim's throughput is within 3% of host mode.

Slim also reduces Memcached latency. Figure 5.7b shows the average latency to complete a memcache operation. The average latency reduces by 42% using Slim compared to RFS. The latency of the default setting (random IRQ load balancing), RPS, and RFS are not significantly different (within 5%). Slim's latency is exactly the same as host mode.

Slim also reduces Memcached tail latency. Figure 5.8 shows the CDF of latency for SET



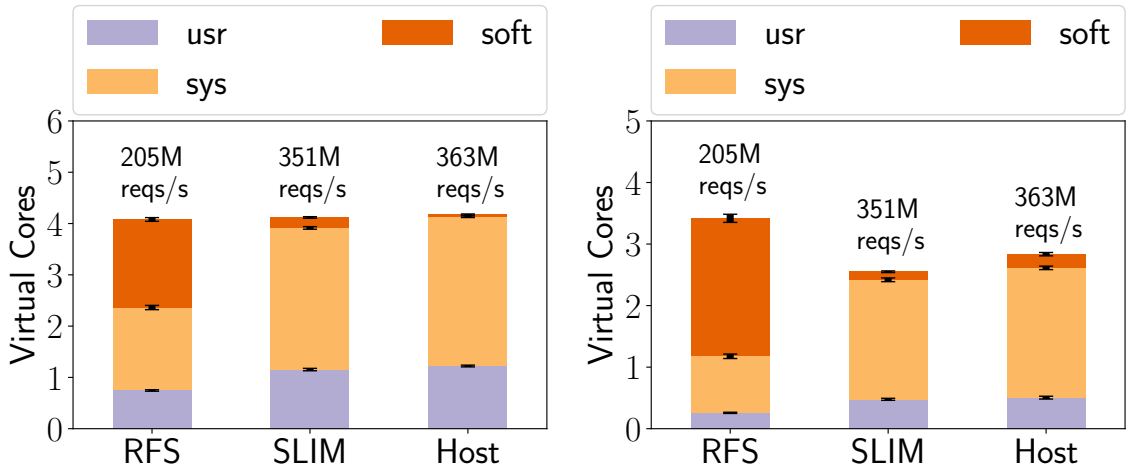
(a) GET

(b) SET

Figure 5.8: Distribution of latency for Memcached SET and GET operations, illustrating tail latency effects. The Slim and Host lines overlap.

and GET operations. The default configuration (i.e., IRQ load balancing) has the worst tail latency behavior. Synchronization overhead depends on temporal kernel state and thus makes latency less predictable. RPS and RFS partially remove the synchronization overhead, improving predictability. Compared to the best configuration, RFS, Slim reduces the 99.9% tail latency by 41%.

Slim reduces the CPU utilization per operation. We measure average CPU utilization on both the client and the Memcached server when *memtier_benchmark* is running. Figure 5.9 shows the result. The total CPU utilization is similar on the client side, while the utilization is 25% lower with Slim on the server compared to RFS. Remember that Slim performs 71% more operations/second. As expected, the amount of CPU utilization in serving software interrupts declines in Slim. We also compare CPU utilization when the throughput is constrained to be identical. Slim reduces CPU utilization by 41% on the Memcached client and 56% on the Memcached server, relative to Weave.



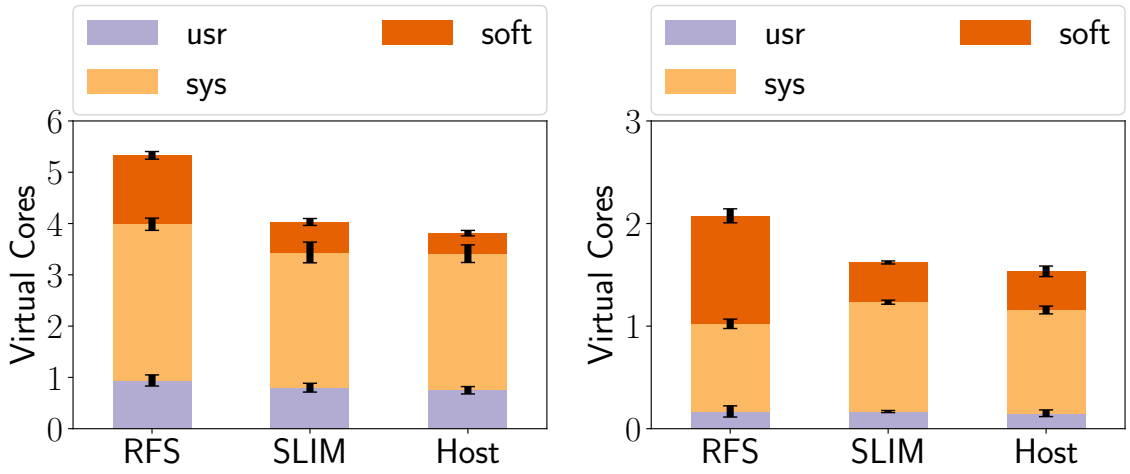
(a) Client (b) Server

Figure 5.9: CPU utilization of Memcached client and server. Error bars denote standard deviations.

Nginx

We run one Nginx (v1.10.3) server in one container and a standard web server benchmarking tool, *wrk2* [148], in another container. Both containers are on two different physical machines. The tool, *wrk2*, spawns 2 threads to create a total of 100 connections to the Nginx server to request an HTML file. This tool lets us set throughput (requests/sec), and it outputs latency. We set up two HTML files (1KB, 1MB) on the Nginx server.

Nginx server's CPU utilization is significantly reduced with Slim. We use *mpstat* to break down the CPU utilization of the Nginx server for scenarios when RFS, Slim, and host can serve the throughput. Figure 5.10 shows the CPU utilization breakdown when the file size is 1KB and the throughput is 60K reqs/second, and also when the file size is 1MB and the throughput is 3K reqs/second. (We choose 60K reqs/second and 3K reqs/second because they are close to the limit of what RFS can handle.). For the 1 KB file, the CPU utilization reduction is 24% compared with RFS. For the 1 MB file, the CPU



(a) 1KB, 60K reqs/s

(b) 1MB, 3K reqs/s

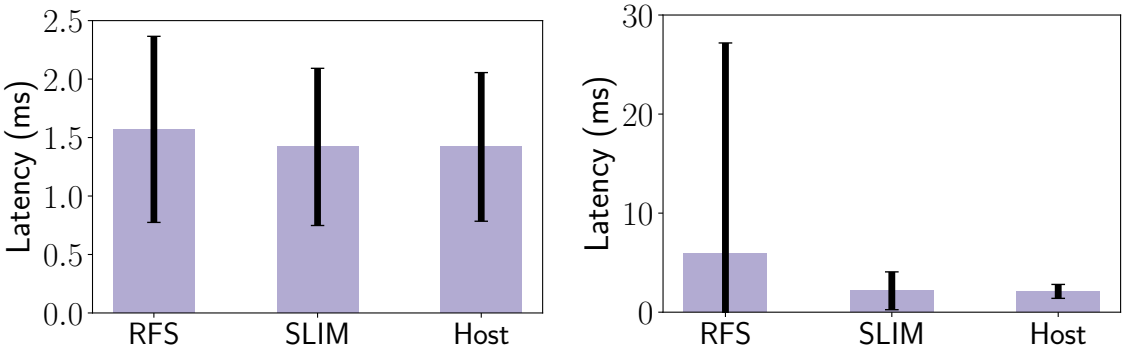
Figure 5.10: CPU utilization breakdown of Nginx. Error bars denote standard deviations.

utilization reduction is 22% compared with RFS. Note that much of the CPU utilization reduction comes from reduced CPU cycles spent in serving software interrupts in the kernel. The CPU utilization still has a 5%-6% gap between Slim and host. We expect this gap is from the longer connection setup time in Slim. Unlike our Memcached benchmark, where connections are pre-established, we observe that *wrk2* creates TCP connections on the fly to send HTTP requests.

While Slim improves the CPU utilization, the improvements to latency are lost in the noise of the natural variance in latency for Nginx. The benchmark tool, *wrk2*, reports the average and the standard deviation of Nginx's latency. Figure 5.11 shows the result. The standard deviation is much larger than the difference of the average latencies.

PostgreSQL

We deploy a relational database, PostgreSQL [128] (version 9.5), in a container and then use its default benchmark tool, *pgbench* [125], to benchmark its performance in another



(a) 1KB, 60K reqs/s

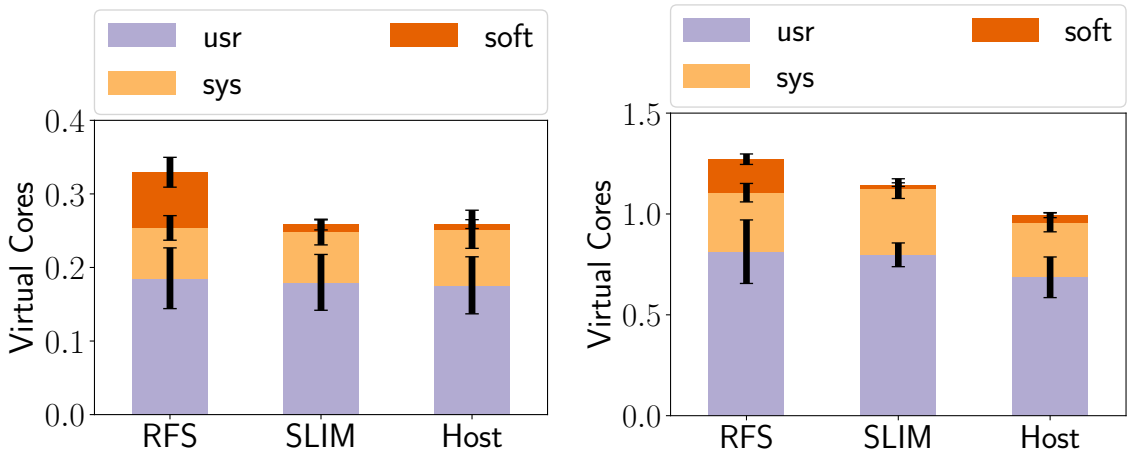
(b) 1MB, 3K reqs/s

Figure 5.11: Latency of Nginx server. Error bars denote standard deviations.

container. The tool, *pgbench*, implements the standard TPC-B benchmark. It creates a database with 1 million banking accounts and executes transactions with 4 threads and a total of 100 connections.

Slim reduces the CPU utilization of PostgreSQL server. We set up *pgbench* to generate 300 transactions per second. (We choose 300 transactions per second because it is close to what RFS can handle.) Figure 5.12a shows the CPU utilization breakdown of the PostgreSQL server. Compared with RFS, Slim reduces the CPU utilization by 22% and the CPU utilization is exactly the same as using host mode networking. Note here, the reduction in CPU utilization is much less than in Memcached and Nginx. The reason is that the PostgreSQL server spends a larger fraction of its CPU cycles in user space, processing SQL queries. Thus, the fraction of CPU cycles consumed by the overlay network is less.

Similar to Nginx, the latency of PostgreSQL naturally has a high variance because of the involvement of disk operations, and it is difficult to conclude any latency benefit of Slim. The benchmark tool, *pgbench*, reports the average and the standard deviation of PostgreSQL's latency. Figure 5.13a shows the results. The standard deviation is much larger than the difference of the mean latencies.



(a) PostgreSQL

(b) Kafka

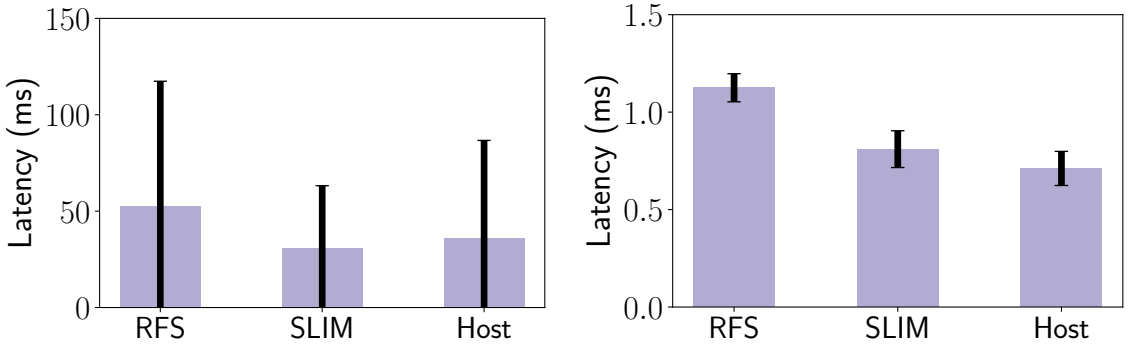
Figure 5.12: CPU utilization of PostgreSQL and Kafka. Error bars denote standard deviations.

Apache Kafka

We now evaluate a popular data streaming framework, Apache Kafka [33, 95]. It is used to build real-time data processing applications. We run Kafka (version 2.0.0) inside one container, and Kafka's default benchmarking tool, *kafka-producer-perf-test*, in another container on a different physical machine.

Slim reduces the CPU utilization of both the Kafka server and the Kafka client. We use *kafka-producer-perf-test* to set throughput to be 500K messages per second. (We choose 500K messages per second because it is close to what RFS can handle.) Each message is 100 bytes and the batch size is 8192. The tool spawns 10 threads that generate messages in parallel. Figure 5.12b shows the breakdown of CPU utilization. The total CPU utilization of the Kafka server reduces by 10% with Slim. The CPU utilization reduction is even smaller than PostgreSQL because Kafka spends more time in user space processing.

Slim reduces message latencies in Kafka. The benchmark tool, *kafka-producer-perf-test*,



(a) PostgreSQL (b) Kafka

Figure 5.13: Latency of PostgreSQL and Kafka. Error bars denote standard deviations.

reports the latency of Kafka. Figure 5.13b shows the results. Kafka's latency reduces by 0.28 ms (28%), compared with RFS. There is still a 0.09 ms latency gap between using Slim and the host mode.

5.5.3 Container Migration

Slim supports container migration. On our testbed, we migrate a Memcached container from one physical server to another physical server on the 40 Gbps network. We test migration 20 times with/without Slim. The container's IP address is kept the same across the migration. Likewise, we do not change the host network's routing table. The container image extracted from the file system is 58 Mbytes.

Using Slim marginally slows down container migration. Table 5.3 is the breakdown of the average container migration time on Weave and on Slim. In total, Slim slows down container migration from 3.34 s to 3.76 s. Slim does not change most of the migration process. The extra overhead is introduced mainly in restoring the file system. With Slim, a container has an additional disk volume containing SlimSocket and also a dummy file to support communication over UNIX domain socket between SlimSocket and SlimRouter.

	Weave	**Slim**
Stop running container	0.75 ± 0.02	0.75 ± 0.02
Extract fs into image	0.43 ± 0.01	0.43 ± 0.01
Transfer container image	0.44 ± 0.01	0.44 ± 0.01
Restore fs	0.82 ± 0.09	1.20 ± 0.10
Start SlimRouter	-	0.003 ± 0.001
Start container	0.90 ± 0.09	0.94 ± 0.17
Total	3.34 ± 0.12 s	3.76 ± 0.20 s

Table 5.3: Time to migrate a Memcached container. The numbers followed by $\pm$ show the standard deviations.

We suspect that the additional disk volume slows down the file system restoration process. Further, starting a container with Slim adds a small amount of additional overhead.

5.6 Discussion

Connection Setup Time. One drawback of Slim is that connection setup time is significantly longer (Chapter 5.5.1). This can penalize applications with many short connections. Slim allows indiviual applications in a container to opt out by detaching SlimSocket. In the future, we want the choice of opting out to be at a per-connection level. We can either (1) allow developers to specify which connection to opt out, or (2) automatically opt out based on predicted flow sizes [56].

Container Live Migration. Although Slim does support quiescent container migration, it does not currently support container live migration. All the TCP connections are disconnected during the migration process, and memory states are not migrated. However, in live migration, live application state has to be fully restored, including state such

as application threads waiting on events inside the OS kernel. Docker is currently experimenting with live checkpointing and restoration with *criu* [50], but it is focused on the simpler case of a single host [53]. Provided a practical live container migration system could be built, Slim would make that more difficult because: (1) the state of the container now includes host namespace file descriptors and (2) data-plane policies (e.g., rate limits) are enforced on host connection identifiers (i.e., five tuples) that would need to be properly translated when migrated.

UDP. The focus of this chapter has been on improving the container communication performance of connection-oriented protocols, such as TCP, by moving operations from the data-plane to connection setup. This poses a challenge for connectionless protocols such as UDP. Slim potentially could support UDP using similar mechanisms as for TCP, by intercepting *socket*, *bind*, *connect*, *sendto*, and *recvfrom*. However, we chose not to do this in our prototype because of two reasons. First, we do not have a good mechanism to support flexible network policy for UDP. In UDP, a file descriptor does not describe a single network pair, but rather an open port to which every node in the virtual network can send packets. Second, the most common use case for UDP in data centers is to avoid the overhead of connection setup; since Slim makes connection setup more expensive, it would subvert some of those benefits. Instead, to work with unmodified applications that may use a mixture of TCP and UDP packets, our prototype simply directs UDP traffic to Weave.

Packet-based Network Policy. A limitation of Slim is that it supports connection-based network policy and not packet-based network policy. For example, a virtual network can be set up to prevent access to a backend database, except from certain containers; Slim supports this kind of access control. Packet-based filters allow the system drop packets if the hash of the overlay packet matches a signature. On Slim, the virtual overlay packet is never constructed and so checking against a signature would be prohibitively expensive. If packet-based network policy is needed, a standard overlay network should be used instead.

LD_PRELOAD. Our prototype uses LD_PRELOAD to dynamically link SlimSocket into unmodified application binaries. Some systems assume statically linked application binaries (e.g., applications written in Go). These can benefit from Slim by patching the application binaries to use SlimSocket instead of POSIX sockets; we do not implement this support in our prototype.

Error Code. Our current prototype implementation is not transparent in one significant way. When an access control list changes, requiring Slim to revoke a file descriptor, the application receives a different error code when it used that file descriptor, relative to Weave. In Slim, a send on a revoked file descriptor returns a bad file descriptor error code, while in Weave the packet would be silently discarded. We believe it is possible to address this but it was not needed for our benchmark applications.

SmartNICs. A recent research trend has been to explore moving common case network data-plane operations to hardware. Catapult [131], for example, moves packet encapsulation required for virtual machine emulation to hardware. Catapult runs as a bump on the wire, however, so in order to offload overlay network processing, Linux would need to be modified to accept virtual network packets on its physical network interface. SR-IOV is commodity hardware, but it suffers from the same problem as macvlan mode. (See Chapter 2.3.) FlexNIC [89] has proposed a flexible model that can incorporate application, guest OS, and virtual machine packet management, but to date it is only experimental hardware.

5.7 Related Work

Network namespace. Mapping resources from a host into a container is not a new idea. In Plan9 [126], resources, such as directories in the file system or process identifiers, are directly mapped between namespaces. Our work revisits Plan9's idea in the networking context, but with performance as a goal, rather than portability. Today's Linux network namespace works at a per-device level, and so is not strong enough for supporting connection-based network virtualization. Slim uses the Linux networking namespace to

isolate the container from using the host network interface.

Host support for efficient virtual networking. Host support for efficient virtual networking is an old topic, mostly in the context of VMs. Menon et al. co-design the driver of the virtual network interface and the hypervisor for efficient virtual network interface emulation [111]. Socket-outsourcing [58], VMCI socket [143], and Slipstream [51] improve intra-host networking. FreeFlow [90] redirects RDMA library calls to create a fast container RDMA network. To the best of our knowledge, Slim is first work that provides network virtualization at TCP connection setup time for unmodified containerized applications.

Redirecting system calls. Redirecting system calls is a useful technique for many purposes, such as taint tracking [60], building user-level file systems [68] and performing other advanced OS kernel features (e.g., sandboxing [91], record and replay [77], and intrusion detection [92]). In a networking context, mTCP [85] redirects socket calls to construct a user-level networking stack. NetKernel [117] redirects socket calls to decouple networking stack from virtual machine images.

Separation of control- and data-plane. The performance gain of Slim comes from moving network virtualization logic from the data- to the control-plane. Separation of the control- and the data-plane is a well-known technique to improve system performance in building fast data-plane operating systems [123, 39] and routing in flexible networks [108].

5.8 Summary

Containers have become the de facto method for hosting distributed applications. The key component for providing portability, the container overlay network, imposes significant overhead in terms of throughput, latency, and CPU utilizations, because it adds a layer to the data-plane. We propose Slim, a low-overhead container overlay network that implements network virtualization by manipulating connection-level metadata. Slim transparently supports unmodified, potentially malicious, applications. Slim improves through-

put of an in-memory key-value store by 71% and reduces latency by 42%. Slim reduces CPU utilization of the in-memory key-value store by 56%, a web server by 22%-24%, a database server by 22%, and a stream processing framework by 10%.

Chapter 6

CONCLUSION

This dissertation explores how to design practical, efficient, and reliable data center communication by addressing problems in existing optical networks and network virtualization solutions. Specifically, we introduce three systems—CorrOpt, RAIL, and Slim.

CorrOpt addresses the problem of packet corruption loss—a substantial type of packet loss in data center networks. This corruption loss can hurt application performance. We have shown that corruption losses have different symptoms and root causes from congestion losses: corruption rate is temporally stable and weakly correlated to its utilization. CorrOpt monitors and mitigates corruption losses by intelligently turning off corrupting links while meeting configured capacity constraints. CorrOpt also generates repair recommendations to speed up link repair. CorrOpt can reduce packet corruption by 3–6 orders of magnitude and can improve repair accuracy from 50% to 80%.

RAIL is a low-cost network architecture for data centers. Our key observation is that today's optical communication technologies are over-engineered for the data center environment, and reducing the over-engineering can lower network cost. We have explored stretching transceiver reach as an approach to reduce the over-engineering. We show that this approach can lower total network cost by up to 10% for 10 Gbps and 44% for 40 Gbps networks. With transceiver reach stretching, a subset of links will become gray links—links with packet corruption losses. RAIL's routing and error correction mechanisms can minimize packet loss affecting application performance in a network with gray links.

Finally, Slim is an operating system kernel design that enables container network virtualization at "almost zero" cost. Unlike traditional network virtualization solutions that depend on packet encapsulation and decapsulation, Slim virtualizes network at a per-

connection level by manipulating per-connection metadata in the operating system kernel. Slim supports unmodified Linux applications. Slim improves the throughput of an in-memory key-value store by 71% and reduces latency by 42%. Slim reduces CPU utilization of the in-memory key-value store by 56%, a web server by 22%-24%, a database server by 22%, and a stream processing framework by 10%.

The three systems described in this dissertation have already been partly adopted in Microsoft. Components of CorrOpt and RAIL are used by Microsoft Azure to monitor and diagnose their optical-based data center network infrastructure. Slim is used by Microsoft Bing to lower network virtualization cost when running distributed machine learning workloads. Facebook has made a similar observation about over-engineering in fiber optics as we did in RAIL. They are currently pushing for a relaxed 100G-CWDM4 specification (reducing reach from 2000m to 500m, link loss budget from 5 dB to 3.5 dB, and operating temperature from 0–70 °C to 15–55 °C) [48].

With the need for higher network bandwidth for large-scale, distributed data processing in the data centers, the design and implementation of efficient and reliable data center networks is going to be increasingly important. Today, it has already been clear that existing approaches for constructing data center communication are outdated (e.g., optical technologies designed for telecommunication, network virtualization based on packet encapsulation), and we need to design communication that is more suitable for the data center environment. While the thesis has improved several aspects of data center networks to achieve greater efficiency and reliability, reaching the final goal of removing communication as the efficiency and reliability bottleneck is going to be a long battle.

BIBLIOGRAPHY

[1] AOI 100G MPO MMF 850nm 70m Transceiver. http://www.high-tech.co.jp/common/sys/product/product00445_01.pdf.

[2] Arista 7050S. https://www.arista.com/assets/data/pdf/Datasheets/7050S_Datasheet.pdf.

[3] Arista 7060CX. https://www.arista.com/assets/data/pdf/Datasheets/7060X_7260X_DS.pdf.

[4] Arista Optics Modules and Cables. https://www.arista.com/assets/data/pdf/Datasheets/arista_transceiver_datasheet.pdf.

[5] Corning OM3 Fiber. https://www.corning.com/media/worldwide/coc/documents/Fiber/PI1468_07-14_English.pdf.

[6] Dropbox. https://www.dropbox.com.

[7] Duplex Zipcord Fiber Optic Cable. http://www.fs.com/c/duplex-zipcord-fiber-optic-cable-1249.

[8] Facebook. https://www.facebook.com.

[9] Finisar. http://www.mouser.com/finisar-corporation/.

[10] Finisar FTLX8571D3BCL. https://www.finisar.com/optical-transceivers/ftlx8571d3bcl.

[11] Fs.com QSFP 40G Transceiver. http://www.fs.com/products/36309.html.

[12] GCI Highlights Tool. https://www.cisco.com/c/en/us/solutions/service-provider/gci-highlights-tool/index.html.

[13] Google Cloud SQL. https://cloud.google.com/sql/.

[14] Google Doc. https://doc.google.com.

[15] HPC Optics. https://www.hpcoptics.com.

[16] IEEE 10 Gb/s Ethernet Task Force. http://grouper.ieee.org/groups/802/3/ae/public/blue_book.pdf.

[17] Microsoft Office. https://www.office.com/.

[18] Netflix Case Study in AWS. https://aws.amazon.com/solutions/case-studies/netflix/.

[19] Optical Monitor MIB. http://www.oidview.com/mibs/9/CISCO-OPTICAL-MONITOR-MIB.html.

[20] PCWholeSale. http://www.pc-wholesale.com/.

[21] ROBOfiber. http://www.robofiber.com/.

[22] Tumblr. https://www.tumblr.com.

[23] VPI. http://www.vpiphotonics.com/index.php.

[24] IEEE802.3ae. *IEEE Std. 1516-2000*, 2000.

[25] 100G PSM4 Specification. http://www.psm4.org/100G-PSM4-Specification-2.0.pdf, 2014.

[26] RAIL VPI Simulation Files. https://github.com/railnsdi2017/rail_VPI, 2016.

[27] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A Scalable, Commodity Data Center Network Architecture. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '08, pages 63–74, New York, NY, USA, 2008. ACM.

[28] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. CONGA: Distributed Congestion-aware Load Balancing for Datacenters. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '14, pages 503–514, New York, NY, USA, 2014. ACM.

[29] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data Center TCP (DCTCP). In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '10, pages 63–74, New York, NY, USA, 2010. ACM.

[30] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less Is More: Trading a Little Bandwidth for Ultra-Low Latency in the Data Center. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 253–266, San Jose, CA, 2012. USENIX.

[31] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pFabric: Minimal Near-optimal Datacenter Transport. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '13, pages 435–446, New York, NY, USA, 2013. ACM.

[32] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A Fast Array of Wimpy Nodes. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 1–14, New York, NY, USA, 2009. ACM.

[33] Apache Kafka. https://kafka.apache.org/.

[34] Behnaz Arzani, Selim Ciraci, Luiz Chamon, Yibo Zhu, Hongqiang Liu, Jitu Padhye, Boon Thau Loo, and Geoff Outhred. 007: Democratically Finding the Cause of Packet Drops. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI'18, pages 419–435, Berkeley, CA, USA, 2018. USENIX Association.

[35] Behnaz Arzani, Selim Ciraci, Boon Thau Loo, Assaf Schuster, and Geoff Outhred. Taking the Blame Game Out of Data Centers Operations with NetPoirot. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '16, pages 440–453, New York, NY, USA, 2016. ACM.

[36] Peter Bailis and Kyle Kingsbury. The Network is Reliable. *Queue*, 12(7):20:20–20:32, July 2014.

[37] Hari Balakrishnan, Venkata N. Padmanabhan, Srinivasan Seshan, and Randy H. Katz. A Comparison of Mechanisms for Improving TCP Performance over Wireless Links. *IEEE/ACM Trans. Netw.*, 5(6):756–769, December 1997.

[38] Andy Bavier, Nick Feamster, Mark Huang, Larry Peterson, and Jennifer Rexford. In VINI Veritas: Realistic and Controlled Network Experimentation. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '06, pages 3–14, New York, NY, USA, 2006. ACM.

[39] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 49–65, Broomfield, CO, 2014. USENIX Association.

[40] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. MicroTE: Fine Grained Traffic Engineering for Data Centers. In *Proceedings of the 7th COnference on Emerging Networking EXperiments and Technologies*, CoNEXT '11, pages 8:1–8:12, New York, NY, USA, 2011. ACM.

[41] Kashif Bilal, Marc Manzano, Samee U. Khan, Eusebi Calle, Keqin Li, and Albert Y. Zomaya. On the Characterization of the Structural Robustness of Data Center Networks. *IEEE Trans. Cloud Computing*, 2013.

[42] Peter Bodík, Ishai Menache, Mosharaf Chowdhury, Pradeepkumar Mani, David A. Maltz, and Ion Stoica. Surviving Failures in Bandwidth-constrained Datacenters. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, pages 431–442, New York, NY, USA, 2012. ACM.

[43] Eric Brewer, Lawrence Ying, Lawrence Greenfield, Robert Cypher, and Theodore Ts'o. Disks for Data Centers. Technical report, Google, 2016.

[44] H. Bulow, W. Baumert, H. Schmuck, F. Mohr, T. Schulz, F. Kuppers, and W. Weiershausen. Measurement of the maximum speed of PMD fluctuation in installed field fiber. In *International Conference on Integrated Optics and Optical Fiber Communication*, 1999.

[45] Calico. https://www.projectcalico.org/.

[46] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. BBR: Congestion-Based Congestion Control. *Queue*, 14(5):50:20–50:53, October 2016.

[47] J. D. Case, M. Fedor, M. L. Schoffstall, and J. Davin. Simple Network Management Protocol (SNMP), 1990.

[48] Abhijit Chakravarty, Katharine Schmidtke, Srinivasan Giridharan, John Huang, and Vincent Zeng. 100G CWDM4 SMF Optical Interconnects for Facebook Data Centers. *Conference on Lasers and Electro-Optics*, 2016.

[49] Xin Chen, J. Abbott, D. Powers, D. Coleman, and Ming-Jun Li. Statistical treatment of IEEE spreadsheet model for VCSEL-multimode fiber transmissions". In *2016 21st OptoElectronics and Communications Conference (OECC) held jointly with 2016 International Conference on Photonics in Switching (PS)*, pages 1–3, July 2016.

[50] CRIU. https://criu.org/Main_Page.

[51] Will Dietz, Joshua Cranmer, Nathan Dautenhahn, and Vikram Adve. Slipstream: Automatic Interprocess Communication Optimization. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 431–443, Santa Clara, CA, 2015. USENIX Association.

[52] Docker. http://www.docker.com.

[53] Docker Checkpoint and Restore. https://github.com/docker/cli/blob/master/experimental/checkpoint-restore.md.

[54] Docker Container Networking. https://docs.docker.com/engine/userguide/networking/.

[55] Docker Swarm. https://docs.docker.com/engine/swarm/.

[56] Vojislav Dukic, Sangeetha Abdu Jyothi, Bojan Karlas, Muhsen Owaida, Ce Zhang, and Ankit Singla. Is Advance Knowledge of Flow Sizes a Plausible Assumption? In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 565–580, Boston, MA, 2019. USENIX Association.

[57] Nandita Dukkipati, Matt Mathis, Yuchung Cheng, and Monia Ghobadi. Proportional Rate Reduction for TCP. In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference*, IMC '11, pages 155–170, New York, NY, USA, 2011. ACM.

[58] Hideki Eiraku, Yasushi Shinjo, Calton Pu, Younggyun Koh, and Kazuhiko Kato. Fast Networking with Socket-outsourcing in Hosted Virtual Machine Environments. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, pages 310–317, New York, NY, USA, 2009. ACM.

[59] Nosayba El-Sayed, Ioan A. Stefanovici, George Amvrosiadis, Andy A. Hwang, and Bianca Schroeder. Temperature Management in Data Centers: Why Some (Might) Like It Hot. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 163–174, New York, NY, USA, 2012. ACM.

[60] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 393–407, Berkeley, CA, USA, 2010. USENIX Association.

[61] Etcd: A Distributed, Reliable Key-Value Store for the Most Critical Data of a Distributed System. https://coreos.com/etcd/.

[62] Robert J. Feuerstein. Field Measurements of Deployed Fiber. In *Optical Fiber Communication Conference and Exposition and The National Fiber Optic Engineers Conference*, page NThC4. Optical Society of America, 2005.

[63] FiberStore. Fiber Optic Inspection Tutorial. http://www.fs.com/fiber-optic-inspection-tutorial-aid-460.html, 2017.

[64] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 51–66, Renton, WA, 2018. USENIX Association.

[65] Flannel. https://github.com/coreos/flannel.

[66] Fiber for Learning. Fiber Hygiene. http://fiberforlearning.com/welcome/2010/09/20/connector-cleaning/, 2017.

[67] Daniel A. Freedman, Tudor Marian, Jennifer H. Lee, Ken Birman, Hakim Weatherspoon, and Chris Xu. Exact Temporal Characterization of 10 Gbps Optical Wide-area Network. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, pages 342–355, New York, NY, USA, 2010. ACM.

[68] Filesystem in Userspace. https://github.com/libfuse/libfuse.

[69] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[70] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[71] Monia Ghobadi, Jamie Gaudette, Ratul Mahajan, Amar Phanishayee, Buddy Klinkers, and Daniel Kilper. Evaluation of Elastic Modulation Gains in Microsoft's Optical Backbone in North America. In *Optical Fiber Communication Conference*, page M2J.2. Optical Society of America, 2016.

[72] Monia Ghobadi and Ratul Mahajan. Optical Layer Failures in a Large Backbone. In *Proceedings of the 2016 Internet Measurement Conference*, IMC '16, pages 461–467, New York, NY, USA, 2016. ACM.

[73] Monia Ghobadi, Ratul Mahajan, Amar Phanishayee, Nikhil Devanur, Janardhan Kulkarni, Gireeja Ranade, Pierre-Alexandre Blanche, Houman Rastegarfar, Madeleine Glick, and Daniel Kilper. ProjecToR: Agile reconfigurable data center interconnect. *SIGCOMM '16*, pages 216–229, 2016.

[74] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '11, pages 350–361, New York, NY, USA, 2011. ACM.

[75] Ramesh Govindan, Ina Minei, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. Evolve or Die: High-Availability Design Principles Drawn from Googles Network Infrastructure. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '16, pages 58–72, New York, NY, USA, 2016. ACM.

[76] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. VL2: A Scalable and Flexible Data Center Network. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '09, pages 51–62, New York, NY, USA, 2009. ACM.

[77] Zhenyu Guo, Xi Wang, Jian Tang, Xuezheng Liu, Zhilei Xu, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. R2: An Application-level Kernel for Record and Replay. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and*

Implementation, OSDI'08, pages 193–208, Berkeley, CA, USA, 2008. USENIX Association.

[78] Sangtae Ha, Injong Rhee, and Lisong Xu. CUBIC: A New TCP-friendly High-speed TCP Variant. *SIGOPS Oper. Syst. Rev.*, 42(5):64–74, July 2008.

[79] Brandon Heller, Srini Seetharaman, Priya Mahadevan, Yiannis Yiakoumis, Puneet Sharma, Sujata Banerjee, and Nick McKeown. ElasticTree: Saving Energy in Data Center Networks. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pages 17–17, Berkeley, CA, USA, 2010. USENIX Association.

[80] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 295–308, Berkeley, CA, USA, 2011. USENIX Association.

[81] Introducing Data Center Fabric, the Next-Generation Facebook Data Center Network. https://engineering.fb.com/production-engineering/introducing-data-center-fabric-the-next-generation-facebook-data-center-network/, 2014.

[82] Iperf. https://iperf.fr/.

[83] Iptables. https://linux.die.net/man/8/iptables.

[84] JDSU. P5000i Fiber Microscope. http://www.viavisolutions.com/en-us/products/p5000i-fiber-microscope, 2017.

[85] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 489–502, Seattle, WA, 2014. USENIX Association.

[86] H. C. Ji, J. H. Lee, and Y. C. Chung. System Outage Probability due to Dispersion Variation Caused by Seasonal and Regional Temperature Variations. In *OFC/NFOEC Technical Digest. Optical Fiber Communication Conference, 2005.*, volume 1, pages 3 pp. Vol. 1–, March 2005.

[87] Edward John Forrest Jr. *How to Precision Clean All Fiber Optic Connections: A Step By Step Guide*. CreateSpace Independent Publishing Platform, 2014.

[88] O. Karlsson, J. Brentel, and P. A. Andrekson. Long-Term measurement of PMD and Polarization Drift in Installed Fibers. *Journal of Lightwave Technology*, 18(7):941–951, July 2000.

[89] Antoine Kaufmann, Simon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. High Performance Packet Processing with FlexNIC. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 67–81, New York, NY, USA, 2016. ACM.

[90] Daehyeok Kim, Tianlong Yu, Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Shachar Raindel, Chuanxiong Guo, Vyas Sekar, and Srinivasan Seshan. FreeFlow: Software-based Virtual RDMA Networking for Containerized Clouds. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 113–126, Boston, MA, 2019. USENIX Association.

[91] Taesoo Kim and Nickolai Zeldovich. Practical and Effective Sandboxing for Non-root Users. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 139–144, Berkeley, CA, USA, 2013. USENIX Association.

[92] Samuel T. King and Peter M. Chen. Backtracking Intrusions. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 223–236, New York, NY, USA, 2003. ACM.

[93] Ramana Rao Kompella, Albert Greenberg, Jennifer Rexford, Alex C. Snoeren, and Jennifer Yates. Cross-Layer Visibility as a Service. In *In Proc. ACM SIGCOMM HotNets Workshop*, 2005.

[94] Teemu Koponen, Keith Amidon, Peter Balland, Martín Casado, Anupam Chanda, Bryan Fulton, Igor Ganichev, Jesse Gross, Natasha Gude, Paul Ingram, Ethan Jackson, Andrew Lambeth, Romain Lenglet, Shih-Hao Li, Amar Padmanabhan, Justin Pettit, Ben Pfaff, Rajiv Ramanathan, Scott Shenker, Alan Shieh, Jeremy Stribling, Pankaj Thakkar, Dan Wendlandt, Alexander Yip, and Ronghua Zhang. Network Virtualization in Multi-tenant Datacenters. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 203–216, Berkeley, CA, USA, 2014. USENIX Association.

[95] Jay Kreps, Neha Narkhede, and Jun Rao. Kafka: a Distributed Messaging System for Log Processing. In *International Workshop on Networking Meets Databases (NetDB 2016*, 2016.

[96] Kubernetes: Cluster Networking. https://kubernetes.io/docs/concepts/cluster-administration/networking/.

[97] Parantap Lahiri, George Chen, Petr Lapukhov, Edet Nkposong, Dave Maltz, Robert Toomey, and Lihua Yuan. Routing Design for Large Scale Data Centers: BGP is a Better IGP! https://www.nanog.org/meetings/nanog55/presentations/Monday/Lapukhov.pdf.

[98] Jonathan C. Li, Kerry Hinton, Peter M. Farrell, and Sarah D. Dods. Optical Impairment Outage Computation. *Opt. Express*, 16(14):10529–10534, Jul 2008.

[99] Hongqiang Harry Liu, Xin Wu, Ming Zhang, Lihua Yuan, Roger Wattenhofer, and David Maltz. zUpdate: Updating Data Center Networks with Zero Loss. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 411–422, New York, NY, USA, 2013. ACM.

[100] Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno P. Lopes, Andrey Rybalchenko, Guohan Lu, and Lihua Yuan. CrystalNet: Faithfully Emulating Large Production Networks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 599–613, New York, NY, USA, 2017. ACM.

[101] Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, and Thomas Anderson. F10: A Fault-Tolerant Engineered Network. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, pages 399–412, Lombard, IL, 2013. USENIX.

[102] Vincent Liu, Danyang Zhuo, Simon Peter, Arvind Krishnamurthy, and Thomas Anderson. Subways: A Case for Redundant, Inexpensive Data Center Edge Links. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '15, pages 27:1–27:13, New York, NY, USA, 2015. ACM.

[103] Networking using a Macvlan Network. https://docs.docker.com/network/network-tutorial-macvlan/.

[104] David Maltz. Keeping Cloud-Scale Networks Healthy. https://video.mtgsf.com/video/4f277939-73f5-4ce8-aba1-3da70ec19345, 2016.

[105] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My VM is Lighter (and Safer) Than Your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 218–233, New York, NY, USA, 2017. ACM.

[106] T. Marian, D. A. Freedman, K. Birman, and H. Weatherspoon. Empirical Characterization of Uncongested Optical Lambda Networks and 10GbE Commodity Endpoints. In *2010 IEEE/IFIP International Conference on Dependable Systems Networks (DSN)*, pages 575–584, June 2010.

[107] Saverio Mascolo, Claudio Casetti, Mario Gerla, M. Y. Sanadidi, and Ren Wang. TCP Westwood: Bandwidth Estimation for Enhanced Transport over Wireless Links. In *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking*, MobiCom '01, pages 287–297, New York, NY, USA, 2001. ACM.

[108] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Computer Communication Review*, 38(2):69–74, March 2008.

[109] Memcached. https://memcached.org/.

[110] Memtier_benchmark. https://github.com/RedisLabs/memtier_benchmark.

[111] Aravind Menon, Alan L. Cox, and Willy Zwaenepoel. Optimizing Network Virtualization in Xen. In *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference*, ATEC '06, pages 2–2, Berkeley, CA, USA, 2006. USENIX Association.

[112] Jeffrey C. Mogul. The Case for Persistent-connection HTTP. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '95, pages 299–313, New York, NY, USA, 1995. ACM.

[113] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing Software Defined Networks. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 1–13, Lombard, IL, 2013. USENIX Association.

[114] Jayaram Mudigonda, Praveen Yalagandula, Jeff Mogul, Bryan Stiekes, and Yanick Pouffary. NetLord: A Scalable Multi-tenant Network Architecture for Virtualized Datacenters. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '11, pages 62–73, New York, NY, USA, 2011. ACM.

[115] Nginx. https://nginx.org/.

[116] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, Lombard, IL, 2013. USENIX.

[117] Zhixiong Niu, Hong Xu, Dongsu Han, Peng Cheng, Yongqiang Xiong, Guo Chen, and Keith Winstein. Network Stack As a Service in the Cloud. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, HotNets-XVI, pages 65–71, New York, NY, USA, 2017. ACM.

[118] netpipe(1) - Linux man page. https://linux.die.net/man/1/netpipe.

[119] Scaling in the Linux Networking Stack. https://www.kernel.org/doc/Documentation/networking/scaling.txt.

[120] Jitendra Padhye, Victor Firoiu, Don Towsley, and Jim Kurose. Modeling TCP Throughput: A Simple Model and Its Empirical Validation. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '98, pages 303–314, New York, NY, USA, 1998. ACM.

[121] David A. Patterson, Garth Gibson, and Randy H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, SIGMOD '88, pages 109–116, New York, NY, USA, 1988. ACM.

[122] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. Fastpass: A Centralized "Zero-queue" Datacenter Network. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '14, pages 307–318, New York, NY, USA, 2014. ACM.

[123] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The Operating System is the Control Plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 1–16, Broomfield, CO, 2014. USENIX Association.

[124] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martin Casado. The Design and Implementation of Open vSwitch. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 117–130, Oakland, CA, 2015. USENIX Association.

[125] Pgbench. https://www.postgresql.org/docs/9.5/static/pgbench.html.

[126] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom. The Use of Name Spaces in Plan 9. *SIGOPS Oper. Syst. Rev.*, 27(2):72–76, April 1993.

[127] Lucian Popa, Sylvia Ratnasamy, Gianluca Iannaccone, Arvind Krishnamurthy, and Ion Stoica. A Cost Comparison of Datacenter Network Architectures. In *Proceedings of the 6th COnference on Emerging Networking EXperiments and Technologies*, CoNEXT '10, pages 16:1–16:12, New York, NY, USA, 2010. ACM.

[128] PostgreSQL. https://www.postgresql.org/.

[129] Proc - Process Information Pseudo-Filesystem. http://man7.org/linux/man-pages/man5/proc.5.html.

[130] Intel P-State Driver. https://www.kernel.org/doc/Documentation/cpu-freq/intel-pstate.txt.

[131] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 13–24, Piscataway, NJ, USA, 2014. IEEE Press.

[132] Zafar Ayyub Qazi, Cheng-Chun Tu, Luis Chiang, Rui Miao, Vyas Sekar, and Minlan Yu. SIMPLE-fying Middlebox Policy Enforcement Using SDN. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '13, pages 27–38, New York, NY, USA, 2013. ACM.

[133] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. Improving Datacenter Performance and Robustness with Multipath TCP. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '11, pages 266–277, New York, NY, USA, 2011. ACM.

[134] Redislab. https://redislabs.com/.

[135] Luigi Rizzo. Effective erasure codes for reliable computer communication protocols. *SIGCOMM Computer Communication Review*, 27(2):24–36, April 1997.

[136] S. Savage, T. Anderson, A. Aggarwal, D. Becker, N. Cardwell, A. Collins, E. Hoffman, J. Snell, A. Vahdat, G. Voelker, and J. Zahorjan. Detour: Informed Internet Routing and Transport. *IEEE Micro*, 19(1):50–59, Jan 1999.

[137] SECure COMPuting with Filters. https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt.

[138] Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick McKeown, and Guru Parulkar. Can the Production Network Be the Testbed? In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 365–378, Berkeley, CA, USA, 2010. USENIX Association.

[139] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '15, pages 183–197, New York, NY, USA, 2015. ACM.

[140] Peng Sun, Ratul Mahajan, Jennifer Rexford, Lihua Yuan, Ming Zhang, and Ahsan Arefin. A Network-state Management Service. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '14, pages 563–574, New York, NY, USA, 2014. ACM.

[141] USConec. Single Fiber Cleaning Tools. http://www.usconec.com/products/cleaning_tools/ibc_brand_cleaners_for_single_fiber_connections.htm, 2017.

[142] Balajee Vamanan, Jahangir Hasan, and T.N. Vijaykumar. Deadline-Aware Datacenter TCP (D2TCP). *SIGCOMM Computer Communication Review*, 42(4):115–126, August 2012.

[143] VMCI Socket Performance. https://www.vmware.com/techpapers/2009/vmci-socket-performance-10075.html.

[144] Weave. https://www.weave.works/.

[145] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. Better Never Than Late: Meeting Deadlines in Datacenter Networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '11, pages 50–61, New York, NY, USA, 2011. ACM.

[146] Damon Wischik, Costin Raiciu, Adam Greenhalgh, and Mark Handley. Design, Implementation and Evaluation of Congestion Control for Multipath TCP. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 99–112, Berkeley, CA, USA, 2011. USENIX Association.

[147] S.L. Woodward, L.E. Nelson, M.D. Feuer, X. Zhou, P.D. Magill, S. Foo, D. Hanson, H. Sun, M. Moyer, and M. O'Sullivan. Characterization of Real-Time PMD and Chromatic Dispersion Monitoring in a High-PMD 46-Gb/s Transmission System. *Photonics Technology Letters, IEEE*, 2008.

[148] Wrk2. https://github.com/giltene/wrk2.

[149] Xin Wu, Daniel Turner, Chao-Chih Chen, David A. Maltz, Xiaowei Yang, Lihua Yuan, and Ming Zhang. NetPilot: Automating Datacenter Network Failure Mitigation. *SIGCOMM Computer Communication Review*, 42(4):419–430, August 2012.

[150] Kyriakos Zarifis, Rui Miao, Matt Calder, Ethan Katz-Bassett, Minlan Yu, and Jitendra Padhye. DIBS: Just-in-time Congestion Mitigation for Data Centers. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 6:1–6:14, New York, NY, USA, 2014. ACM.

[151] Junlan Zhou, Malveeka Tewari, Min Zhu, Abdul Kabbani, Leon Poutievski, Arjun Singh, and Amin Vahdat. WCMP: Weighted Cost Multipathing for Improved Fairness in Data Centers. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 5:1–5:14, New York, NY, USA, 2014. ACM.

[152] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion Control for Large-Scale RDMA Deployments. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '15, pages 523–536, New York, NY, USA, 2015. ACM.

[153] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y. Zhao, and Haitao Zheng. Packet-Level Telemetry in Large Datacenter Networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '15, pages 479–491, New York, NY, USA, 2015. ACM.

[154] Danyang Zhuo, Monia Ghobadi, Ratul Mahajan, Klaus-Tycho Förster, Arvind Krishnamurthy, and Thomas Anderson. Understanding and Mitigating Packet Corruption in Data Center Networks. In *Proceedings of the Conference of the ACM Special*

Interest Group on Data Communication, SIGCOMM '17, pages 362–375, New York, NY, USA, 2017. ACM.

[155] Danyang Zhuo, Monia Ghobadi, Ratul Mahajan, Amar Phanishayee, Xuan Kelvin Zou, Hang Guan, Arvind Krishnamurthy, and Thomas Anderson. RAIL: A Case for Redundant Arrays of Inexpensive Links in Data Center Networks. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 561–576, Boston, MA, 2017. USENIX Association.

[156] Danyang Zhuo, Kaiyuan Zhang, Yibo Zhu, Hongqiang Harry Liu, Matthew Rockett, Arvind Krishnamurthy, and Thomas Anderson. Slim: OS Kernel Support for a Low-Overhead Container Overlay Network. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 331–344, Boston, MA, 2019. USENIX Association.

Appendix A

NP-COMPLETENESS OF FINDING THE BEST SET OF LINKS TO DISABLE IN CorrOpt

To prove Theorem 1, we first prove the following Lemma:

Lemma 1. *Let $N = (V, E)$ be a degraded Fat-Tree, where some links are turned off. Let $L \subseteq E$ be the set of enabled links with corruption. Finding a set $L' \subseteq L$ whose removal minimizes the impact of packet corruption s.t. all ToR switch pairs are still connected the the spine via valley-free routing after the removal of L' is NP-hard.*

Proof. Our NP-hardness reduction is via the NP-complete problem 3-SAT, in the variant with exactly three literals per clause. Let I be an instance of 3-SAT [69], with k clauses $C_1, \ldots, C_k$ and variables $x_1, \ldots, x_r$, $k \geq r$. We create an instance I' of our problem as follows: Consider a $4k$-Fat-Tree, consisting of the three layers ToR, fabric, and the spine, where the fabric switches have $2k$ links down- and upwards, respectively. Pick one pod P with $2k$ ToR switches, $C_1, \ldots, C_k$, corresponding to the clauses in I, and $H_1, \ldots, H_k$, as helper switches, and lastly, $2r$ fabric switches $X_1, \neg X_1, \ldots, X_r, \neg X_r$, corresponding to the possible literals in I, and, $2k - 2r \geq 0$ further fabric switches $A_1, \ldots, A_{2k-2r}$. Let only the following three sets of links not be turned off in this pod P: 1) For each C_i, the links pointing to the fabric switches representing the corresponding literals contained in the clause in I, 2) for each H_j from $H_1, \ldots, H_r$, one link each to $X_j, \neq X_j$, 3, for each $H_{r+1}, \ldots, H_k$, one link each to $X_1, \neg X_1$. For the connection of the fabric switches in P to the neighboring spine switches, let only the following links L not be turned off, with $|L| = 2r$: From each fabric switch $X_1, \neg X_1, \ldots, X_r, \neg X_r$ in P, one link to a neighboring spine switch. We set all links in L to have the same corruption properties greater than zero. The construction is illustrated in Figure A.1.

To guarantee valley-free connection of all ToR switches in P to all other ToR switches in the other pods via the spine, each ToR switch C_i and H_j needs to have a connection to the spine, with the last part of each of those paths being a link from L. To maximize the set of links $L' \subseteq L$ that can be turned off, consider the following: First, for each pair of fabric switches $X_j, \neg X_j$, at least one needs to remain connected to the spine, or the ToR switch H_j would be disconnected (or for the case of $j = 1$, even more switches), meaning $|L'| \geq r$. Second, to connect each switch C_i to the spine, at least one its fabric switches (representing the literals of the clause) have to be connected to the spine. As thus, a solution to a satisfiable 3-SAT instance I tells us how to to pick which of the links from each $X_i, \neg X_i$ pair should remain connected to the spine, and vice versa, a solution with $|L'| = r$ from I' shows how to satisfy I. On the other hand, should I not be satisfiable, then no solution with $|L'| \leq r$ can exist for I', and vice versa as well. We note that the size of the instance I' is polynomial in k, i.e., we showed NP-hardness. □

Since we assumed the same f_l on every link, I can be chosen arbitrarily, as long as $I(f_l) > 0$. We can now prove Theorem 1:

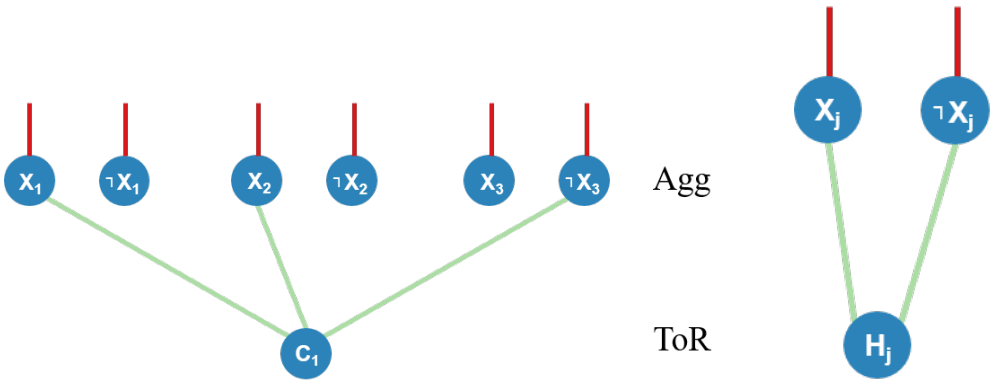
Proof. 1 assumed the network to be already degraded, i.e., some links $\overline{L}$ are turned off. Note that a Fat-Tree is a special case of a Clos topology. We can extend the NP-hardness to the setting of Theorem 1 as well, by the following change in construction: Assume the errors on every link in $\overline{L}$ are so high in comparison to the errors on the links from L, that for every link $l \in \overline{L}$ holds: Disabling l is more efficient regarding the impact of currption than turning off *all* links in L. Lastly, to show that the underlying decision problem is in NP, observe that checking the capacity constraints and total impact of packet corruption of a given solution can be performed in polynomial time. □

The above proof constructions can also be used as follows:

Optimizing for link removal Another objective instead of total packet corruption could be the number of further links that can be removed. However, as all links in the proof of

1 had the same error properties, NP-completeness still holds.

From ToR-spine connectivity to ToR-ToR connectivity Lastly, in the above problem formulations, we wanted to maintain the ToR to spine connectivity of all ToR switches. However, we just considered a single pod P in each Fat-Tree construction, with the ToR switches in all other pods retaining all their connectivity to the spine. As thus, the above problems are also NP-complete for ToR-ToR connectivity, with the same proof constructions.



(a) Clause gadget for $C = (X_1 \vee X_2 \vee \neg X_3)$. Each clause needs to connect to the spine via one of its contained literals.

(b) Variable gadget

Figure A.1: Reduction from 3-SAT: Each clause C is represented by a ToR switch, the literals X and $\neg X$ of each variable are represented by fabric switches. The ToR switches C are connected to the literals contained in their corresponding clause, cf. the clause gadget in Figure A.1a: Hence, at least one of the three literals needs to be connected to the spine. Each literal pair $X, \neg X$ is connected to at least one further ToR switch H, as shown in the variable gadget in Figure A.1b, ensuring that at least one literal of each variable is connected to the spine. The literals (fabric switches) only have one connection each to the spine, and all these connections are faulty, with the same error properties each. Finding a maximum set of these faulty links to turn off, s.t. all Tor switches still have valley-free connections to the remaining network, is NP-hard.

Appendix B

ANALYSIS OF EXPECTED FRACTION OF GOOD PATHS IN RAIL

Definition 2. *A path is good when all unidirectional links on the path are good.*

Lemma 2. *For a path with h hops, when each hop has the probability of at least l to be good and independent, the path is good with the probability of at least l^h.*

Proof. Every hop's quality is independent. □

Theorem 2. *When every path is good with the probability of at least P_{each}, the expected fraction of good paths is at least P_{each}.*

Proof. Let P_i be the probability of a path i to be good. Let's define an indication variable I_p such that

$$I_p = \begin{cases} 0 \text{ , if path p is not good,} \\ 1 \text{ , if path p is good} \end{cases}$$

If the network has n paths, the expected fraction of paths is

$$\text{Expected fraction of good paths} = E(\frac{\sum_{p \in \text{all paths}} I_p}{n})$$

Using the linearity of expected value, we get

$$\text{Expected fraction of good paths} = \frac{\sum_{p \in \text{all paths}} E(I_p)}{n}$$

Because $E(I_p) \geq P_{each}$,

$$\text{Expected fraction of good paths} \geq \frac{nP_{each}}{n} = P_{each}$$

□

Theorem 3. *If each link has the probability of at least l to be good and independent, and the longest path in the network has h hops, then the expected fraction of good paths is at least l^h.*

Proof. Use 2 to get a lower bound of the probability of any path to be good. Then use Theorem 2. □

Appendix C

FIND THE WORST PATH IN CLOS TOPOLOGY IN RAIL

We seek an efficient solution to the following problem: Assuming ECMP routing, given a subset of links on a Clos network and every link's unidirectional packet error rate (LPER), find the worst end-to-end path with the highest path packet error rate (PPER).

This problem can be solved efficiently because of the data center's unique topology and its simple ECMP routing scheme. We describe our algorithms assuming the Clos topology has an oversubscription ratio of 1 across all stages. All our results hold when oversubscription is introduced.

Definition 3. *A Clos network is a multi-stage network. A switch at (i)th stage can only connect to switches at neighbor stages (i.e., $(i \pm 1)$th stage). Every stage has n switches, except for the top stage which has $\frac{n}{2}$ switches. All switches in the network have $2k$ ports. The network has $\log_k n$ stages. Every switch except on the top stage has k ports facing the upper stage and k ports facing the lower stage. Switches on the top stage have all $2k$ ports facing the lower stage.*

Today, data center operators chose ECMP up-down routing as the basic routing algorithm. In this method, packets first travel to an upper stage switch and then down to the destination top-of-rack (ToR) switch. ECMP requires packets to take one of the shortest paths.

Definition 4. *A path from ToRs (stage = 1) switches a to b ($a \neq b$) is called an up-down path when there is a switch c on the path such that the stage number of each switch monotonically increases from a to c and monotonically decreases from c to b.*

Clos topology has a unique property, in that every up-down path can actually be chosen to forward traffic. All up-down paths between the same ToR-pair have exactly the

same number of hops. When we want to find the worst path, we only have to consider all the up-down paths.

The worst path is defined as the following.

Definition 5. *A worst path is the up-down path with highest PPER. PPER of path p is* $1 - \prod_{l \in p}(1 - LPER_l)$.

It is computationally infeasible to track PPER for every path in the network. Tracking PPER takes at least $O(n^3)$ running time because a Clos network has $O(n^3)$ paths (i.e., $O(n)$ paths for every ToR pair and there are $O(n^2)$ ToR pairs). As an alternative, the worst path computation must quickly neglect paths with no hope of becoming the worst path in the run-time.

Algorithm 2 identifies the worst path with a running time of $O(n \log n)$ and is provably optimal in asymptotic running-time. In this section, we prove the path produced by the algorithm is indeed the worst path; we then prove no faster algorithm exists.

Lemma 3. *The output of Algorithm 2 is an up-down path.*

Proof. We only need to show $\forall s, top_s$ is a valid path. From Algorithm 2, top_s is nothing but a monotonically up-going path to s and a monotonically downward path from s.

Thus, top_s is an up-down path if the source and the destination ToR of top_s are different. In Algorithm 2, we see when we calculate top_s, we enforce the direct children of s to be different. In Clos network, this means the source and the destination ToR are in different branches of the Clos (subtree of fat tree) containing non-overlapping sets of ToRs. Thus, top_s is an up-down path. □

Theorem 4. *The output of Algorithm 2 is the worst path.*

Proof. Proof by contradiction. Our algorithm outputs path p. The worst path is p' such that $PPER_{p'} > PPER_p$.

Without loss of generality, p' is an up-down path from a' to b'. Because p' is also an up-down path, by definition, there must be a c' on p' such that c' is the highest stage

switch on p', a' to c' is a monotonic upward path, and c' to b' is a monotonic downward path.

When our algorithm goes through switch c', there are two situations. (1) $p' = top_{s'}$ (2) $p' \neq top_{s'}$. Let's talk about both situations.

If $p' = top_{s'}$, because $PPER_p = max_{s \in all\ switches}(top_s)$, $PPER_p \geq top'_s = p'$. This contradicts the assumption that $PPER_{p'} > PPER_p$.

If $p' \neq top_{s'}$ and p' is valid, p' must includes two children of s'. Then, it must be the case that $p' < top_{s'}$ because, otherwise, p' will be chosen when computing $top_{s'}$. Because $PPER_p = max_{s \in all\ switches}(top_s)$, then $PPER_p \geq top_{s'} > p'$. This contradicts the assumption that $PPER_{p'} > PPER_p$.

Overall, $PPER_{p'} \leq PPER_p$. Thus, the output of our algorithm is the worst path.

□

Theorem 5. *The running time of our algorithm is* $\Theta(n \log n)$.

Proof. Our algorithm passes through each switch once, and the update algorithm is $\Theta(k^2)$. Because k is constant, the update part is in the order of number of switches, $\Theta(n \log_k n)$. The comparison part compares top_s for all switches which takes another $\Theta(n \log_k n)$. Thus, our algorithm finishes in $\Theta(nlog_k n)$. Because k is constant, our algorithm finishes in $\Theta(n \log n)$. □

Theorem 6. *Any algorithm that can compute the worst path has a running time of at least* $\Theta(n \log n)$.

Proof. Proof by contradiction. Assume an algorithm exists that computes the worst path with running-time less than $\Theta(n \log n)$.

We construct an adversarial oracle that always returns LPER = 0.5 for every link queried by the algorithm. After the algorithm finishes, the algorithm returns a worst path p. Because the number of links is in the order of $\Theta(nlog_k n)$, the algorithm does not have enough time to read LPER on every link. There must be a fraction of links that are

not read by the algorithm. Let l be one of the unidirectional links whose LPER is not read by the algorithm.

There are two situations here. (1) l is on p; (2) l is not on p. We discuss both situations.

If l is on p, the oracle sets $LPER_l \leftarrow 0$ and LPER of all remaining links (those not read by the algorithm) at 0.5. Thus, l cannot be the worst path because it has lower PPER than other paths. This contradicts the assumption that p is the worst path.

If l is not on p, the oracle sets $LPER_l \leftarrow 1$ and LPER of all remaining links (those not read by the algorithm) at 0.5. Thus, l must be on the worst path of any path going through l has higher PPER (PPER = 1) than any path that does not go through l. This contradicts the assumption that p is the worst path.

Overall, the algorithm cannot output the correct worst path without taking at least $\Theta(n \log n)$.

□

Appendix D

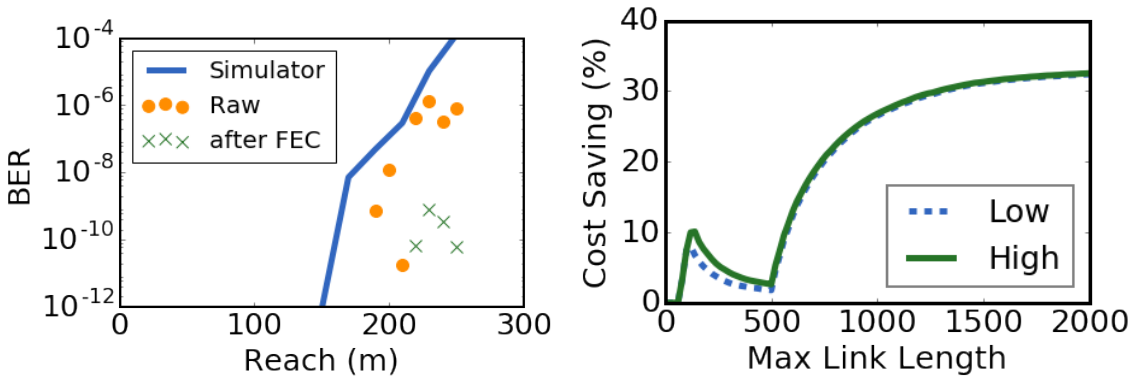
OVER-ENGINEERING IN 100 GBPS IN RAIL

We repeat our stretch experiments using 100G-SR4 transceivers [1] (standard reach 70m) and observe similar levels of over-engineering. However, an important distinction between 100 Gbps and 10/40 Gbps technologies is the presence of a Forward Error Correction (FEC) module in 100 Gbps switches. This means the pre-FEC BER requirement is reduced from 10^{-12} to 5×10^{-5} and the FEC module boosts BER back to the standard on every hop.

Figure D.1a shows pre- and post- FEC BER as we stretch the fiber length without additional attenuation. Interestingly, even pre-FEC BER (Raw) is lower than 10^{-12} at 180m; this is 2.5 times higher than the standard reach, once again confirming the degree of over-engineering. We simulate this transceiver in VPI and conservatively bound the stretch to 110m and 130m to achieve *network reliability bounds* of 99% and 95% respectively where FEC is off.

From simulation in VPI, we show PSM4 technology can be stretched from 500m to 2km. Figure D.1b shows the cost saving for uniform random link length distribution. Before 70m, there is no cost saving because all links are covered by 100G-SR4. The cost saving peaks at 150m, the stretched reach, because the largest fraction of the link can use 100G-SR4. Cost saving decreases at this point, because 500m PSM4 technology's cost is cheap. Cost saving increases after 500m because PSM4 technologies can replace higher cost 100G-LR4 technologies between 500-2km. Overall, we find the savings in 100G can be up to 30%.

When FEC is turned on, the link quality distribution among all the links is narrower. The amount of over-engineering remains the same, which means stretching transceivers



(a) Reach Test

(b) Cost Saving

Figure D.1: Figure D.1a shows the reach test for 100G-SR4. Blue line is simulated transceiver from VPI. Yellow dots are raw BER and green dots are corrected BER. Figure D.1b shows cost saving for uniform random link length distribution for different max length.

can still reduce the cost of data center networks. However, switching gray links off is likely enough to protect the network. We leave this problem for future investigation.