UNIVERSITY OF CALIFORNIA SAN DIEGO

# Simplifying datacenter fault detection and localization

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

in

Computer Science

by

Arjun Roy

Committee in charge:

Alex C. Snoeren, Co-Chair
Ken Yocum, Co-Chair
George Papen
George Porter
Stefan Savage
Geoff Voelker

2018

The Dissertation of Arjun Roy is approved and is acceptable in quality and form for publication on microfilm and electronically:

---

---

---

---

---

Co-Chair

---

Co-Chair

University of California San Diego

2018

# DEDICATION

Dedicated to my grandmother, Bela Sarkar.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to acknowledge Alex C. Snoeren for all of his support and guidance over the years. In addition, I would like to acknowledge Ken Yocum for his important role during the start of my doctoral journey. Furthermore, I would like to acknowledge the role played by my thesis committee. I would like to thank Geoff Voelker, George Papen, George Porter and Stefan Savage for their roles either as co-authors, teachers, or (on rare occasion) poets over the years.

Over the years, I have had the chance to collaborate with several talented researchers and engineers within industry on various projects that appear within this dissertation. While listed in full below, I would like to express my thanks and gratitude to James Zeng, Jasmeet Bagga and Omar Baldonado of Facebook, as well as the entire Facebook NetSystems team; and Deepak Bansal, David Brumley, Harish Kumar Chandrappa, Parag Sharma, Rishabh Tewari and Behnaz Arzani of Microsoft, as well as the entire VNET and Datapath teams at Microsoft.

Significant acknowledgment goes to my family, including my mother, Aditi Roy, grandmother, Bela Sarkar, father, Parthasarathi Roy, and sister, Rhea Roy, for being supportive and understanding throughout this process. Acknowledgments are also due to my friends and colleagues, at UC San Diego and elsewhere, for making the academic process fun and enriching. Kudos are due to Antonio Gellineau, Gabriel Lefkowitz and Prish Dunstan in particular for having seen this journey through from day one. Special acknowledgment goes to Carla Sun for being by my side through the highs and lows, and for putting up with my foibles in general.

Chapter 4 is an adapted reprint of the material as it appears in "Inside the Social Network's Datacenter Network", which was published in ACM SIGCOMM 2015. The dissertation author was the primary author of this paper. The paper was co-authored with Hongyi (James) Zeng and Jasmeet Bagga of Facebook, and George Porter and Alex C. Snoeren of UC San Diego. The original work was supported in part by the National Science Foundation through grants CNS-1314921 and CSR-1018808. We are indebted to Theo Benson, Nick McKeown, Remzi Arpaci-Dusseau, our shepherd, Srikanth Kandula, and the anonymous reviewers for their comments and suggestions on earlier drafts of the originally published manuscript. Petr Lapukhov, Michal

Burger, Sathya Narayanan, Avery Ching and Vincent Liu provided invaluable insight into the inner workings of various Facebook services. Finally, and most significantly, Omar Baldonado catalyzed and facilitated the collaboration that enabled the publication of the original manuscript.

Chapter 5, not including Section 5.6, is an adapted reprint of the material as it appears in "Partial datacenter fault detection and localization", which was published in USENIX NSDI 2017. The dissertation author was the primary author of this paper. The paper was co-authored with Hongyi (James) Zeng and Jasmeet Bagga of Facebook, and Alex C. Snoeren of UC San Diego. The original work as published in USENIX NSDI 2017 was supported in part by the National Science Foundation through grants CNS-1422240 and CNS-1564185. We are indebted to Aaron Schulman, Ariana Mirian, Danny Huang, Sunjay Cauligi, our shepherd, Dave Maltz, and the anonymous reviewers for their feedback on the originally published manuscript. Alex Eckert, Alexei Starovoitov, Daniel Neiter, Hany Morsy, Jimmy Williams, Nick Davies, Petr Lapukhov and Yuliy Pisetsky provided invaluable insight into the inner workings of Facebook services. Finally, we thank Omar Baldonado for his continuing support.

Section 5.6 contains material currently being prepared for submission to IEEE Transactions on Networking. The dissertation author was primary author for this material. The work is co-authored with Hongyi (James) Zeng and Jasmeet Bagga of Facebook, and Rajdeep Das and Alex C. Snoeren of UC San Diego.

Chapter 6 is an adapted reprint of material currently being prepared for submission to ACM IMC 2018. The dissertation author was primary author for this material. The work is co-authored with Deepak Bansal, David Brumley, Harish Kumar Chandrappa, Parag Sharma, Rishabh Tewari and Behnaz Arzani of Microsoft, and Alex C. Snoeren of UC San Diego. Nimish Aggarwal, Milan Dasgupta, Abhishek Ellore Sreenath, Alex Fu, Daniel Firestone, Shefali Garg, Deven Jagasia, Abhishek Kumar, Neeraj Motwani, Jae Park, Chaitanya Raje, Pranjal Shrivastava, Abhishek Shukla, Qaseem Zuhair, Michal Zygmunt, and the entire VNET and VFP teams at Microsoft provided valuable insight into Azure Networking.

# VITA

| | |
|---|---|
| 2009 | Bachelor of Science in Computer Science, Columbia University in the City of New York, New York, NY. |
| 2011 | Masters of Science in Computer Science, Stanford University, Stanford, CA. |
| 2011–2018 | Graduate Research Assistant, University of California, San Diego |
| 2017 | Teaching Assistant, Department of Computer Science and Engineering, University of California, San Diego |
| 2018 | Doctor of Philosophy, University of California, San Diego |

ABSTRACT OF THE DISSERTATION

Simplifying datacenter fault detection and localization

by

Arjun Roy

Doctor of Philosophy in Computer Science

University of California San Diego, 2018

Alex C. Snoeren, Co-Chair
Ken Yocum, Co-Chair

The proliferation of distributed internet services has reaffirmed the need for reliable and high-performance networks, not only in the WAN bringing users to the services, but within the datacenters where services themselves reside. Services consist of distributed applications running across thousands of servers within datacenters, with stringent performance, scaling and reliability requirements. To support these requirements, datacenter networks are comprised of thousands of servers, links and ports and over hundreds of switches providing multiple paths between any pair of servers. Because any given component has a small but non-zero failure rate, the large number of components means that failures are endemic inside datacenters. Unfortunately, not all failures are easily diagnosable within datacenter environments.

In particular, datacenters are susceptible to insidious parasitic performance loss due to a class of network component fault known as partial faults—where a component is nominally healthy, but intermittently drops or delays traffic. These faults have been noted as being particularly difficult to detect and localize, though mitigation can be straightforward once the faulty component is determined. Pinpointing partial faults quickly is crucial, because they are capable of inflicting a disproportionately high toll on application performance.

Unfortunately, partial faults can confound existing fault detection methods in several ways, including interactions between faults, application traffic, and network hardware. For example, network switches may fail to detect faults due to unreliable or otherwise insensitive monitoring capabilities. Traffic volume and variability may complicate analyzing server-based application and network metrics, as well as mask fault impact. Moreover, the myriad paths available to network flows complicate localization even if servers detect partial faults.

However, this work shows that the scale and regular design of contemporary datacenters can simplify partial-fault localization. In particular, the combination of large-scale load-balanced multipath topologies and high-volume datacenter traffic enables simple, low-overhead, application-agnostic, and root-cause-agnostic partial-fault localization via passive, link-by-link outlier analysis of application network performance. I validate the effectiveness of my approach within large-scale first-party production datacenters, and examine the additional challenges and complexities raised by third-party cloud datacenters.

# Chapter 1

# Introduction

Widespread and ever-growing internet access has continued to drive the proliferation of large-scale services aimed at improving the lives of people worldwide. For example, services may provide web-based information indexing and retrieval, online storefronts, and social networking platforms to upwards of billions of users [43, 44]. Due to scale and reliability requirements, service operators often implement their services as distributed applications that run on vast numbers of networked servers. As these services have grown in scale and capability, the number of servers required to support them, and their network demands [19], have grown as well.

The physical installation housing such servers, the servers themselves, and the network connecting them together are collectively referred to as a 'datacenter'. Modern datacenters continue to increase in scale, speed, and complexity. As these massive computing infrastructures expand—to hundreds of thousands of multi-core servers with 10- and 40-Gbps NICs [112] and beyond—so too do the sets of applications they support: Google recently disclosed that their datacenter network fabrics support literally thousands of distinct applications and services [117].

Oftentimes, datacenter operators do not control the applications running therein. So-called 'tenant' datacenters instead provide rentable computing infrastructure to third-party customers and organizations. In this case, the datacenter network operator owns and maintains the infrastructure, while tenants rent computing resources to host their own services and applications.

Yet the practicality of operating such multi-purpose datacenters depends on effective management. While any given service might employ an army of support engineers to ensure its efficient operation, these efforts can be frustrated by the inevitable failures that arise within the network fabric itself. While fault detection is a classical problem in distributed systems [13,25,39, 40, 55, 110] and networks [26, 76, 99], modern datacenter environments and large-scale services give rise to a confluence of challenges that requires re-examining fault detection in this setting. In particular, large-scale services have a plethora of highly latency-sensitive [83] users [43, 44]. Applications implementing large-scale services require both low-latency and high-bandwidth connectivity [112, 117], leading to high path-diversity network topologies [14, 19, 117].

Experience indicates that modern datacenters are rife with hardware and software failures, due to the use of commodity hardware (exhibiting low but non-zero failure rates) coupled with the misconfigurations, software anomalies and hardware failures attendant with large-scale deployments [126]. To provide acceptable performance to applications despite failures, datacenter designs leverage redundancy in a bid to be robust to large numbers of faults. For example, after a failure manifests, network operators may pinpoint the affected component and route traffic around it using excess capacity to avoid application performance degradation [93, 126]. However, this approach depends on effective fault localization, which large-scale deployment can complicate. Moreover, recent work has indicated that the types and impacts of faults common in modern datacenters [126, 133] differ from those typically encountered in the wide-area [111, 122] and enterprise [121]; we focus on one particularly insidious class of such faults in this work.

Specifically, Microsoft studies describe [64, 133] a rogue's gallery of datacenter faults: dusty fiber-optic connectors corrupting packets, switch software bugs, hardware faults, incorrect ECMP load balancing, untrustworthy counters, and more. These failures can be intermittent and partial: rather than failing completely, a link or switch might only affect subsets of traffic, thus complicating detection and diagnosis. Despite being intermittent, these 'partial faults' can still significantly impact application performance, while the difficulty of partial-fault localization may exacerbate impact length. For example, the authors of NetPilot [126] describe how a single

link dropping a small percentage of packets, combined with cut-through routing, resulted in degraded application performance and a multiple-hour goose chase to identify the faulty device.

Network virtualization further complicates matters within many third-party tenant datacenters [54]. Servers in such datacenters often take on additional network management responsibilities—including routing, packet processing and network address indirection—leading to scenarios where server-based performance issues can manifest as network-based performance issues from the point of view of tenant applications. In other words, a network-impacting partial fault in such a datacenter could be due to a network switch or link, or a server issue.

To make matters worse, both partial faults and common datacenter traffic characteristics can cause packet loss and latency spikes. Traffic may be bursty [131], variable [30] and potentially unpredictable at micro [112] and macro [32] timescales. The extent of these effects have motivated datacenter operators and network protocol designers to expend significant effort on application and network level traffic engineering [18, 34, 112, 117] to stave off harmful effects such as flash congestion [129] and latency-inducing persistent queue buildups [18].

All of these effects can complicate partial-fault detection (for example, bursty traffic can mask fault-driven packet loss and latency), application impact analysis (due to the sheer traffic volume and variability) and fault localization (path-diversity means that faults may impact any one or more of *thousands* of links [113]). At the same time, end-user desire for low end-to-end service latency [83] raises the stakes and harshens the consequences of undiagnosed faults.

Production methods for pinpointing datacenter network faults typically involve watching for anomalous network events (for example, scanning switch drop and link utility/error counters) and/or monitoring server performance metrics. Such methods consider events independently: "Did a drop happen at this link? Is RPC latency unusually high at this server?" Yet, in isolation, knowledge of these events is of limited utility. There are many reasons servers could observe poor network performance; similarly, transient congestion may cause in-network packet loss as well as persistent faults. Hence, datacenter operators frequently fall back to active probing and a degree of manual analysis to diagnose and localize detected performance anomalies [64, 133].

Instead, we propose an alternative approach: rather than looking at anomalies independently, we consider the impacts of faults on aggregate application performance. Modern datacenter topologies are typically designed with a plethora of disjoint but equally performant paths and operators work hard to load balance traffic across both paths and servers [19, 34, 112, 117]. When combined with the large volume of datacenter traffic, such designs result in highly regular flow performance regardless of path—in the absence of network faults [112]. An (un-mitigated) fault, on the other hand, will manifest itself as a performance anomaly visible to end hosts. Hence, to detect faults, we can compare performance end hosts observe along different paths and hypothesize that outliers correspond to faults within the network.

Thus, we claim that various fundamental datacenter characteristics, borne of the underlying design goals of contemporary datacenters, enable a partial-fault localization methodology based on hypothesis testing and outlier analysis. In particular, the combination of large-scale load-balanced multipath topologies and high-volume datacenter traffic enables simple, low-overhead, application-agnostic, and root-cause-agnostic partial-fault localization via passive link-by-link outlier analysis of application network performance.

This dissertation investigates the current state of the art in datacenter fault management, focusing on partial-fault detection and localization. First, it highlights how partial faults can cause disproportionately severe performance degradations within common application workloads, and describes why pinpointing them is difficult despite impact severity. It then demonstrates how multipath datacenter topologies and high traffic volume—which ostensibly complicate fault localization—can be harnessed to simplify pinpointing networking-component-based partial faults. We evaluate a prototype partial-fault localization system within Facebook datacenters under a variety of production application workloads and fault scenarios, including instances where our system notices faults before production monitoring infrastructure [113]. While existing research examines partial faults that occur within the physical network plane (i.e. within the switches and links that comprise the network), a relative dearth of information is present on datacenter application behaviour and faults that occur within network virtualization overlays

inside tenant datacenters. Thus, this dissertation provides a first look into tenant SDN-virtualized datacenter monitoring, focusing especially on partial-fault behaviors specific to such networks. In all, this dissertation makes the following specific contributions:

1. Based on the topologies and application traffic patterns of contemporary datacenters, it presents as the primary contribution a methodology for finding network-component-based partial faults leveraging statistical outlier detection. It evaluates a prototype system based on this methodology within a large-scale production Facebook datacenter environment.
2. It presents the results of a detailed application traffic pattern study at Facebook, a datacenter operator providing a popular social networking service used worldwide. It highlights relevant traffic characteristics both in support of this work's proposed fault localization system, and of broader interest to networking research. In particular, it upends various assumptions made regarding datacenter traffic based on prior studies.
3. It examines the additional kinds of partial faults that can occur within tenant datacenters that use server-based tenant-network-isolation techniques—above and beyond the network-component-based partial faults that continue to occur within such datacenters.

This dissertation is organized as follows. Chapter 2 scopes the problem we try to solve and provides context on datacenter topologies, traffic patterns, and network faults. Chapter 3 examines contemporary production and academic fault localization techniques and discusses the efficacy and shortcomings of these approaches before presenting the intuition for this work's solution. Chapter 4 presents a detailed study of Facebook datacenter application traffic, focusing both on implications for fault localization and topics of broader interest to networking research. Chapter 5 leverages these characteristics and presents the design, implementation and evaluation of a prototype partial-fault localization system that represents this work's primary contribution. Following this, Chapter 6 discusses additional challenges impacting partial-fault localization within virtualized tenant datacenters. Chapter 7 concludes by motivating future work, focusing especially on fault diagnosis and mitigation.

# Chapter 2

# Datacenters, applications, and failures

To effectively appreciate the importance of datacenter fault detection, and to properly evaluate the effectiveness of any proposed solution, it is important to develop sufficient context. To begin with, we circumscribe the scope of this dissertation's main contribution. Prior work suggests, for partial faults affecting switches and links, that fault diagnosis and mitigation are relatively simple problems to tackle after fault localization [126]. Simply put, links and switches causing loss or delay can be disabled or routed around, thus preventing application traffic from incurring performance penalties. In various cases, simply rebooting a network switch is sufficient to resolve faulty behaviour [126]. For virtualized tenant networks, however, this dissertation finds that diagnosis and mitigation can be significantly more complex, as discussed in Chapter 6.

Thus, this work concerns itself primarily with detecting and localizing partial faults within network hardware, focusing primarily on non-tenant, non-virtualized 'first-party' datacenters. In this chapter, we first discuss datacenter applications, network design, network faults, and fault impact on applications. We then argue in favour of passively monitoring application network performance for partial-fault localization, rather than actively probing the network or instrumenting it to detect specific failures. Following this, we discuss the metrics we may use for passive monitoring, categorized broadly into whether they are collected at network switches or servers. In particular, we discuss how partial-fault localization leveraging only network-switch-based metrics, or only server-based metrics, can underperform due to the challenges imposed

by partial faults in conjunction with multipath topologies and application workloads. Instead, we propose a unified approach leveraging information from both network switches and servers that overcome these challenges, discussing datacenter characteristics that help us do so. We then discuss related work and motivate our solution in Chapter 3.

## 2.1 Datacenter applications, networks and faults

In this section, we discuss the needs of common datacenter applications and how they drive datacenter network design. We then focus on how large networks, built out of commodity hardware with a low but non-zero failure rate, result in a non-trivial rate of performance-sapping 'partial faults'. Following this, we motivate the importance of rapidly detecting and localizing partial faults by describing their impact on datacenter application performance.

### 2.1.1 Datacenter application patterns

Datacenter applications can present different and possibly conflicting requirements to the underlying network. For example, user-facing services require low latency to maximize user engagement, while batch-processing 'big-data' workloads require abundant bandwidth and are often less sensitive to network latency. Real-world services may require a mix of such traffic; consider a web search service that needs to manage a large corpus of data within its search index, while rapidly responding to user queries from the internet [117]. Note that this is not a complete list of datacenter applications; for example, video-delivery services are user facing but depend heavily on efficient utilization of bandwidth [12]. However, a combination of big-data and latency-sensitive user facing workloads exist at a variety of large-scale datacenter operators, including Google, Microsoft and Facebook [77, 112, 117]. We focus on such traffic in this work.

#### Big-data workloads

Datacenters commonly support large-scale computations that cannot fit within a small ($\leq$ the server capacity of a ToR switch) number of servers. For example, these can include indexing

webpage content in support of web searches, or computing large-scale analytics to recommend products to online shoppers. Specifically, the volume of data processed may not fit within either the primary or secondary memories of any single or small set of servers. This dissertation describes this class of computations as 'Big-data workloads'. Big-data workloads use divide-and-conquer approaches that split the input to a computational problem across a large number of servers across the datacenter; each server then operates on this data independently, with a recombination step providing a single unified output for the overall computation [46]. While various proposals have described alternative computational models [70, 128], the fundamental divide-and-conquer strategy and attendant bandwidth requirements do not change.

Due to large data volume, big-data workloads frequently utilize many-to-many and all-to-all traffic patterns, thus driving the development of high-bandwidth networks that can support such patterns [14, 62, 63, 117]. Maximizing network and server utilization receives significant attention; various proposals have targeted both more efficient application-layer job scheduling and better network utilization [127]. Strategies include optimally using available bandwidth [68, 71], avoiding bandwidth-wasting traffic imbalances [15], and promoting network topologies that are resilient to (or can otherwise deal with) inevitable network failures [93, 118]. Various big-data processing systems exploit data locality when able [46] to avoid needless network-core congestion. When using the network core is unavoidable, effective congestion control is critical [18, 117]. All of this engineering effort informs us that network performance is of great importance to big-data workloads; as we will see, effectively dealing with datacenter faults is a necessity for maintaining high performance.

**Latency sensitive workloads**

User-facing services handle requests from the outside world, and possess significant differences compared to big-data style workloads. With the notable exception of services like video streaming, user requests frequently have a short lifespan compared to big-data workloads. For example, a user might wish to search for a particular item and receive immediate results

when attempting to use an online storefront; indeed, delays of greater than 100 milliseconds may discourage user interaction with such services [83]. Thus, a focus on latency has driven or influenced a significant chunk of application, topology and hardware design [35, 82, 125, 129].

Unlike big-data workloads, where organizations can schedule tasks and optimize network utilization, user-driven traffic is inherently randomized. Various statistical methods can analyze aggregate user behaviour [28, 90]; however, user-driven traffic can still induce unpredictability in network traffic patterns in various ways. For example, certain pieces of content can become virally popular and cause sudden traffic spikes from users requesting that content; this could lead to hot spots in the underlying datacenter network, at least until the content can be replicated and load-balanced to account for its popularity [34]. Furthermore, request desynchronization, inherent to large and uncoordinated user bases, can lead to unpredictably bursty traffic in the network which can cause loss inside shallow-buffered datacenter switches [131].

A typical high level architecture for interactive user services is as follows. Users make requests, which arrive at datacenter load balancers. Load balancers split user requests amongst a tier of stateless application servers. The data backing the service resides within a tier of databases; application servers are responsible for querying databases for the appropriate data to satisfy user requests. Once the necessary data is assembled, application servers respond to the user. As a common optimization for services with read-mostly requests, popular data may be stored in read-optimized cache servers which may be queried instead of the databases.

One challenge such an architecture presents is that any given user-level request is likely to trigger several (tens of, or more) network flows to satisfy the request [30, 34, 112]. The overall runtime of the high-level request depends on the runtime of all of the flows dispatched to service it. Thus, if any one of these flows is delayed, the overall runtime of the user-level request increases. Some user requests may be amenable to a trade-off between request latency and degraded result—for example, a search query may only yield the subset of search results that could be returned within a fixed window of time, if latency mattered more than result quality—but services in general may not necessarily support or welcome such trade-offs [129]. To provide low

network latency for these applications, datacenter operators once again leverage high-bandwidth network topologies. Unlike big-data workloads, link utilization may be low [112]; the high network fabric capacity is instead used to ensure there is sufficient headroom for bursty traffic so that loss (and the consequent latency spikes) rarely occurs.

### 2.1.2 Datacenter networks

The growth in the demands placed on internet-facing services, the specific requirements of datacenter application patterns, and the desire to reduce costs have served as key motivators for datacenter topology research and engineering efforts [63, 117]. In particular, both big-data and latency-sensitive workloads have driven a desire for datacenters of ever-increasing server capacity and network bandwidth. Balancing scale and cost concerns has driven both equipment and topology designs in ways that influence both the overall reliability of datacenter networks, and the difficulty of finding faults when they do occur.

Historically, large datacenter operators like Google and Facebook have used so-called 'four-post' topologies, where a cluster consists of multiple racks of a few tens of servers each, interconnected via four high-radix cluster switches repurposed from Wide Area Network (WAN) environments [19, 117]. While the switches themselves were considered of high quality, and while it is expedient to reuse hardware that is already developed and validated, attitudes towards switch (and thus network) design have evolved in a few key ways.

#### Commodity switching hardware

Due to stringent reliability and interoperability needs, WAN switch design prioritizes high availability and protocol diversity, thus increasing costs [117]. Contrastingly, datacenters often use cheaper commodity switches, indicating a trade-off between cost and complexity for slightly reduced reliability [117]. When aggregating this trade-off over a large datacenter, faults are inevitable. Perhaps as a consequence of using lower-availability hardware, datacenter operators have noted that network fabric redundancy degrades as hardware ages; in some cases,

unforeseen failure modes develop [117]. Thus, the combination of large datacenter networks and mostly-reliable commodity hardware exacerbates fault incidence rates. However, careful topology design retains high network availability and performance even with failures [93]; traffic can traverse alternate paths and use excess network capacity to maintain performance—if network monitoring systems can rapidly detect and localize failures.

**Scaling via multipath topologies**

Consider a simplified logical model of a datacenter as a single non-blocking switch; adding more servers requires both larger port counts and greater switch fabric capacity to provide increased aggregate bandwidth. Google four-post datacenters used the highest density switches available at the time, which imposed a natural limit on cluster size [117]. Furthermore, there was a tension between the number of servers such a cluster could support, and the amount of external bandwidth such a cluster had for connecting it to other clusters [19]. Both external and internal bandwidth are crucial, since a single cluster is a relatively small component within large-scale service infrastructure that must communicate with other clusters [112].

Beyond providing connectivity, datacenters must provide high bandwidth. Oversubscribed four-post clusters imposed constraints on application traffic that were categorically the network's fault—a computational task might have had enough computing resources spread through the network but not enough bandwidth to connect them [117]. Researchers have noted that bandwidth constraints complicated application design and limited overall performance [14].

Thus, datacenters require high port counts and (internal and external) bandwidth. Rather than developing higher-radix switches, scalability is achieved by interconnecting larger numbers of smaller commodity switches. Scalability thus depends on careful topology design [19, 117]. High bandwidth requirements, in support of Map-Reduce [46] style big-data workloads, motivated topology design in several cases [14, 62, 63, 117]. In each case, large-scale multipath provides scalability. Multipath aids reliability as well; *if* a network fault is pinpointed, excess network fabric capacity can absorb the load normally routed over the faulty components.

Contemporary datacenter networks are thus organized as multi-rooted trees offering redundant equal-cost paths between datacenter servers [14, 19, 117]. Relatively involatile topologies and centralized control simplify datacenter routing, while switch ECMP groups allow traffic to both leverage all network paths in the absence of faults and avoid particular hops if a fault is detected. Switch control software is, in some cases, offloaded from relatively under-provisioned switch CPUs to servers [117]. However, large-scale multipath can complicate fault localization; even if applications detect degraded network performance, pinpointing the responsible link or switch may be challenging, as we shall see later in this chapter.

**Switch buffering and congestion**

Switch buffering impacts the degree of attainable network performance. High bandwidth flows on high latency WANs depend on sufficiently large buffer sizes to effectively use all available bandwidth. For a single flow, the 'Bandwidth-Delay Product' (BDP) rule of thumb describes the necessary buffer sizes for high utilization [21]. Low bandwidth flows may still be bursty and require sufficiently high buffer headroom to avoid packet loss [112].

However, the specific degree of buffer sizing remains a contentious issue. Research targeting WAN environments with high degrees of statistical flow multiplexing has suggested that the BDP is unnecessary, and that higher degrees of multiplexing can allow switches to provision smaller buffers [21]. Different traffic studies, focusing on different datacenter operators, reveal that one cannot take for granted the degree of statistical multiplexing present within datacenter links—depending on workload, high [112] or low [18] levels of multiplexing may occur. Consequently, certain arguments favour deep switch buffers [29] for datacenters while others do not [36, 56, 98]. Furthermore, a consequence of ever-increasing link speeds is that large buffers may be unfeasible or undesirable for cost and performance reasons [98].

Thus, contemporary commodity datacenter switches tend towards shallower buffers [112, 117], with careful server-based control of network congestion to manage queue sizes [117]. Even so, the combination of ever-increasing datacenter link speeds, bursty application traffic [112],

and shallow switch buffers means that effects such as flash congestion [129], incast [18], and microbursts [131] are endemic within datacenters. We argue that these effects greatly complicate partial-fault detection and localization. Both faults and congestion-based effects can cause packet loss and delay; thus it can be difficult to reason whether an application performance degradation is due to congestion resulting from shallow-buffered switches or an actual fault.

**Centralized control**

Traditionally, WANs comprised of autonomous and independent subdivisions have leveraged complex distributed routing and directory service protocols [94, 101, 103]. While commodity datacenter switches may not support all the myriad protocols that a WAN switch might, datacenters may still allow fine-grained centralized control since a single operator controls the entire network. Thus, they are amenable to centralized control schemes that allow the entire network to work in concert. Various large datacenter operators today employ a control scheme known as Software Defined Networking (SDN) [33, 54, 117]. Switching hardware is comprised of a forwarding plane (hardware ASICs comprising the switching fabric of a network switch) and a control plane (which configures the forwarding plane). While a WAN switch might have a switch-local control plane running a distributed routing protocol that configures hardware routing tables, an SDN controlled switch may offload the decision making process to one or more logically centralized controllers which calculate forwarding behaviour and program the forwarding hardware appropriately. In addition to controlling physical network switches [117], SDN has been used to control server-based 'virtual switches' which are used to provide network connectivity to virtual machines (VMs) inside tenant datacenters [54].

SDN networks enable a potentially rich set of behaviours and configurations for network components—for example, bandwidth allocation policies [68, 71] may leverage switch queuing and limiter resources, or servers may perform network load balancing [67]. However, greater complexity can increase the incidence of errors or unexpected behaviour. Improper network configuration may lead to long term link utilization imbalances or incorrect queuing and quality

of service, thus impacting application performance. Managing rulebase and hardware configuration complexities within SDN networks is an active field of research, with various proposed methodologies [81, 102]. Even if networking state is properly computed and managed, however, inaccuracies may occur when programming network hardware with this state [133] or due to inconsistencies borne of the distributed control plane [80, 89]. Thus, it is important to be able to characterize traffic performance in relation to traversed network components; if we can localize network components that coincide with diminished traffic performance, we may receive early (or additional) warning of potential misconfigurations or faults.

### 2.1.3 Datacenter partial faults

Contemporary datacenter scale means that faults are inevitable, and topology design and routing protocols must account for them [62, 63, 93]. Links or switches might fail to forward traffic; in such a case, traffic must flow around damage and fallback onto alternate network paths. That said, the efficacy of failure mitigation depends on the ability to realize a failure has occurred, and where in the network it resides. In some cases, this may not be obvious.

As a motivating example, Microsoft datacenter operators discovered dusty optical cable connectors inducing packet corruption and loss [126]. While application traffic felt loss-driven impact, fault localization was complicated since cut-through routing meant packet corruption was detected at devices other than the one actually responsible for packet corruption. This failure represents an insidious class of network fault—where the network appears healthy (no links or switches have failed entirely and still transmit traffic) but a subset of traffic is lost or delayed. The fault mechanism may limit the network's ability to accurately self-report the fault's location. These faults may cause disproportionately severe application impacts even with small loss rates. We call these faults "partial faults"—where the state of a networking component is nominally healthy, but a subset of traffic is either lost or delayed. Experience shows that partial faults can occur for myriad reasons; we investigate a few examples next.

**Partial fault taxonomy**

For example, consider 'black-holes', where routing misconfigurations can cause packet loss due to a router having no forwarding rule for certain traffic. Microsoft datacenter operators have noted occasions where a switch may inadvertently create a black-hole due to bitflips in TCAM-based routing tables [133]. TCAM bitflips cause faults where the network configuration is nominally correct (when queried, the switch control plane will insist that routes are programmed as intended) and yet a subset of traffic that would hit the rule is dropped. Furthermore, in certain cases packets may be dropped silently, where neither drop nor error counters increment [133].

Another example arises from inexplicable ECMP imbalances caused by seemingly malfunctioning switches [126]. In this scenario, traffic is unevenly spread across links in an ECMP group—here, no drop counters or error counters may trigger, but latency may be impacted for traffic traversing higher utilization links. Switch reboots may fix the problem [64, 126], without revealing whether buggy switch hardware [64] or software control planes [113, 117] were at fault. Moreover, imbalance detection may be complicated since since it can be hard to distinguish faulty ECMP behaviour from naturally uneven traffic caused by elephant flows [15], especially in datacenters with relatively low statistical flow multiplexing [18].

Note that partial faults are not necessarily caused by hardware-centric anomalies; instead, they may also arise from hardware misconfiguration [64, 117]. While there have been a variety of efforts towards simplifying datacenter configuration and providing adequate manageability of the networking rulebase [81, 102], inevitable oversights in software engineering [117] can confound these efforts and torpedo the efficacy of network devices.

It is important to note that while there are already many different ways partial faults may occur, the opportunities for such faults to occur continue to increase as datacenter networks grow in size and complexity. Indeed, a significant chunk of network switch errors have causes that are never diagnosed [126] at all. The advent of network virtualization and software-defined networking based overlays within tenant datacenters provides additional opportunities for partial-

fault style behaviour to occur (we discuss these effects more in Chapter 6). Next, we discuss the impact partial faults have on application traffic.

**Partial faults and application impact**

While latency-sensitive and big-data workloads workloads differ along several axes, they do share one commonality: both utilize scatter-gather workloads. Despite different operation timescales, high-level workload execution time in either case is gated by the performance of the slowest sub-operation involved. Scatter-gather style workloads are thus highly susceptible to tail-latency increases, leading significant amounts of research into improving tail performance [18, 125, 129]. To illustrate the impact of tail latency, consider a simple scenario where a high-level request forks 10 parallel sub-requests. Suppose any given sub-request has a 1% chance to be delayed to an inacceptable degree, and a 99% chance of completing within an acceptably short period of time. The probability that a high level request will be slowed down is $1 - 0.99^{10}$ in this scenario—a full order of magnitude more likely than the probability that any given sub-request is slowed down. Such scenarios are an important design consideration for large datacenter operators providing interactive services where low latency matters [18, 129].

The authors of DeTail, a tail latency mitigating methodology, assert that the typical culprit for high tail latency is 'flash congestion' in the network, which causes packet loss and attendant delay [129]. They note further that uneven load balancing exacerbates the issue, causing more opportunities for congestion to occur. However, in addition to regular congestion, the presence of partial faults can also increase tail latency—and unlike temporary congestion, partial faults can persist within the network core for a relatively long period of time before being mitigated [126]. Partial faults within datacenter networks can thus be heavily detrimental to application performance. A fault inducing a small packet loss rate or added delay can have a disproportionate impact on scatter-gather workloads.

While big-data workloads network flows may be longer than latency-sensitive user traffic [112], a similar effect can occur. While bulk-flows may be latency-insensitive (microsecond

respond times may not matter for a flow lasting multiple seconds), packet loss can slow down reliable protocol transfer rates (e.g., TCP considers loss as a sign of congestion), leading to slow transfers and higher overall job execution times than in the absence of faults. MapReduce jobs may be delayed if even a single worker stalls due to slow network transfers. Loss affecting some paths but not others can lead to flows on fully functional paths outcompeting flows mapped to paths with at least one faulty link when they share a common bottleneck link or switch buffer.

## 2.2 Partial faults require passive impact monitoring

The diverse and unpredictable nature of partial faults indicates that building proactive monitoring systems to search for specific failure modes is inevitably futile [117]. Furthermore, diagnosing specific failures may be unnecessary if simply rebooting affected hardware can clear faults [126]. Instead, deleterious partial-fault-driven traffic impacts suggest that one appropriate methodology may be to search for such impacts and use them to localize faults. For example, we may monitor synthetic probe traffic [11, 64] or production application traffic for loss or latency indicative of a partial fault. Once we localize a fault, we may mitigate its impacts (perhaps by disabling faulty components) before remedial actions are applied [126].

However, we contend that monitoring synthetic probe traffic is not a panacea, due to the possibility of partial faults that may only impact specific traffic subsets [133]. Furthermore, active-probing monitoring systems may possess resource-consumption constraints that limit allowable network overheads, as we discuss further in Chapter 6. Thus, since probe traffic volume is constrained, and since it may not trigger latent fault mechanisms, it may delay partial-fault localization. On the other hand, passively monitoring production-traffic performance impact alleviates both concerns; this dissertation devises a system that passively examines *all* production traffic and rapidly pinpoints partial faults affecting small subsets of this traffic.

If we are to rely on passively examining application network performance to detect and localize partial faults, we have to decide what metrics we will observe. Our options include

network-device metrics (for example, switch-based drop, error or queue occupancy counters), server metrics (for example, TCP statistics or application-level metrics like end-to-end request latency) or some combination of both. We refer to these options as adopting particular viewpoints, depending on which metrics we observe—thus, we can adopt either a network-centric, server-centric, or unified viewpoint. While both network-centric and server-centric viewpoints can give indications of application network performance, it is the position of this dissertation that examining either in isolation has significant drawbacks, which we discuss next. Instead, we propose a unified approach which we motivate in Section 2.3.

### 2.2.1 Multipath hampers server-centric monitoring

Servers possess deep insight into application level performance. In particular, we can characterize whether metrics like request latency, flow completion and file transfer times meet various service-level agreements that dictate acceptable service-performance bounds. Furthermore, we can examine transport-level statistics through server network stacks; for example, TCP retransmits can convey the degree of packet loss. A fundamental issue, however, is that while we may be able to detect performance degradation at servers, we cannot ascertain the root cause. As an example, a network-based partial fault may cause high application-level request latencies; on the other hand, remote-server scheduling or disk-access latency may also delay requests. Similarly, a network-based partial fault may cause packet loss and trigger TCP retransmits; alternatively, server-based effects like insufficient TCP buffer space may do the same.

Theoretically, network operators may systematically examine metric reactions to various controlled stimuli (specifically, injected faults of differing mechanisms at network components or servers) and develop a baseline per-metric expected-behaviour model; they can then use machine-learning classifiers to determine whether network-based faults cause production-observed loss or delays [23]. In addition to requiring model-generation and complicated analysis, this methodology is incapable of fault localization (if performance degradation is deemed a network-caused issue) due the lack of server-based knowledge about impacted-traffic network path.

For production datacenters, multihop and multipath topologies yield a combinatorial explosion of paths between communicating server pairs [113], which serves both as a boon and a curse. Path-diversity both increases fault likelihood and complicates fault localization; however, it also provides alternative paths that (hopefully) do not contain faults. Fallback-path availability supports one possible argument: that in fact, it does not matter if servers can localize faults or not. Rather, servers can simply attempt to influence application traffic path if they detect performance degradation. Server based path-oblivious and fault-reactive protocols can thus attempt to mitigate the partial-fault impact. For example, MP-TCP breaks a single TCP flow into a collection of TCP sub-flows operating in parallel; if a sub-flow performs poorly, MP-TCP shunts data away onto other sub-flows [109]. FlowBender reactively tweaks outgoing-packet header bits when detecting degraded performance in an attempt to place traffic onto alternative network paths [75].

However, fault-reactive methodologies have drawbacks. First, they must detect performance degradation before mitigating faults; short-lived flows may end before benefiting. Furthermore, longer flows encountering early loss may be competitively disadvantaged even after mitigation, compared to flows that never traversed fault-containing paths. In cases where the network is ultimately not responsible for observed performance degradation, fault-reactive methods may also induce unnecessary network churn. Finally, fault-reactive methods do not perform fault localization, and thus cannot pro-actively mitigate fault impact for new flows. On the other hand, this dissertation shows that if servers are privy to application-traffic network path, they can leverage their knowledge of application performance to rapidly pinpoint specific network components that may contain a partial fault. Consequently, network operators may proactively mitigate partial-fault impacts for all traffic.

### 2.2.2 Partial faults confuse network-centric monitoring

In theory, network switches collectively possess full knowledge of application-traffic network paths. Each switch, for each packet, can determine the immediately neighouring device that the packet came from and is going to. Furthermore, switches can correlate locally-imposed

packet latency (via buffer occupancy counters), loss or locally observed corruption with specific links and ports. In practice, however, leveraging switch-based knowledge can be difficult. Switch CPUs are relatively weak and cannot neither examine nor report paths for every packet or flow. Switch-based counters have, in some cases, been unreliable; for example, Microsoft datacenter operators have detected 'silent' drops that are undetected by switch counters [133]. Thus, network-centric monitoring may underperform at partial-fault detection and localization.

As we have already seen, a variety of underlying mechanisms may cause partial faults. The specific fault mechanism and network configuration may impact the effectiveness of the fault-detection strategy in use. For example, for the motivating optical network packet-corruption scenario presented earlier, per-switch error counters were ineffective in the presence of cut-through routing; with store-and-forward routing they may have proved sufficient. On the other hand, for TCAM bit-flip induced black holes, leveraging error counters would not catch a problem since there was no packet corruption, nor any (packet) error from the perspective of the switch forwarding plane. Switch control-plane based routing table examination will also not catch the error, since the configuration is correct from the control plane's point of view. While switch-based unroutable-packet drop counters may trigger, it is difficult to distinguish black-hole misconfiguration from spoofed or erroneous traffic—either will increment such counters.

Instead, partial faults may require a perspective far removed from the switches themselves to catch the error; in the case of TCAM bit-flips, the fault was only localized by using a path diagnosis tool [133] that knew the full traffic network path for impacted traffic. Similarly, for ECMP imbalance errors, switch-based byte utilization counters are not enough due to the possibility of imbalanced elephant traffic. In this case, understanding application-driven network demands may be essential. For silent-drop partial faults, where no counters are incremented, server statistics coupled with network-path knowledge may be the only recourse. Thus, for many different kinds of partial faults, network-centric fault-localization approaches leveraging only switch-based knowledge may be insufficient. For the particular faults in question, only a combination of network-based and server-based monitoring proved effective.

**Table 2.1.** Fault localization mechanism capabilities by type.

| Type | Localizes fault | Root-causes fault | Fault agnostic | Application agnostic | Correlate fault with performance hit |
|---|---|---|---|---|---|
| Server based | No | No | Yes | Maybe | No |
| Network based | Maybe | Maybe | No | Yes | No |
| Unified (this work) | Yes | No | Yes | Yes | Yes |

## 2.3 Unifying network and server centric monitoring

Unifying network and server centric monitoring benefits partial-fault localization for several reasons summarized in Table 2.1. Unified monitoring enables partial-fault diagnosis that:

1. Can localize partial faults to specific components [22, 113, 133], unlike server-centric mechanisms [23, 64, 75, 100]. Network-centric monitoring may successfully localize faults [59] or fail to do so [38, 126] depending on circumstance.

2. Is fault-agnostic (unlike network-switch based scans [38, 112, 131] for specific errors like packet drop, corruption or high latency), due to examining fault-driven impact on server-based performance metrics. For the same reason, server-based approaches are also fault-agnostic [64, 75, 100], in some cases being able to broadly classify the perceived fault mechanism [23]. As a result, however, neither server-based approaches nor this dissertation's approach can root-cause partial faults, while network-based approaches may do so (if the fault mechanism is accounted for, which is not always the case [117]).

3. Is application-agnostic, assuming sufficient traffic volume as discussed in Chapter 5. Network-based monitoring is also application-agnostic [38, 112, 131], while server-based approaches may be application-agnostic [64, 75] or not [23, 100].

4. Can correlate faults with application impacts, allowing services to re-route impacted traffic without inducing churn (by not needlessly re-routing if 'normal' congestion was the cause).

While a variety of production systems and academic proposals target datacenter faults, including in some cases partial faults [22, 126], production monitoring systems often adopt a network-centric perspective [11, 59]. In other words, they answer questions like "How many packets are lost on this link?" and "What is the queuing delay incurred on this switch?". While such effects may point to partial faults, they may also indicate (non-fault-driven) network congestion. Error counters may not suffice depending on partial-fault mechanism and network configuration [126], or if the network device itself is buggy or unreliable [64, 133].

Service owners, on the other hand, leverage an application and server level view: they can answer questions like "What is the $99^{\text{th}}$ percentile latency for requests in this application?" and "How much loss is incurred by TCP flows on this server?", without any network-path visibility. Consequently, service owners cannot correlate observed loss and delay with network faults. Thus, even if a fault is detected, the service owner cannot know for sure if any observed application performance degradation was due to that fault or due to (non-fault-driven) network congestion.

Our method avoids purely network or server centric partial-fault localization. Instead, we acquire path-information for all flows from switches and expose it to servers, while avoiding expensive and unscalable on-switch computation. Servers use path information to correlate server-based metrics with network components, providing partial-fault detection and localization—even with unreliable or otherwise insensitive switches. If network operators can correlate performance degradation with network faults, they can preemptively mitigate fault impact until they fix the faults. Our work evokes prior network-tomography approaches [22, 58, 95], though datacenters exhibit properties that simplify analysis. We discuss some of these properties next, before leveraging these properties to simply partial-fault detection and localization in Chapter 5.

### 2.3.1 Load-balanced links mean outliers correspond with partial faults

While multipath topologies can complicate fault localization, they do possess fundamentally helpful characteristics. The desire to support big-data traffic patterns [5, 117] underscores a drive for network uniformity and even-load balancing [14, 15, 62, 63]. Evenly-distributed

network-core traffic is essential for achieving maximum Fat-Tree bisection bandwidth [14], and avoiding congestion losses [117]. Both application [34] and network [15] approaches actively seek and flatten imbalances. However, topology imbalance (caused by incremental deployment *or faults*) can cause imbalanced traffic even if traffic is normally balanced [62]. As we discuss in Chapter 3, imbalanced component-by-component network performance may indicate outlier-component affecting partial faults; this observation underpins the partial-fault localization system we present as our main contribution. In particular, we leverage server-based application and network-stack metrics, in collaboration with switch-provided network path information, to rapidly pinpoint partial-fault indicating performance anomalies.

### 2.3.2 Centralized network control enables collating viewpoints

Centralized control provides advantages for expediently detecting and localizing partial faults. Rather than relying on limited computational resources and performance metrics available at individual switches, or an incomplete network picture at servers, centralized control lets us devise a hybrid methodology using switches and servers in-concert to detect and localize partial faults more rapidly and accurately than any single device acting alone. Furthermore, centralized control allows us to proactively mitigate faults once localized [126].

In the next chapter, we discuss related work in detail, including existing production network health/fault monitoring systems, and academic proposals. We discuss datacenter traffic characteristics and how they can confound our task. Next, we motivate at a high level the methodology we use to localize partial faults, before fully presenting our solution in Chapter 5.

# Chapter 3

# Related work, challenges and a solution

Given the inevitability of datacenter network faults, network operators and academic researchers alike have deployed or proposed a large variety of fault detection and localization systems. This chapter positions our contributions vis-à-vis prior efforts. First, it presents criteria that we use as a guideline for evaluating fault-localization methodology efficacy. Next, it discusses existing work in the context of these criteria, noting both where the methodologies are successful and where they might fall short. Finally, it presents a high-level motivation for a partial-fault localization methodology that meets our criteria while avoiding various shortcomings present in existing work. This server-based, passively-monitoring, and outlier-analyzing methodology leverages datacenter network and traffic characteristics discussed in Chapters 2 and 4 and is further developed into our main contribution in Chapter 5.

## 3.1 Fault localization effectiveness criteria

In an ideal world, datacenter networks are completely reliable and performance-sapping partial faults never occur. In the real world, partial faults do occur; they can consume large amounts of time and effort to pinpoint [126], affect small subsets of traffic [133], be misattributed [126] and have a variety of underlying causes [64, 117, 126, 133]. Detection methodologies have to contend with various difficulties borne of computational and storage overheads [64, 133] as well as the engineering effort needed to change applications and hard-

ware [64, 66, 75]. These complicating factors suggest a set of criteria that can be used to evaluate the success of all fault-localization methodologies.

1. **Detection speed**

   Faster partial-fault localization can minimize fault application-impact. Academic proposals can detect non-partial faults within tens of microseconds and reroute traffic around failed links [93]; localizing partial faults, on the other hand, may potentially require minutes to hours of debugging and analysis [126, 133]. On the other hand, rapid partial-fault localization would enable datacenter operators to use reroute-based mitigation to reduce partial-fault application impact [75, 93], or at least disable faulty links until diagnosed and fixed. Therefore, our methodology aims to detect partial faults as quickly as possible—ideally, within tens of seconds rather than minutes or hours.

2. **Sensitivity to minuscule faults and false negatives**

   Partial faults may have particularly minuscule impacts, affecting only a small subset of traffic. For example, a fault may only impact traffic destined to a particular subnet, or mapped to a single forwarding rule [133]. Alternatively, a faulty link might affect all traffic, but with a drop rate that is difficult to distinguish from congestion-based losses. Consequently, detection systems may fail to diagnose faults. We call undiagnosed faults 'false-negatives'. Our methodology aims to detect partial faults regardless of confounding factors and minimize the false-negative rate.

3. **Accuracy and false positives in the face of network variability**

   Given widespread oversubscription in historic datacenter architectures [117], various applications have favoured rack-based data locality to avoid overloading the network core [5, 30, 47, 77]. Consequently, cross-rack traffic may be relatively sparse; servers in a Microsoft map-reduce style datacenter communicate either with in-rack servers only, or with a small fraction (1-10%) of servers outside the rack [77]. Cross-rack traffic sparseness means that large swathes of the traffic matrix may be empty or underutilized

within such datacenters; however, hot spot behaviour can still occur in portions of the traffic matrix. Perhaps due to hot spots, congestion frequently occurs within big-data style datacenters [77, 131]. Studies show that congestion is not static, however; it can last from seconds [32] to tens or even hundreds of seconds [77] before dissipating or manifesting elsewhere. Traffic may be bursty and exhibit on-off send behaviours [30]. Thus, temporal and spatial traffic instability can plague the network.

Like partial faults, hot spots and network congestion may cause packet loss or high-latency. Thus, a partial-fault detection system may incorrectly detect the presence of a fault, or incorrectly attribute a partial fault to a congested network component—we refer to this case as a 'false-positive' event. Datacenters can contain thousands of individual links, and thus false-positives can result in costly goose-chases that can waste time scrutinizing correctly functioning equipment [126]. Our methodology aims to minimize false-positive incidence, even in the face of congestion-based loss and delay.

4. **Ability to detect varied faults**

   Due to the variety of partial fault root causes, we may conceivably produce a system that can detect one type of fault (i.e. with a specific root cause) but not another. For example, packet corruption may be detected via framecheck error counters [126], but silent packet drops may not [133]. Drop and error counters are incapable of detecting latency-impacting faults. Given the proliferation of different partial-fault varieties, we aim for a fault-agnostic detection and localization methodology.

5. **Ability to pinpoint fault location and correlate application impact**

   While certain methodologies can detect partial faults, they may fall short of localizing them [23, 75]. Beyond fault localization, however, we argue that it is important to be able to correlate faults to specific application performance impact. Without the ability to correlate specific faults to application impacts, even if partial faults are pinpointed, a user is unaware whether any given fault causes observed application performance degrada-

tion. Thus, it is unclear whether applying mitigation steps (such as reactively rerouting application traffic [75]) will be helpful, or just create network churn.

Not localizing a fault has further downsides; even if we can mitigate fault impact sans localization, we have to do so reactively—we only route a network flow away from a faulty component after flow performance degrades. Short flows may send most of their traffic across a performance-impacting faulty component; longer flows may be disadvantaged compared to flows that did not traverse a faulty component. Thus, it is desirable to quickly pinpoint faulty network components so they can be fixed, or at least disabled to prevent future traffic from suffering performance degradation.

6. **Resilience against faulty monitors**

Fault monitors may be unreliable. For example, Microsoft datacenter operators discovered switches that 'silently' drop packets [133], where switches failed to increment drop counters. Given the possibility of unreliable network monitors, it is imperative to develop a fault detection methodology that successfully localizes faults even if individual monitors are unable to detect a fault. Our methodology aims to aggregate metrics collected at every server to robustly detect partial faults, even if certain servers or switches are unable to detect the fault. However, we do not handle the case where certain servers act as adversaries and maliciously and inaccurately detect (or fail to detect) faults.

7. **Computation, storage and network overheads**

Datacenter switches may possess limited computational resources [45]. Servers can possess greater resources, but non-application use represents cost-inducing overhead. Furthermore, storage space and network bandwidth dedicated to network monitoring may be limited. Large traffic volume combined with resource limitations may result in certain compromises. For example, per-flow monitoring may be eschewed [45] in favour of flow sampling [112] or aggregation [45]. Certain systems may only be able to process a subset of network traffic [133] or operate on-demand for short periods of time, rather than

continuously [22]. To provide an always-on fault detection methodology, our methodology aims to find partial faults with low computational, storage and network overheads.

8. **Need to modify hardware and application software**

   Academic proposals that improve network performance or reliability at the cost of making invasive hardware changes may not see widespread deployment [49, 66, 73], possibly due to the high engineering cost of modifying hardware. On the other hand, software-only backwards-compatible improvements, as well as systems leveraging existing capabilities have seen higher adoption [18, 75, 133]. Thus, our methodology strives to avoid hardware and (onerous) software modifications.

## 3.2 Existing fault management techniques

Having developed criteria for evaluating partial-fault localization effectiveness, we now examine existing approaches. As taxonomy, we categorize prior works by whether they use server-based metrics, network-based metrics, or a combination of server and network based information. In each case, we discuss how the methodology's viewpoint impacts its ability to satisfy our success criteria. After doing so, we possess enough context to compare these works with our main contribution, which we motivate at a high level in Section 3.3.

### 3.2.1 Server-centric fault detection

Server-centric fault detection leverages server-based information. Applications track a variety of metrics to quantify service health and, in various cases, facilitate service operation. For example, Hadoop MapReduce jobs comprise several worker tasks, with a job coordinator monitoring each task to ensure adequate overall progress (the coordinator restarts stalled tasks to prevent an entire job from stalling) [5]. The underlying HDFS filesystem tracks more granular metrics, such as the amount of time network traffic spends waiting on network system calls [20]. Latency sensitive applications can use metrics such as $99^{\text{th}}$ percentile RPC completion time.

In either case, service owners can set thresholds for what they consider as unacceptably poor performance; if a metric crosses the threshold, they can investigate. Note, however, that high level application requests can comprise several different network flows, any of which could potentially be responsible for diminished performance. Thus, datacenter operators leverage transport-level monitoring as well to get an insight into network flow performance [88, 100, 112].

**Fundamental deficiencies and impact on fault localization**

Due to multipath topologies and the prevalence of switch based ECMP routing, servers do not typically know network traffic path. Contemporary networks provide tracing functionality that enables servers to determine the path for subsets of traffic [11, 133]. However, servers may still not know what other traffic (from other servers) coincides on the paths traversed by their own traffic. Thus, when a server observes loss and latency for its own traffic, it is non-trivial to determine whether these effects are due to a fault in the network, network congestion (possibly caused by traffic from other servers) or a non-network cause such as server load. As a result, server-based methodologies may not satisfy several criteria from our fault localization wishlist.

In particular, path-oblivious approaches may be insensitive to minuscule faults. Multipath networks may dilute server's ability to notice consistent loss and latency since congestion losses may hide fault-driven losses. For example, consider a $k = 20$ fat-tree network connecting 2,000 servers. For any server pair, 100 paths connect them. If a single path contains a link with a (relatively high) 1% packet loss rate, and if packets are evenly spread across each path, the server-observed loss rate is 0.01%—in line with observed microburst-driven ToR-server loss rates [131]. Thus, relatively high-impact faults may be unnoticed. Furthermore, even if server-based systems detect [23] or mitigate [75] partial faults in spite of dilution, they cannot localize faults unless provided with some modicum of path information [84]. To make matters worse, long-lived congestion [15, 32, 77] may trick systems into reporting faults when none exist.

**Additional challenges noted in practice**

Additional challenges apply to server-centric partial-fault detection and localization methodologies. Since servers can possess tens to hundreds of thousands of active flows [112], comprehensively characterizing flow performance can be unrealistic due to computational, storage and network bandwidth overheads [112, 133]. To reduce overheads, we may sample traffic; either by considering 'interesting' subgroups (like control traffic or flow start/end packets) [133] or randomly [112]. Even with sampling, however, processing cost [112], data retention [112], and network bandwidth overheads [133] are considered difficult challenges. Furthermore, reducing the set of monitored traffic can impact fault-detection sensitivity (if we skip over impacted traffic) or fault-detection speed (network operators may have to wait till impacted traffic is sampled).

Deciding which traffic to examine and analyze can be a non-trivial task. TCP flows make up a significant chunk of datacenter traffic [18, 112]. While various studies examine TCP performance [21, 37], they often make simplifying assumptions to ease analysis. However, real application traffic has to contend with myriad effects that can complicate analysis. While some flows may be bulk-data and unbottlenecked, such as those moving large files across a distributed file system, others may perform relatively low-volume tasks such as synchronizing distributed systems or making RPC calls. Flows may stay dormant for long periods of time, and only send traffic within short bursts [30, 112]. Coupled with the large number of flows, understanding flow performance and the significance of collected metrics might be complicated.

For example, packet loss is generally considered harmful to flow performance. Flows with large bandwidth demands may encounter congestion-driven losses while those with limited demands (assuming they do not coincide with high-demand flows on the same path) might not; simply comparing the degree of packet loss across flows may pick out 'natural' differences in traffic behaviour rather than evidence of a network partial fault. Further confounding analysis is the fact that the limited-bandwidth flows might have bursty demands, resulting in more losses than high bandwidth flow when faced with shallow switch buffers.

One way to mitigate analysis complexity is to eschew analyzing application traffic and instead monitor latency and loss for injected probe traffic [11, 59, 64]. Probe-based methods can have downsides, however. While certain faults may impact probes and application traffic alike (for example, randomized packet loss), there may be faults that affect applications but not probes. For example, certain partial faults only affect specific destination subnets or ports [133]; they may harm application traffic while remaining invisible to probes. Probe traffic can cause computational overheads, consuming resources available for application traffic.

### 3.2.2 Network-centric fault detection

Network-centric fault detection refers to methodologies that use information available at network switches. For example, this includes flow information embedded within packet headers, link error and drop counters, and buffer occupancy information. As defined, it is not privy to information normally tracked at servers, like the number of retransmits or congestion window for a given TCP flow. Network-centric methodologies are theoretically advantaged in being able to correlate error, loss or delay with specific ports, links, queues, or other relevant components. Furthermore, it can theoretically reveal exactly which traffic is traversing afflicted components. In practice, however, network-centric approaches can be confounded in various ways.

#### Fundamental deficiencies and impact on fault localization

Network-centric fault detection methods are not privy to server-based information. At first, server-based information seems redundant—surely, switches can leverage their own monitoring to pinpoint faulty components! However, we argue that lacking server-based information is sub-optimal since it precludes desirable capabilities. Specifically, if a detection methodology can correlate server-based performance metrics to specific components (for example, ports on a switch), it can enable root-cause-agnostic fault detection and localization, as this dissertation will show. Root-cause-agnostic operation is advantageous for two reasons.

1. While switches can be designed to detect specific faults (for example, packet checksums to detect corruption), not every failure can be anticipated during initial switch engineering. Production experience suggests that unexpected and varied faults can and do occur, and that the right kind of monitoring is not always anticipated and provisioned [117].
2. Switches may be unreliable in various fault scenarios, either missing [64, 133] or misattributing [126] partial faults. Given that switch fabric reliability may degrade with age [117], we cannot always assume switches will be able to detect faults; in which case, pinpointing faults via performance impact is a valuable ability.

Thus, network-centric fault detection methodologies that do not have access to server-based information have a comparatively harder task of detecting partial faults with variable and unanticipated root causes. They also risk insensitivity to certain failure modes, either due to compromised reliability or because the failure mode was unanticipated to begin with.

**Additional challenges noted in practice**

Like some server-centric approaches (c.f. TCP retransmits and NIC error counters), switches may use counters to detect packet loss or latency spikes [38, 112]. Counter-based approaches may pinpoint certain faults; for example, comparing port error and drop counters can reveal packet-corrupting links. However, faults like ECMP imbalance [126] or misconfigurations would not be detectable using error counters, and neither would silent packet drops [133].

Unfortunately, aggregating traffic into counters prevents switches from revealing what traffic is traversing the faulty component. While granular counters (perhaps tracking certain application port numbers or subnets) can provide some insight, they may compete with essential features (like routing tables) for hardware resources. An alternative approach is to sample flows traversing switch ports. However, sampling is subject to bias; elephant flows are more likely to be sampled, even though mouse flows can still carry critical traffic [15]. Additionally, switch CPUs are often relatively low performance [45]; thus, we may spend too many computational resources on monitoring at the expense of important tasks like participating in routing protocols.

Consequently, partial-fault monitoring sensitivity may suffer. While subsets of traffic can be checked on a case by case basis—for example, a network operator could configure a counter to check the drop rate for a specific protocol—in general it can be hard to comprehensively characterize traffic performance across a switch, and in particular if some subset of traffic is suffering from performance degradation. Furthermore, since scalably examining all traffic is infeasible, attributing a network fault to specific impacts on specific application traffic is difficult.

### 3.2.3 Hybridized approaches

Hybridized fault detection and localization methodologies leverage information present within the network core (for example, counters or sampled flow data located within network switches or middleboxes) and within the servers at the network edge.

In order to provide more insight into network path, and to attempt to localize faults to a given path, certain methodologies leverage tracing the path of packets through the network on a hop by hop basis. For example, Everflow [133] classifies certain subsets of application traffic (including, for example, protocol control traffic) as being 'interesting'; switches traversed by the traffic in question then mirror packet headers to a collection and analysis infrastructure that can pinpoint faults like routing black holes. Similarly, NetSight creates a postcard per packet per switch that is sent to a logically central analyzer; a collection of postcards comprises a packet history that can be used to debug network performance [66]. Another path recovery approach is to leverage traceroute-like behaviour in conjunction with sending parallel requests to characterize inordinate packet loss [11] on various network paths. Finally, active programming style approaches have attempted to modify switch design in order to enable the execution of programs on a per packet basis, thus providing greater visibility to servers [73, 74].

Other systems focus more on the health of individual links, rather than the hop-by-hop performance of application traffic. For example, Netbouncer [59] combines server-originated probe traffic with packet encapsulation to inject probes into specific links in the network, allowing servers to characterize packet loss and latency on a link-by-link basis in the network.

### Hybrid methodology benefits and impact on fault localization

Leveraging both network-based and server-based information provides various advantages to hybrid methodologies. In particular, while network-centric methodologies might be unable to scalably examine all traffic or correctly determine what subset of traffic to examine, hybridized approaches like Everflow can leverage server-based context to focus on important traffic. Specifically, servers can mark specific flows that might be experiencing delay or loss or otherwise harming application performance, and determine both the nominal network path for the traffic and whether the network is dropping the packet at a given hop. Probe traffic injected via Netbouncer-like services can provide further data to diagnose a potentially faulty link.

As a result, hybridized methodologies may satisfy several criteria presented at the start of this chapter. Detection speed is aided since fault investigations can be triggered by passively monitoring production traffic as it is generated—datacenter operators need not wait for the 'correct' type of probe traffic to trigger latent faults. Sensitivity is similarly improved, especially since hybrid fault detection systems need not lose information via counter-based aggregation at switches or servers. Since hybrid systems can leverage server-based metrics and trigger on application performance impact, it can pinpoint faults regardless of root cause [133]. Furthermore, application impact can and has been attributed to specific partial faults using such systems, even in the presence of unreliable monitors [133]. Finally, complicated processing can be offloaded from relatively weak switch CPUs and onto dedicated fault-monitoring servers [133].

### Challenges noted in practice

In practice, various challenges still face hybridized fault detection and localization methodologies. Traceroute approaches can induce CPU load at switch CPUs, while Everflow style match-mirror approaches can cause significant network overheads (requiring Everflow in particular to curtail the traffic it looks at [133]). Despite the relatively high degree of control a datacenter operator has on software and networking hardware, there appears to yet be significant inertia when it comes to making invasive or elaborate changes. Production systems [59, 133] and

successfully deployed protocols [11, 18] tend to leverage existing hardware capabilities while methodologies requiring switch modifications [49, 73, 79] do not see large-scale deployment.

## 3.3 Accounting for traffic and faults with outlier analysis

Thus far, we have established success criteria for fault localization methodologies, and examined how existing approaches have either satisfied or fallen short of satisfying those criteria. In particular, we have seen that server-centric and network-centric methodologies have various shortcomings that can hinder fault localization effectiveness, while existing hybridized systems provide significant benefits but have various overheads to cope with. It is this dissertation's goal to contribute a methodology that satisfies *all* of the success criteria, by leveraging our understanding of datacenter topologies and traffic patterns.

In addition, we wish to devise a system that can not only find network-based partial faults regardless of root cause, but also enable correlation between faults and impacted traffic. We desire the system to run continuously without requiring manual intervention by either users, network administrators or user applications. We argue that datacenter topology regularity and large-scale traffic together enable a low-overhead outlier-analysis methodology. Here, we present our intuition for why such a method is viable and how it satisfies our criteria. In Chapter 4, we discuss Facebook datacenters and how they allow particularly effective outlier-analysis. We then formalize our methodology and present a working prototype in Chapter 5.

### 3.3.1 Localizing faults via passive monitoring and outlier analysis

This dissertation claims that datacenter topologies and traffic patterns enable an outlier-analysis-based fault localization methodology that relies on passive monitoring of server-based statistics. To support this claim, we first present our intuition of why passive-monitoring of server-based statistics could reveal partial faults if we can correlate with path information. Next, we show that naïve outlier-analysis is unlikely to succeed, but carefully chosen comparisons can make the approach viable within contemporary datacenter environments.

**Passive monitoring could reveal partial faults with path correlation**

In general, observing application metrics or network flow performance to ascertain network performance quality is complicated by several effects. For example, if we consider a simple RPC protocol network and examine the request completion latency as a network health indicator, we might notice elevated request latency either due to a faulty network component, or due to unrelated effects like scheduler latency or disk access for servers handling RPC calls. Thus, a single bad performance metric in isolation is not necessarily indicative of a problem.

On the other hand, consider a set of links in an ECMP group carrying RPC traffic, where one link hosts a partial fault that randomly corrupts packets. Once can surmise that traffic traversing the faulty link is more likely to encounter loss than within the non-faulty links. Conversely, assuming sufficient traffic volume, we may expect roughly similar performance across all the non-faulty links. Thus, we may consider comparing link-by-link traffic performance for this ECMP group, and consider outliers to be indicative of a partial fault.

However, comparing performance of application traffic across links or switches will not always yield meaningful results. Consider, for example, a Top-of-Rack (ToR) switch connected to several A server that is sending or receiving a larger rate of traffic is more likely to incur loss than a server that is barely sending any traffic at all. Comparing the performance of applications and network flows across the respective Host-to-Tor (access) links would falsely indicate a performance issue on the access link for the busier server.

**Accounting for traffic pattern hot spots**

Datacenter hot spots [30, 47] can invalidate naïve link-by-link performance comparisons. However, we argue that extensive application [34, 112] and network load balancing enable comparing certain components. For example, consider ECMP routed networks containing equal-cost paths between servers. An initial approach would be to compare, for traffic between any server pair, path-by-path traffic performance. Equal-cost paths and ECMP routing suggest that the suggest relative mix of traffic on any path should be similar in nature and in performance.

However, this approach is not generally feasible—the very existence of hot spots suggests that the amount of traffic might be minuscule between any given pair of servers in various datacenters. Coupling this with the tens of thousands of paths that contemporary datacenters may provide between any pair of servers [112], and we end up with a situation where a relatively small set of flows may be split amongst a large number of paths, leading to effects such as elephant flows on one path degrading the perceived performance of that path compared to others [15].

Instead, we argue for comparing server traffic on a link-by-link basis, irrespective of the destination server and in spite of traffic-matrix hot spots, for certain subsets of links. Suppose we have a fat-tree network with a skewed traffic pattern containing one hot spot. Specifically, one network pod receives traffic from all other pods; the other pods do not exchange any traffic with each other, and neither does the recipient pod contain any intra-pod traffic. In this case, how will traffic be spread within the network core? To make the example concrete, suppose the fat-tree contains $C$ core switches and $S$ servers. Suppose each source server (in one of the pods other than the destination pod) sends $F$ flows to a random destination-pod server, for a total of $S \times F$ flows. From the point of view of a single source server, each of its $F$ flows has an equal probability of traversing one of the four available core switches to the destination pod; the expectation being that each core switch ends up with $F \div C$ flows from that server.

Now suppose that the flows are infinitely long, but have varying rates; suppose 25% of the flows are limited to 1 Mbps, 25% are limited to 10 Mbps and 50% have no rate limit. Since the ECMP forwarding decision is independent of the flow characteristics, we would expect to see each core server end up with $F \div (2 \times C)$ non-rate-limited flows from the server. Similarly, we expect to see $F \div (4 \times C)$ 1 Mbps flows per core, and the same number of 10 Mbps flows. Thus, the server's expected network utilization on a core switch by core switch basis is identical.

This observation is true for every server sending traffic, regardless of the specific traffic characteristics of the server's traffic. The expected mix of traffic per core switch from any given server is the same; if the number of flows is significantly higher than the number of core switches, we hypothesize that the utilization per core switch will be similar over time. We further

hypothesize that, in the absence of faults, the performance observed by applications and server network stacks will be similar on a core-by-core basis as well.

On the other hand, if a given core switch contains a partial fault—for example, suppose it drops a small percentage of packets randomly—we hypothesize that the performance observed by servers for traffic on the faulty core switch will be strictly worse than performance observed through other switches. In other words, despite the presence of traffic matrix hot spots, we should be able to compare traffic performance on a core switch by core switch basis on a fat-tree network, and treat deviations from the norm as potentially indicating a partial fault.

**Outlier analysis enables traffic-agnostic fault detection**

Outlier analysis thus enables the detection of anomalous network performance on a core-by-core basis, thus potentially allowing us to find partial faults in a manner that is root-cause-agnostic (since it looks at server-based metrics and only requires that a fault have some measurable impact) and traffic-agnostic (it only requires that the number of flows is large compared to the number of components that the flows traverse). We argue that a fault localization system built on such a system can localize faults for every link and component in the datacenter where the presence of equivalent traffic can be demonstrated, and claim this encompasses most links within contemporary datacenters. Furthermore, we argue that such a system can satisfy all of our success criteria from the start of this chapter in Chapter 5. First, however, we demonstrate that our intuition for outlier-analysis is sound by validating our hypotheses using real applications and synthetic workload on a private fat-tree testbed.

### 3.3.2 Outlier analysis demonstration

To demonstrate outlier analysis in action, we perform a set of synthetic experiments on a small testbed that is representative of real world network topologies—specifically, a $k = 6$ fat-tree [14] at 50% occupancy (3 out of 6 pods fully populated, 3 empty). We emulate real world workloads by running real world applications; Redis [7] and Hadoop [5] in order to

examine latency sensitive and bulk traffic workloads respectively. We expect core-by-core traffic performance to be similar in the absence of partial faults, and worse than normal on a core switch with an induced packet-loss partial fault. Furthermore, we expect this to be the case regardless of application traffic pattern and flow characteristics. Thus, we consider representative real world applications that a datacenter might run and examine the efficacy of outlier analysis at finding anomalous network behaviour that we can correlate with partial faults.

**Latency sensitive traffic**

Redis is an in-memory cache implemented as a client-server key-value store. Redis data occupies main memory; thus, disk accesses do not impact latency. Thus, Redis is highly sensitive to poor network performance, where packet drops can yield order-of-magnitude request latency increases. Thus, we use request latency as a poor-performance signal, expecting that packet drops will increase request latency for clients connected via the bad link.

To verify this, we deploy Redis on our testbed using 9 servers and 18 clients. Each client creates a large number of connections spread amongst the Redis servers. Over each connection, the client requests objects of various sizes (either small objects fitting in a single packet response, or larger objects several kilobytes in size). We record per-server request-latency distributions grouped by transited core switch. The clients in this experiment occupy two of the three available pods. For any given client/server pair, there are nine core switches the flow may transit. In this example, we pick one core switch and induce a partial fault, randomly dropping 1% of all carried packets; thus we expect to see a correlation between request latency and core switch.

Figure 3.1 shows packet-loss impact on Redis request latency. Eight series, corresponding to non-faulty switches, demonstrate nearly identical distributions: every request takes approximately 20 milliseconds to complete. For the remaining series, shaded gray and corresponding to the faulty switch, $99^{\text{th}}$-percentile request latency increases to over 200 milliseconds. In this scenario, each server response fits within one IP packet. With a 1% packet drop rate, 1% of responses drop and incur TCP retransmit timeout latency from the server TCP stack. This case

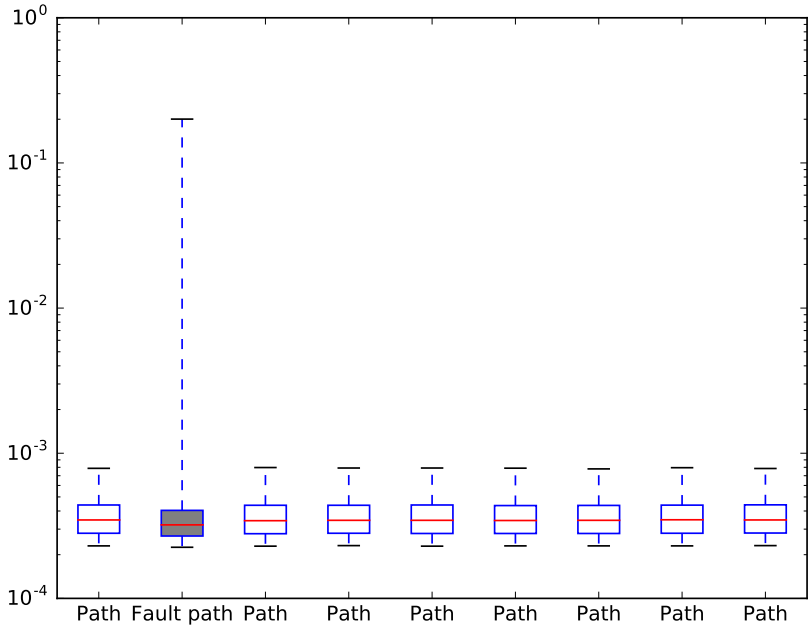



**Figure 3.1.** Redis request latency. The *y*-axis is in seconds; whiskers correspond to the 99$^{\text{th}}$ percentiles.

represents an ideal scenario; with larger response sizes, any given request is more likely to encounter packet loss, and thus a larger percentage of requests will encounter packet loss and subsequent latency jumps (e.g. a 10k response in the presence of 1% packet loss means we will see 95$^{\text{th}}$-percentile latency jump by an order of magnitude).

To summarize, the large number of flows within the experiment leads to variations in application flow performance—in this case, latency—being evenly spread across the core switches in our testbed network. Consequently, partial-fault-like performance anomalies centered on a given core switch visually appears as a clear outlier compared to normally operating switches. While this experiment leveraged an even distribution of traffic—each client evenly spreads connections amongst the available servers—further experimentation in Chapter 5 shows us that outlier analysis can be effectively used even with gross traffic imbalances.

**Bulk traffic workloads**

Hadoop is a bulk-data processing framework hosting computation jobs that can shuffle large quantities of data across the network. Jobs can vary in several ways including input data size, the number of servers involved, or the amount of computation. Consequently, the number, size and frequency of network flows can vary between jobs. Suppose a partial fault existed in a Hadoop-serving network, which caused randomized packet corruption and loss. With such a fault, an unpredictable subset of the flows supporting a Hadoop job can be impacted. While characterizing the specific impact of a fault on an overall Hadoop job can be complicated, in general packet loss can slow down transfer rates and increase job completion time.

Intuitively, since flows use TCP, we expect flows routed over a faulty link to experience packet loss and thus poor throughput. A Hadoop job is comprised of several flows of varying size; we expect that larger flows (with more packets) on lossy links are more likely to suffer loss—and poor throughput. Consequently, for flows experiencing good throughput over a faulty link, we expect flow size distribution to skew smaller. Thus, one possible outlier-analysis based approach would be to characterize the flow size distribution and flow rate together on a link-by-link basis. For non-faulty links, we would expect similar flow rates and flow sizes. However, a faulty link would likely possess skewed statistics, where only smaller flows achieve high rate. Hadoop instrumentation provides us with flow size and start and end times, thus providing us flow rate. Our testbed simulates ECMP routing via server-based source routing, thus providing us the ability to correlate flow size and rate information to network component.

To test the proposed approach, we deployed a Hadoop installation on our testbed and installed a query framework called Hive [120] on top. We executed a variety of queries of different size and computational requirements against a free data set [2]. By varying workload complexity, we aim to demonstrate the efficacy of outlier analysis on non-homogeneous workloads—in other words, whether it is effective regardless of the mix of flow size, count or temporal behaviour. In this experiment, we picked one core switch and induce a partial fault, dropping 2% of all

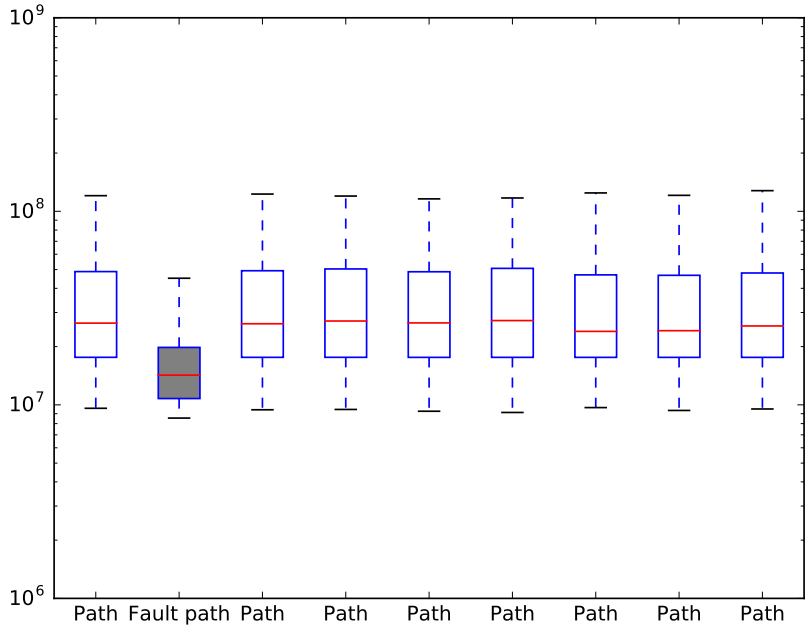



**Figure 3.2.** Flow size distribution of high-throughput flows for Hadoop. The *y*-axis is in bytes.

packets that transit it; we expected to see similar flow size and rate performance on the non-faulty switches, and diminished flow sizes amongst the high-rate flows for the faulty switch.

We classify the fastest 10% of flows as "high-throughput", and plot the flow size distribution in Figure 3.2. Any flow can traverse one of nine core switches. As expected, the gray-shaded faulty-core-switch flow size distribution skews smaller, by half an order of magnitude in the median case; i.e., the median flow size on non-impacted switches is roughly 500% larger. Hence, faulty-switch flow performance is clearly distinguishable from non-faulty switches.

### 3.3.3 Outlier analysis in real world datacenters

While partial-fault localization via outlier analysis shows promise, several hurdles need to be cleared to build an effective fault detection and localization system:

1. Our testbed allows us to recover all flow network paths, since we simulate ECMP via source routing. In general, though, servers are not privy to flow path information without

computationally-expensive (and thus limited scale) methods like traceroute [133]. Thus, we need to recover paths for all flows scalably.

2. Outlier analysis effectiveness depends on traffic behaviour. For example, the number of flows must be relatively high. If the traffic matrix is sparse enough, there might not be enough cross-core traffic to develop enough confidence in outlier-analysis based results unless we wait for longer periods of time (thus allowing the aggregate traffic matrix to shift and eliminate cold spots). Thus, we need to properly characterize the traffic characteristics of any datacenter where an outlier detection based methodology may apply.
3. While outliers are apparent visually to human observers, it must be automatable to deal with datacenter scale, in a manner that does not induce false positives or false negatives.

Thus, we spend the next chapter performing a detailed study of production Facebook datacenters that reveal significant differences compared to prior studies focusing on Microsoft; in particular, we note several characteristics that our advantageous to the outlier analysis fault detection approach. Following this we formalize our methodology and present the design and implementation of a prototype fault detection system based upon it at Facebook in Chapter 5.

# Chapter 4

# Facebook datacenters and outlier analysis

Outlier analysis via equivalence set analysis, while showing promise within private testbed experiments, may vary in effectiveness depending on the nature of datacenter traffic. In this chapter, we investigate the datacenter traffic at Facebook, a large-scale social media website, examining various characteristics that aid partial fault localization. Historically, the bulk of datacenter traffic studies have focused on Microsoft datacenters, and have set a variety of expectations or rules of thumb regarding datacenter traffic characteristics. We find that Facebook datacenters upend several of these expectations, and thus also discuss various characteristics of broader interest to the computer networking community.

While Facebook operates traditional services like Hadoop, its core Web service and supporting cache infrastructure exhibit behaviors that contrast with those reported in the literature. We report on the contrasting locality, stability, and predictability of network traffic in Facebook's datacenters, and comment on their implications for network architecture, traffic engineering, switch design and partial fault localization—in particular, on how various characteristics of Facebook traffic are particularly amenable to outlier-analysis-based partial-fault detection.

## 4.1 Inside the social network's (datacenter) network

Datacenters are revolutionizing the way in which we design networks, due in large part to the vastly different engineering constraints that arise when interconnecting a large number

**Table 4.1.** Each of our major findings differs from previously published characterizations of datacenter traffic. Many systems incorporate one or more of the previously published features as design assumptions.

| **Finding** | **Previously published data** | **Potential impacts** |
|---|---|---|
| Traffic is neither rack local nor all-to-all; low utilization (§4.4) | 50–80% of traffic is rack local [30, 47] | Datacenter fabrics [14, 60, 118] |
| Demand is wide-spread, uniform, and stable, with rapidly changing, internally bursty heavy hitters (§4.5) | Demand is frequently concentrated and bursty [30, 31, 32] | Traffic engineering [15, 32, 65, 123] |
| Small packets (outside of Hadoop), continuous arrivals; many concurrent flows (§4.6) | Bimodal ACK/MTU packet size, on/off behavior [30]; ¡5 concurrent large flows [18] | SDN controllers [8, 61, 87, 106, 115]; Circuit/hybrid switching [17, 53, 92, 123] |

of highly interdependent homogeneous nodes in a relatively small physical space, as opposed to loosely coupled heterogeneous end points scattered across the globe. While many aspects of network and protocol design hinge on these physical attributes, many others require a firm understanding of the demand that will be placed on the network by end hosts. Unfortunately, while we understand a great deal about the former (i.e., that modern cloud datacenters connect 10s of thousands of servers using a mix of 10-Gbps Ethernet and increasing quantities of higher-speed fiber interconnects), the latter tend to be not disclosed publicly.

Thus, many recent proposals are motivated by lightly validated assumptions regarding datacenter workloads, or, in some cases, workload traces from a single, large datacenter operator [30, 77]. These traces are dominated by traffic generated by a major Web search service, which, while significant, may differ from the other major cloud service's demands. Here, we study Facebook datacenter workloads. We find that traffic studies in the literature are not entirely representative of Facebook's demands, calling into question the applicability of some proposals based upon prevalent assumptions on datacenter traffic behavior. This situation is particularly acute when considering novel network fabrics, traffic engineering protocols, and switch designs.

As an example, a great deal of effort has gone into identifying effective topologies for datacenter interconnects [14, 52, 60, 118]. The best choice (in terms of cost/benefit trade-

off) depends on the communication pattern between end hosts [107]. Lacking concrete data, researchers often design for the worst case, namely an all-to-all traffic matrix in which each host communicates with every other host with equal frequency and intensity [14]. Such an assumption leads to the goal of delivering maximum bisection bandwidth [14, 62, 118], which may be overkill when demand exhibits significant locality [47].

In practice, production datacenters tend to enforce a certain degree of oversubscription [30, 60], assuming that either the end-host bandwidth far exceeds actual traffic demands, or that there is significant locality in demand that decreases the need for full connectivity between physically disparate portions of the datacenter. The precise degree of oversubscription varies, but there is general agreement amongst operators that full connectivity is rarely worthwhile [27]. To mitigate potential "hot spots" caused by oversubscription, researchers have suggested designs that temporarily enhance connectivity between portions of the datacenter [15, 65, 132]. The utility of these approaches depends upon the prevalence, size, and dynamics of such hot spots.

In particular, researchers have proposed inherently non-uniform fabrics which provide qualitatively different connectivity to certain portions of the datacenter through hybrid designs, typically including optical [92, 123] or wireless links [65, 132]. If demand is stable and/or predictable over reasonable time periods, it may be feasible to provide circuit-like connectivity between portions of the datacenter [53]. Alternatively, network controllers could select among existing paths in an intelligent fashion [32]. Regardless of the technology involved, all of these techniques require traffic to be predictable over non-trivial time scales [32, 53, 65, 92, 123].

Finally, many have observed that the stylized nature of datacenter traffic opens up avenues for increasing switching hardware efficiency. While some have proposed straightforward modifications like decreased buffering, port count, or sophistication [14] in various switching fabric layers, others have proposed replacing conventional packet switches either with circuit or hybrid designs that leverage locality, persistence, and predictability of traffic demands [92]. More extreme solutions advocate connecting servers directly [60, 62]. Obviously, when, where, or if any of these approaches makes economic sense hinges tightly on offered loads [107].

While there have been a number of studies of university [32] and private datacenters [30], many proposals cannot be fully evaluated without significant scale. Almost all of the previous studies of large-scale (10K hosts or larger) datacenters [15, 30, 32, 47, 60, 65, 77] consider Microsoft datacenters. While Facebook's datacenters have some commonality with Microsoft's, such as eschewing virtual machines[1] [32], they support a very different application mix. As a result, we observe a number of critical distinctions that may lead to qualitatively different conclusions; we describe those differences and explain the reasons behind them.

Our study is the first to report on production traffic in a datacenter network connecting hundreds of thousands of 10-Gbps nodes. Using both Facebook-wide monitoring systems and per-host packet-header traces, we examine services that generate the majority of the traffic in Facebook's network. While we find that the traffic patterns exhibited by Facebook's Hadoop deployments comport well with those reported in the literature, significant portions of Facebook's service architecture [24, 34] vary dramatically from the MapReduce-style infrastructures studied previously, leading to vastly different traffic patterns. These patterns possess several characteristics that greatly advantage outlier-analysis based partial-fault localization, including:

- Traffic is overwhelmingly cross-rack across multiple services, though not all-to-all. Instead, locality depends upon the specific service but is stable across time periods from seconds to days. This temporally stable traffic is spread amongst a plethora of equal-cost paths traversing the network core, in both the classic four-post and newer 'Fabric' topologies [19]. These characteristics are instrumental to supporting outlier-analysis based fault localization; we formalize the specific requirements in Section 5.1, and demonstrate how temporal stability impacts how rapidly we can pinpoint partial faults in Section 5.6.
- Non-Hadoop flows are long-lived but not very heavy. Load balancing effectively distributes traffic across hosts; so much so that traffic demands are quite stable over even sub-second intervals. As a result, heavy hitters are not much larger than the median flow, and the

[1] However, this dissertation also examines Microsoft Azure datacenters that do use virtual machines.

set of heavy hitters changes rapidly. Instantaneously heavy hitters are frequently not heavy over longer time periods, likely confounding many approaches to traffic engineering. However, this property yields a traffic matrix that lacks both temporal and spatial hot spots. Consequently, we show in Section 5.4 that we can effect partial fault detection that pinpoints faults to specific links on timescales ranging from a few seconds to a few tens of seconds (due to the lack of persistent temporal hot spots) and either on a per-link or per-destination basis (due to the lack of persistent spatial hot spots).

- Packets are small (median length for non-Hadoop traffic is less than 200 bytes) and do not exhibit on/off arrival behavior. Servers communicate with 100s of hosts and racks concurrently (i.e., within the same 5-ms interval), but the majority of traffic is often destined to (few) 10s of racks—although the set of racks changes from interval to interval. ECMP routing, Facebook traffic granularity and high flow counts yield per-link traffic loads that carry not only similar numbers of flows, but similar numbers of bytes across links in the same layer of the network topology. This enables link-by-link comparisons of traffic network performance, which we highlight in Section 5.3.2.

While we do not offer these workloads as any more representative than others—indeed, they may change as Facebook's services evolve—they do suggest that the space of cloud datacenter workloads is more rich than the literature may imply. As one way to characterize the significance of our findings, Table 4.1 shows how our results compare to the literature, and cites exemplar systems that incorporate these assumptions in their design.

The rest of this chapter is organized as follows. Section 4.2 surveys previous datacenter traffic studies. Section 4.3 provides a high-level description of Facebook's datacenters, services, and our collection methodologies. We then analyze traffic aspects within a number of Facebook's datacenters that impact provisioning (Section 4.4), traffic engineering (Section 4.5), and switch design (Section 4.6). Section 4.7 highlights relevant characteristics that impact our outlier-analysis based partial fault detection methodology that we present in Chapter 5.

## 4.2 Related datacenter traffic characterization work

Initial datacenter workload studies were conducted via simulation [16] or on testbeds [50]. Subsequently, however, a number of studies of production datacenter traffic have been performed, primarily within Microsoft datacenters. It is difficult to determine how many distinct Microsoft datacenters are reported on in literature, or how representative that set might be. Kandula *et al.* observe that their results "extend to other *mining* data centers that employ some flavor of map-reduce style workflow computation on top of a distributed block store," but caution that "*web* or *cloud* data centers that primarily deal with generating responses for web requests (e.g., mail, messenger) are likely to have different characteristics." [77]. By that taxonomy, Facebook's datacenters clearly fall in the latter camp. Jalaparti *et al.* [72] examine latency for Microsoft Bing services that are similar in concept to Facebook's service; we note both similarities to our workload (relatively low utilization coupled with a scatter-gather style traffic pattern) and differences (load appears more evenly distributed within Facebook datacenters).

Three major themes are prevalent in prior studies, and summarized in Table 4.1. First, traffic is found to be heavily rack local, likely as a consequence of the application patterns observed; Benson *et al.* note that for cloud datacenters "a majority of traffic originated by servers (80%) stays within the rack" [30]. Studies by Kandula *et al.* [77], Delimitrou *et al.* [47] and Alizadeh *et al.* [18] observe similarly rack-heavy traffic patterns.

Second, traffic may be bursty and unstable across various timescales—an important observation, since traffic engineering often depends on relatively long-lived, predictable flows. Kapoor *et al.* observe that packets to a given destination often arrive in trains [78]; while Benson *et al.* find a strong on/off pattern with a log-normal packet inter-arrival distribution [31]. Changing observation timescale can ease prediction; Delimitrou *et al.* [47] note that while locality varies day-by-day, it remains consistent at the scale of months. Conversely, Benson *et al.* [32] claim that while traffic is unpredictable at timescales $\geq 150$ seconds, it may be relatively stable on a few seconds timescale, and discuss traffic engineering mechanisms for such traffic.

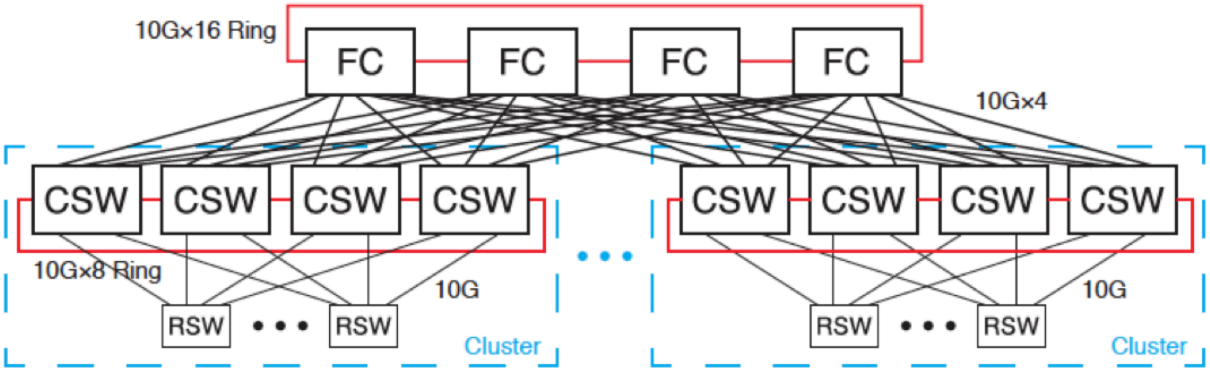



**Figure 4.1.** Facebook's 4-post cluster design [52]

Finally, previous studies have consistently reported a bimodal packet size [30], with packets either approaching the MTU or remaining quite small, such as a TCP ACK segment. We find that Facebook's traffic is very different, with a consistently small median packet size despite the 10-Gbps link speed. Researchers have also reported that individual end hosts typically communicate with only a few (e.g., less than 5 [18]) destinations at once. For some Facebook services, an individual host maintains orders of magnitude more concurrent connections.

## 4.3 A Facebook datacenter

In order to establish context necessary, this section provides a brief overview of Facebook's datacenter network topology, as well as a description of the services that it supports; more detail is available elsewhere [9, 19, 24, 34, 52]. We then describe the distinct collection systems used to assemble the network traces analyzed in the remainder of the chapter.

### 4.3.1 Datacenter topology

Facebook's network consists of multiple datacenter sites connected by a backbone network. Each site contains one or more buildings (henceforth datacenter), each containing multiple clusters. A cluster is a unit of deployment in Facebook datacenters. Each cluster employs a conventional 3-tier topology depicted in Figure 4.1, reproduced from a short paper [52].

Machines are organized into racks and connected to a top-of-rack switch (ToR) via 10-Gbps Ethernet links. The number of machines per rack varies from cluster to cluster. Each ToR in turn is connected by 10-Gbps links to four aggregation switches called cluster switches (CSWs). All racks served by a particular set of CSWs are said to be in the same cluster. Clusters may be homogeneous in terms of machines—e.g. Cache clusters—or heterogeneous, e.g. Frontend clusters which contain a mixture of Web servers, load balancers and cache servers. CSWs are connected to each other via another layer of aggregation switches called Fat Cats (FC). As will be seen later in this chapter, this design follows directly from the need to support a high amount of intra-cluster traffic. Finally, CSWs also connect to aggregation switches for intra-site (but inter-datacenter) traffic and datacenter routers for inter-site traffic.

Most of Facebook's current datacenters employ this 4-post Clos design. Facebook's datacenters are migrating, however, to a next-generation Fabric architecture[2] [19]. This chapter analyzes data collected from machines in traditional 4-post clusters, although Facebook-wide statistics (e.g., Table 4.3) cover hosts in both traditional 4-post clusters and newer Fabric pods.

One distinctive aspect of Facebook's datacenters is that servers typically have precisely one role: Web servers (Web) serve Web traffic; MySQL servers (DB) store user data; query results are stored temporarily in cache servers (Cache)—including leaders, which handle cache coherency, and followers, which serve most read requests [34]; Hadoop servers (Hadoop) handle offline analysis and data mining; Multifeed servers (MF) assemble news feeds [96]. While other roles exist, these roles represent the majority, and will be the focus of our study. Only a relatively small number of machines do not have a fixed role and are dynamically repurposed. Facebook's datacenters do not typically house virtual machines: each service runs on a physical server. Moreover—and in contrast to previously studied datacenters [30]—to ease provisioning and management, racks typically contain only servers of the same role. The combination of co-locating same-role servers by rack, coupled with the type of inter-service interaction, significantly aids outlier-analysis based partial fault localization as we discuss in Section 4.7.

[2]In particular, we evaluate our partial fault localization system within Fabric-equipped datacenters in Chapter 5.

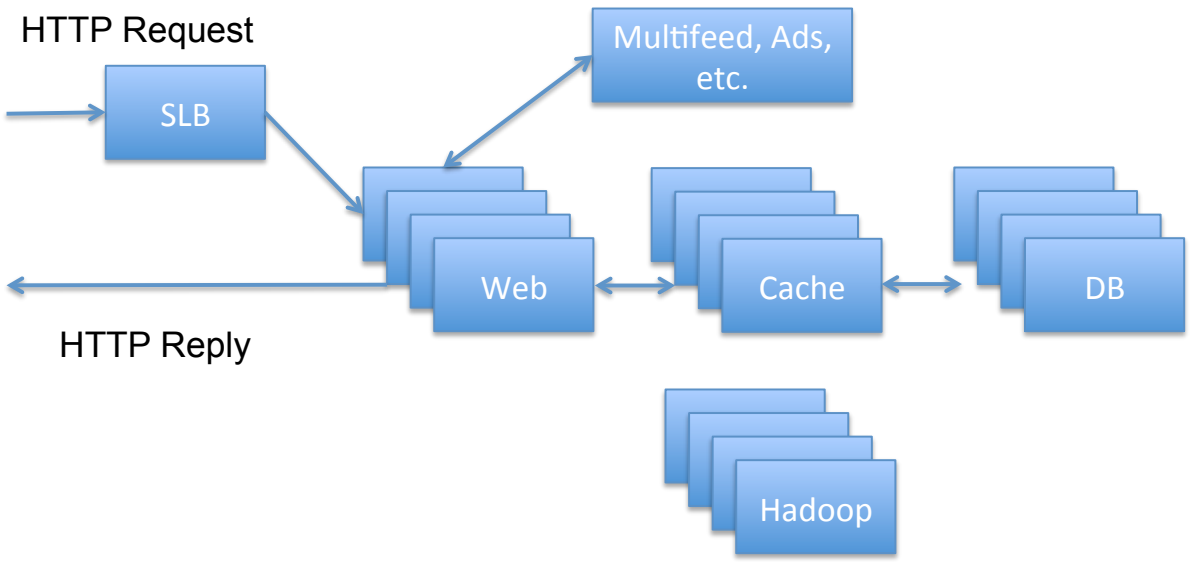



**Figure 4.2.** How an HTTP request is served

### 4.3.2 Constituent services

The organization of machines within a cluster—and even a datacenter—is intimately related to the communication patterns between the services they support. We introduce the major services by briefly describing how an HTTP request is served by `http://facebook.com`, shown in Figure 4.2. When an HTTP query hits a Facebook datacenter, it arrives at a layer-four software load balancer (SLB) [119], which redirects the query to a Web server. Web servers are stateless and contain no user data, instead fetching data from the cache tier [34]. In case of a miss, a cache will then fetch data from the database tier. At the same time, the Web server may communicate with other backend machines to fetch objects such as news stories and ads. Table 4.2 quantifies the relative traffic intensity between services by classifying the outbound traffic from different servers—a Web server, cache leader (cache-l), cache follower (cache-f), and Hadoop—based upon the role of the destination host. (We describe our data source in Section 4.3.3.)

In contrast to most service tiers, Hadoop nodes are not involved with serving end-user requests. Instead, Hadoop clusters perform offline analysis such as data mining. HDFS and Hadoop MapReduce are the main applications running on these servers.

**Table 4.2.** Breakdown of outbound traffic percentages for four different host types

| Type | Web | Cache | MF | SLB | Hadoop | Rest |
|---|---|---|---|---|---|---|
| Web | - | 63.1 | 15.2 | 5.6 | - | 16.1 |
| Cache-l | - | 86.6 | 5.9 | - | - | 7.5 |
| Cache-f | 88.7 | 5.8 | - | - | - | 5.5 |
| Hadoop | - | - | - | - | 99.8 | 0.2 |

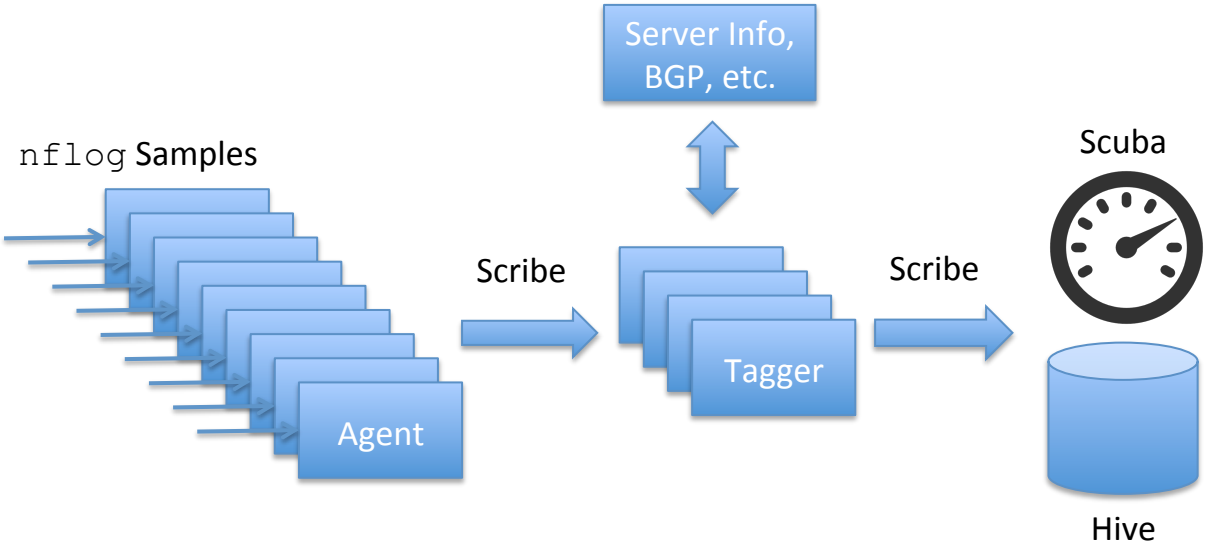



**Figure 4.3.** Fbflow architecture

### 4.3.3 Data collection

Due to the scale of Facebook's datacenters, it is impractical to collect complete network traffic dumps. Instead, we consider two distinct sources of data. The first, Fbflow, constantly samples packet headers across Facebook's entire global network. The second, port mirroring, focuses on a single machine (or rack) at a time, allowing us to collect complete packet-header traces for a brief period of time at particular locations within a single datacenter.

#### Fbflow

Fbflow is a production monitoring system that samples packet headers from Facebook's entire machine fleet. Its architecture, comprised of two main component types—*agents* and *taggers*—is shown in Figure 4.3. Fbflow samples packets by inserting a Netfilter `nflog` target into every machine's `iptable` rules. The datasets we consider in this chapter are collected with

a 1:30,000 sampling rate. A user-level Fbflow agent process on each machine listens to the `nflog` socket and parses the headers, extracting information such as source and destination IP addresses, port numbers, and protocol. These parsed headers—collected across all machines in Facebook's datacenters—along with metadata such as machine name and capture time, are streamed to a small number of taggers using Scribe [9], a log aggregation system.

Taggers, running on a subset of machines, read a portion of the packet-header stream from Scribe, and further annotate it with additional information such as the rack and cluster containing the machine where the trace was collected, its autonomous system number, etc., by querying other data sources. Taggers then convert each annotated packet header into a JSON object and feed it into Scuba [10], a real-time data analytics system. Samples are simultaneously stored into Hive [120] tables for long-term analysis.

**Port mirroring**

While Fbflow is a powerful tool for network monitoring and management, its sampling-based collection prohibits certain types of data analysis. Specifically, in production use, it aggregates statistics at a per-minute granularity. In order to collect high-fidelity data, we deploy a number of special-purpose trace collection machines within the datacenter that collect packet-header traces over short intervals.

We deploy monitoring hosts in five different racks across Facebook's datacenter network, locating them in clusters that host distinct services. In particular, we monitor a rack of Web servers, a Hadoop node, cache followers and leaders, and a Multifeed node. In all but one (Web) instance, we collect traces by turning on port mirroring on the ToR and mirroring the full, bi-directional traffic for a single server to our collection server. For the hosts we monitor, the ToR is able to mirror the selected ports without loss. In the case of Web servers, utilization is low enough that we are able to mirror traffic from a rack of servers to our collection host. We did not measure database servers that include user data in this study.

Recording packet traces using a commodity server is not entirely trivial, as `tcpdump` is unable to handle more than approximately 1.5 Gbps of traffic in our configuration. In order to support line-rate traces, we employ a custom kernel module that effectively pins all free RAM on the server and uses it to buffer incoming packets. Our module extracts packets immediately after the Ethernet driver hands them to the kernel to avoid additional delays or overhead. Once data collection is complete, the data is spooled to remote storage for analysis. Memory restrictions on our collection servers limit the traces we collect in this fashion to a few minutes in length.

**Limitations**

Our methodology imposes a few limitations on the scope of this study. Using end hosts to capture and timestamp packets introduces scheduler-based variations on timestamp accuracy. In addition, we can only capture traffic from a few hosts at a time without risking drops in packet collection. Together, these constraints prevent us from evaluating effects like incast or microbursts, which are noted as being contributors to poor application performance [72]. Further, per-host packet dumps are necessarily anecdotal and ad hoc, relying on the presence of an unused capture host on the same rack as the target. While Fbflow is deployed datacenter-wide, the sheer amount of measurement data it provides presents another challenge—specifically, one of data processing and retention—which limits the resolution at which it can operate.

## 4.4 Provisioning

The appropriate design, scale, and technology of a datacenter interconnect depends heavily on the traffic demands of hosted services. Here, we quantify the traffic intensity, locality, and stability across three different types of clusters inside Facebook datacenters; in particular, we examine clusters supporting Hadoop, Frontend machines serving Web requests, and Cache.

Our study reveals that while Facebook's Hadoop deployments exhibit behavior largely consistent with the literature, the same cannot be said for clusters hosting Facebook's other services. In particular, most traffic is *not* rack-local, yet locality patterns remain stable within

and across both long (multiple-day) and short (two-minute) time intervals. We define stable traffic as being close to constant (low deviation from a baseline value) over a time interval, and slowly changing across time intervals. Note that this definition is dependent upon the length of the interval being considered; accordingly, we examine several different timescales.

### 4.4.1 Utilization

Given that Facebook has recently transitioned to 10-Gbps Ethernet across all of their hosts, it is not surprising that overall access link (i.e., links between hosts and their ToR) utilization is quite low, with the average 1-minute link utilization less than 1%. This observation comports with the utilization levels reported for other cloud-scale datacenters [30, 47]. Demand follows typical diurnal and day-of-the-week patterns, although the magnitude of change is on the order of 2× as opposed to the order-of-magnitude variation reported elsewhere [30], Even the most loaded links are lightly loaded over 1-minute time scales: 99% of all links are typically less than 10% loaded. Load varies considerably across clusters, where the average link utilization in the heaviest clusters (Hadoop) is roughly 5× clusters with light load (Frontend).

As in other datacenters with similar structure [30, 31], utilization rises at higher levels of aggregation. Focusing on the links between ToRs and CSWs, median utilization varies between 10–20% across clusters, with the busiest 5% of the links seeing 23–46% utilization. These levels are higher than most previously studied datacenters [30, Fig. 9], likely due to the disproportionate increase in edge-link technology (1→10 Gbps) vs. aggregation links (10→40 Gbps). The variance between clusters decreases, with the heaviest clusters running 3× higher than lightly loaded ones. Utilization is higher still on links between CSWs and FC switches, although the differences between clusters are less apparent because different clusters are provisioned with different numbers of uplinks depending on their demand. We examine link utilization at finer timescales in Section 4.6.

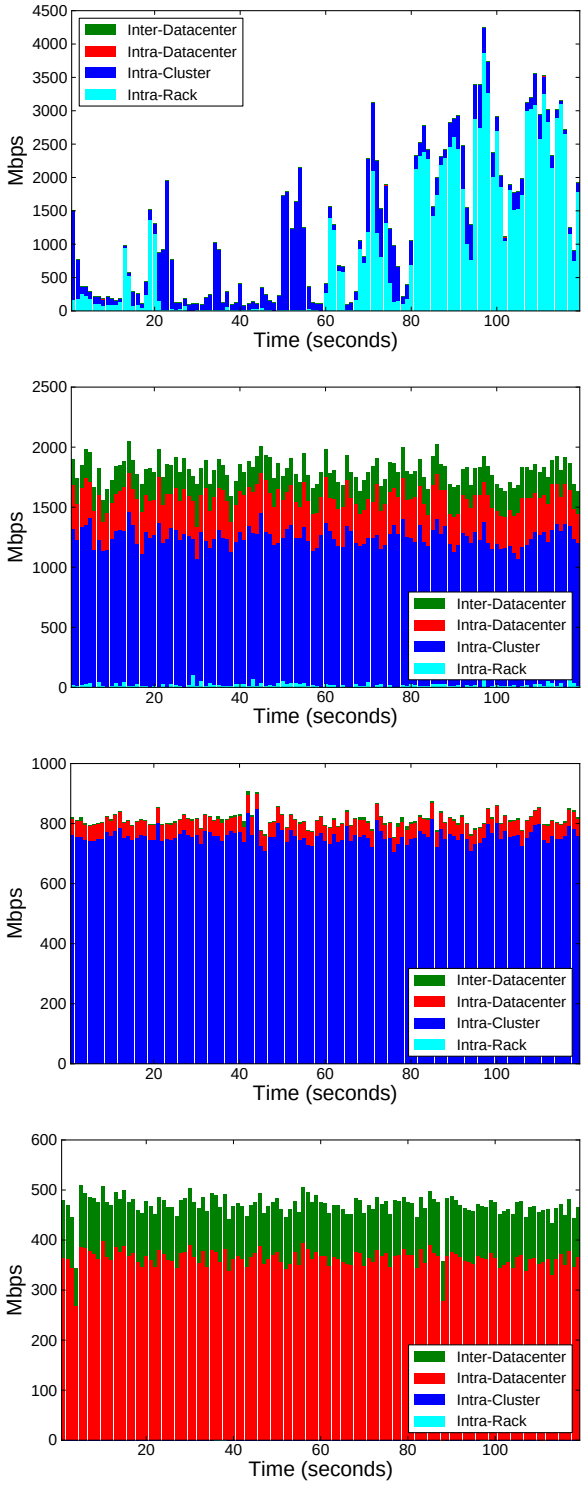



**Figure 4.4.** Per-second traffic locality by system type over a two-minute span: Hadoop (top left), Web server (top right), cache follower (bottom left) and leader (bottom right) (Note the differing *y* axes)

### 4.4.2 Locality and stability

Prior studies have observed heavy rack locality in datacenter traffic. This behaviour seems in line with applications that seek to minimize network utilization by leveraging locality, allowing for topologies with high levels of oversubscription. We examine the locality of Facebook's traffic from a representative sampling of production systems across various times of the day.

Figure 4.4 shows the breakdown of outbound traffic by destination for four different classes of servers: a Hadoop server within a Hadoop cluster, a Web server in a Frontend cluster, and both a cache follower and a cache leader from the same Cache cluster. For each server, each second's traffic is represented as a stacked bar chart, with rack-local traffic in cyan, the cluster-local traffic in blue, the intra-datacenter traffic in red, and inter-datacenter traffic in green.

Among the four server types, Hadoop shows the most diversity—both across servers and time: some traces show periods of significant network activity while others do not. While all traces show both rack- and cluster-level locality, the distribution between the two varies greatly. In one ten-minute-long trace captured during a busy period, 99.8% of all traffic sent by the server in Figure 4.4 is destined to other Hadoop servers: 75.7% of that traffic is destined to servers in the the same rack (with a fairly even spread within the rack); almost all of the remainder is destined to other hosts within the cluster. Only a vanishingly small amount of traffic is inter-cluster.

In terms of dispersion, of the inter-rack (intra-cluster) traffic, the Hadoop server communicates with 1.5% of the other servers in the cluster—spread across 95% of the racks—though only 17% of the racks receive over 80% of the server's traffic. This pattern is consistent with that observed by Kandula *et al.* [77], in which traffic is either rack-local or destined to one of roughly 1–10% of the hosts in the cluster.

Hadoop's variability is a consequence of a combination of job size and the distinct phases that a Hadoop job undergoes—any given data capture might observe a Hadoop node during a busy period of shuffled network traffic, or during a relatively quiet period of computation.

By way of contrast, the traffic patterns for the other server classes are both markedly more stable and dramatically different from the findings of Kandula *et al.* [77]. Notably, only a minimal amount of rack-local traffic is present; even inter-datacenter traffic is present in larger quantities. Frontend cluster traffic, including Web servers and the attendant cache followers, stays largely within the cluster: 68% of Web server traffic during the capture plotted here stays within the cluster, 80% of which is destined to cache systems; the Multifeed systems and the SLB servers get 8% each. While miscellaneous background traffic is present, the volume of such traffic is relatively inconsequential. As we discuss in Section 4.7, both the lack of rack locality and the temporal stability of traffic will benefit outlier-analysis based partial fault detection.

Cache systems, depending on type, see markedly different localities, though along with Web servers the intra-rack locality is minimal. Frontend cache followers primarily send traffic in the form of responses to Web servers (88%), and thus see high intra-cluster traffic—mostly servicing cache reads. Due to load balancing (see Section 5.2), this traffic is spread quite widely; during this two-minute interval the cache follower communicates with over 75% of the hosts in the cluster, including over 90% of the Web servers. Cache leaders maintain coherency across clusters and the backing databases, engaging primarily in intra- and inter-datacenter traffic—a necessary consequence of the cache being a ”single geographically distributed instance.” [34]

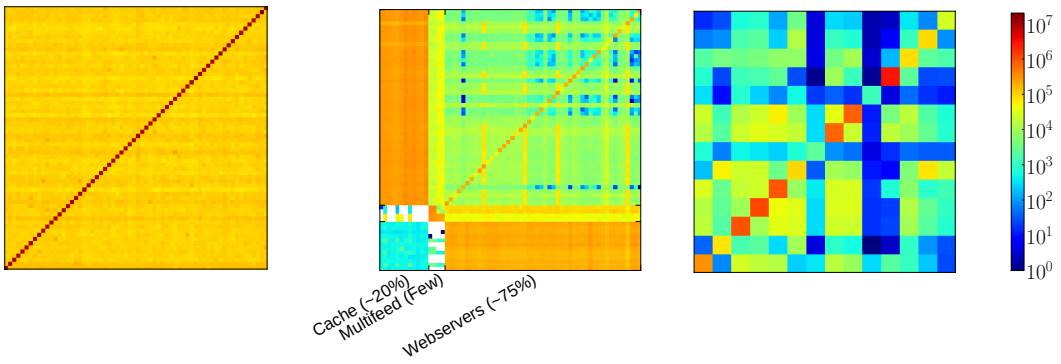
The stability of these traffic patterns bears special mention. While Facebook traffic is affected by the diurnal traffic pattern noted by Benson *et al.* [30], the relative proportions of the locality do not change—only the total amount of traffic. Over short enough periods of time, the graph looks essentially flat and unchanging. In order to further investigate the cause and particulars of this stability, we turn our attention to the traffic matrix itself.

### 4.4.3 Traffic matrix

In light of the surprising lack of rack locality and high degree of traffic stability, we examine traffic from the more long-term and zoomed-out Fbflow perspective. Table 4.3 shows the locality of traffic generated by all of Facebook’s machines during a 24-hour period in January

**Table 4.3.** Different clusters have different localities; last row shows each cluster's contribution to total network traffic

| Locality | All | Hadoop | FE | Svc. | Cache | DB |
|---|---|---|---|---|---|---|
| Rack | 12.9 | 13.3 | 2.7 | 12.1 | 0.2 | 0 |
| Cluster | 57.5 | 80.9 | 81.3 | 56.3 | 13.0 | 30.7 |
| DC | 11.9 | 3.3 | 7.3 | 15.7 | 40.7 | 34.5 |
| Inter-DC | 17.7 | 2.5 | 8.6 | 15.9 | 16.1 | 34.8 |
| Percentage | | 23.7 | 21.5 | 18.0 | 10.2 | 5.2 |




**(a)** Rack-to-rack, Hadoop cluster

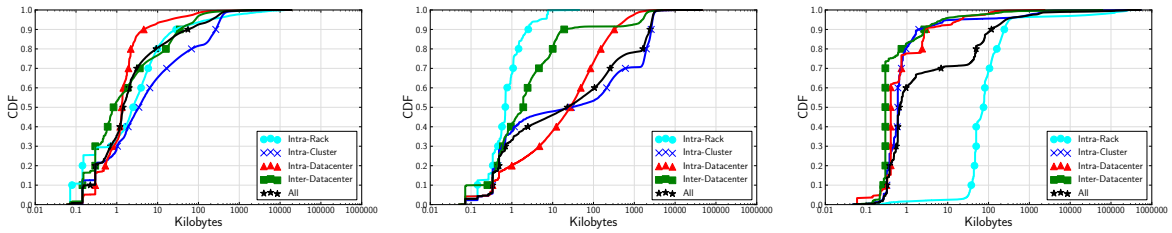
**(b)** Rack-to-rack, Frontend cluster

**(c)** Cluster-to-cluster

**Figure 4.5.** Traffic demand by source ($x$ axis) and destination ($y$ axis). The graphs are each normalized to the lowest demand in that graph type (i.e., the Hadoop and Frontend clusters are normalized to the same value, while the cluster-to-cluster graph is normalized independently).

2015 as reported by Fbflow. Facebook's traffic patterns remain stable day-by-day—unlike the datacenter studied by Delimitrou *et al.* [47]. The clear majority of traffic is intra-cluster but not intra-rack (i.e., the 12.9% of traffic that stays within a rack is not counted in the 57.5% of traffic labeled as intra-cluster). Moreover, more traffic is inter-datacenter than intra-rack.

Table 4.3 further breaks down traffic locality of the top-five cluster types which, together, account for 78.6% of Facebook datacenter traffic. Hadoop clusters generate the most traffic (23.7% of all traffic), and are significantly more rack-local than others, but even its traffic is far from the 40–80% rack-local reported in the literature [30, 47]. Instead, Hadoop is cluster local. Frontend (FE) traffic is cluster local by design, but not very rack-local, and the locality of a given rack's traffic depends on its constituent servers (e.g., Web, Multifeed, or cache).



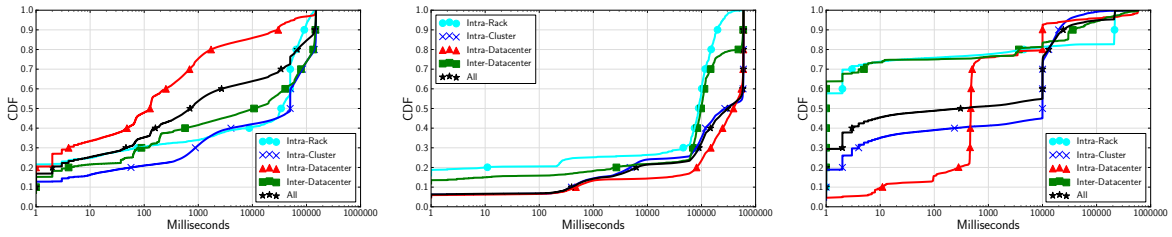

**(a)** Web servers **(b)** Cache follower **(c)** Hadoop

**Figure 4.6.** Flow size distribution, broken down by location of destination

This distinction is clearly visualized in Figure 4.5, generated in the style of Delimitrou *et al.* [47]. The two left portions of the figure graph the relative traffic demands between 64 racks within clusters of two different types. While we show only a subset of the total set of racks in each cluster, the pattern is representative of the cluster as a whole.

Traffic within the Hadoop cluster (left) is homogeneous with a very strong diagonal (i.e., intra-rack locality). The cluster-wide uniformity outside the local rack accounts for intra-cluster traffic representing over 80% of Hadoop traffic—even though traffic to the local rack dominates any given other rack in isolation. Map tasks are placed to maximize read locality, but there are a large number of concurrent jobs which means that it is possible that any given job will not fit entirely within a rack. Thus, some amount of traffic would necessarily need to leave the rack during the shuffle and output phases of a MapReduce job. In addition, the cluster serves data requests from other services which might not strive for as much read locality, which would also contribute to reduced overall rack locality.

The Frontend cluster (center) exhibits three different patterns according to rack type, with none being particularly rack-local. In particular, we see a strong bipartite traffic pattern between the Web servers and the cache followers in Webserver racks that are responsible for most of the traffic, by volume, within the cluster. This pattern is a consequence of placement: Web servers talk primarily to cache servers and vice versa, and servers of different types are deployed in distinct racks, leading to low intra-rack traffic.




**(a)** Web servers **(b)** Cache follower **(c)** Hadoop

**Figure 4.7.** Flow duration distribution, broken down by location of destination

This striking difference in Facebook's locality compared to previously studied Internet-facing applications is a consequence of the realities of serving a densely-connected social graph. Cache objects are replicated across clusters; however, each object typically appears once in a cluster (though objects may be replicated to avoid hot spots, which we discuss in Section 4.5). Since each Web server needs to be able to handle any request, they might need to access data in a potentially random fashion due to load balancing. Section 4.7 discusses how this traffic matrix, and in particular service load-balancing, aid outlier-analysis-based partial-fault detection.

To make this argument more concrete, loading the Facebook news feed draws from a vast array of objects in the social graph: different people, relationships, and events comprise a large graph interconnected in a complicated fashion. This connectedness means that the working set is unlikely to reduce even if users are partitioned; the net result is a low cache hit rate within the rack, leading to high intra-cluster traffic locality. In addition, partitioning the graph such that users and their data are co-located on racks has the potential to introduce failure modes which disproportionately target subsets of the user base, leading to a suboptimal experience.

The other three cluster types exhibit additional distinctive behaviors (not shown). Traffic in cache leader clusters, for example, has very little intra-rack demand, instead spreading the plurality of its traffic across the datacenter. Traffic in back-end database clusters is the most uniform, divided almost evenly amongst nodes within the cluster, the same datacenter, and worldwide. Service clusters, which host racks supporting a variety of supporting services, exhibit a mixed traffic pattern that lies between these extreme points.

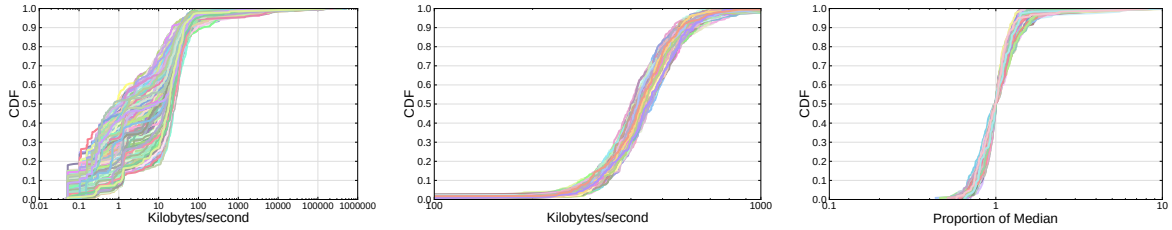
Inter-cluster communication varies considerably by cluster type. Figure 4.5c plots the traffic demand between 15 clusters within a single datacenter for the a 24-hour period. Hadoop clusters, for example, have a very small proportion of inter-cluster traffic, while cache leader clusters have a large amount of inter-cluster traffic, split between cache followers in other clusters and database clusters. While each cluster may possess the same four-post structure internally, it may make sense to consider heterogeneous inter-cluster communication fabrics, as demand varies over more than seven orders of magnitude between cluster pairs.

While the 4-post cluster remains prevalent in Facebook datacenters, Facebook recently announced a new network topology that is being implemented in datacenters going forward [19]. While servers are no longer grouped into clusters physically (instead, they comprise pods where all pods in a datacenter have high connectivity), the high-level logical notion of a cluster for server management purposes still exists to ease the transition. Accordingly, the rack-to-rack traffic matrix of a Frontend "cluster" inside one of the new Fabric datacenters over a day-long period (not shown) looks similar that shown in Figure 4.5.

### 4.4.4 Implications for connection fabrics

Low network-edge utilization levels reinforce common practice of oversubscribing the aggregation and core of the network, although it remains to be seen whether utilization will creep up as the datacenters age. The highly contrasting locality properties of the different clusters imply a single homogeneous topology will either be over-provisioned in some regions or congested in others—or both. This reality argues that non-uniform fabric technologies that can deliver higher bandwidth to certain locations than others may find use. Researchers are exploring techniques to ameliorate traffic hot spots. The stability of the traffic patterns we observe, however, suggest that rapid reconfigurability may not be as necessary as some have assumed.

Somewhat surprisingly, the lack of significant levels of intra-rack locality (except in the Hadoop cluster) hints that ToRs that deliver something less than full non-blocking line-rate connectivity between all of their ports may be viable. In particular, the bipartite traffic pattern




**(a)** Hadoop (rate, each series is 1 second)

**(b)** Cache (rate, each series is 1 second)

**(c)** Cache (stability, each series is a rack)

**Figure 4.8.** Per-destination-rack flow rate distribution (for both Hadoop and cache) and stability (cache).

between end hosts and ToR uplinks may afford optimizations in switch design. We return to consider further implications for switch design in Section 4.6.

## 4.5 Traffic engineering

Prior studies suggest that datacenter traffic stability depends on observation timescale. Here, we analyze Facebook's traffic at fine timescales, with an eye towards understanding how applicable various traffic engineering approaches may be under such conditions.

### 4.5.1 Flow characteristics

Figures 4.6 and 4.7 plot the size and duration, respectively, of flows (defined by 5-tuple) collected in 10-minute (2.5-minute for the Web-server rack) packet traces of three different node types: a Web-server rack, a single cache follower (cache leader is similar to follower and not shown due to space constraints), and a Hadoop node. We show the overall distribution (in black) as well as per-destination curves. Consistent with the literature [77, Fig. 9], we find that most flows in Facebook's Hadoop cluster are short. As discussed previously, the traffic demands of Hadoop vary substantially across nodes and time. We plot the results from tracing one node over a relatively busy 10-minute interval; traces from other nodes or even the same node at different times reveal somewhat different distributions, so we caution against examining the specific distribution too carefully. Even in the graphed interval, however, 70% of flows send less

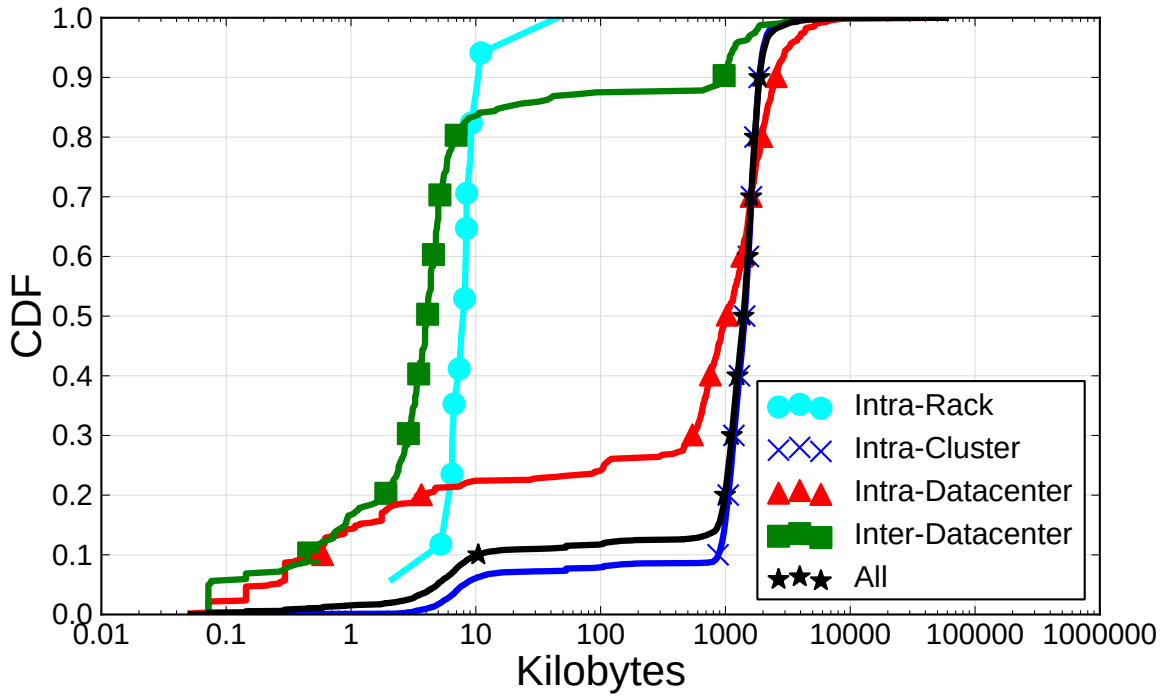



**Figure 4.9.** Cache follower per-host flow size

than 10 KB and last less than 10 seconds; the median flow sends less than 1 KB and lasts less than a second. Less than 5% of the flows are larger than 1 MB or last longer than 100 seconds; almost none exceed our 10-minute trace.

Conversely, traces from other service types are much more representative due to load balancing. Moreover, many of Facebook's internal services use some form of connection pooling [91], leading to long-lived connections with relatively low throughput. Pooling is especially prevalent for cache follower(leader, not shown) nodes, where only 30(40)% of flows are less than 100 seconds in length, with more than 40(25)% of flows exceeding our 10-minute capture period. That said, most flows are active (i.e., actually transmit packets) only during distinct millisecond-scale intervals with large intervening gaps. In other words, regardless of flow size or length, flows tend to be internally bursty. In general cache flows are also significantly larger than Hadoop; Web servers lie somewhere in the middle.

If we consider higher aggregation levels, i.e., grouping flows by destination host or rack, the flow size distributions simply shift to the right for Web servers (retaining its basic shape). The behavior is starkly different for cache followers, however: the wide flow-size distribution apparent at a 5-tuple granularity (Figure 4.6b) disappears at host and rack levels, replaced by a very tight distribution around 1 MB per host (Figure 4.9). This distribution arises as a consequence of the decision to load balance incoming user requests across all Web servers, combined with large numbers of requests. Since requests and responses are typically small (on the order of a few kilobytes) we do not observe any imbalance created by unequal response sizes. As we discuss in Section 4.7, this significantly aids outlier-analysis based partial fault detection.

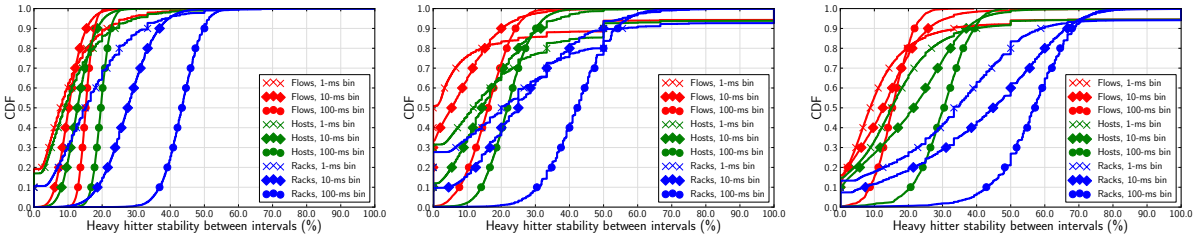
### 4.5.2 Load balancing

Existing traffic engineering efforts seek to leverage variability of traffic; highly regular traffic does not provide much opportunity for improvement. In the previous section, we note that Facebook's approach to load balancing is highly effective on timescales lasting minutes to hours, leaving less room for traffic engineering. We now consider traffic characteristics over the course of a few seconds to determine whether traffic engineering might be effective on short timescales.

Specifically, we consider how the traffic from a host varies from one second to next. We examine the distribution of flow rates, aggregated by destination rack, per second over a two-minute period and compare each second to the next. Intuitively, the better the load balancing, the closer one second appears to the next.

We first examine the Hadoop cluster by looking at 120 consecutive 1-second intervals. Figure 4.8a plots a CDF of per-destination-rack flow sizes for each interval (i.e., there are 120 separate curves). While we do not claim this particular server is representative, it does depict widely varying rates (i.e., over three orders of magnitude) which are common in our observations.

In and of itself, this is unsurprising—Hadoop has periods of varying network traffic, and a production cluster is likely to see a myriad jobs of varying sizes. It is this variability of traffic that existing network traffic engineering schemes seek to leverage. Orchestra [41] relies




**(a)** Cache follower **(b)** Cache leader **(c)** Web servers

**Figure 4.10.** Heavy-hitter stability as a function of aggregation for 1/10/100-ms time windows

on temporal and per-job variation to provide lower task completion times for high-priority tasks, while Hedera [15] provides non-interfering route placement for high bandwidth elephant flows that last for several seconds, which are prevalent within Hadoop workloads.

A different story emerges for Frontend traffic, and the cache in particular. Recall from Table 4.2 that the largest share of cache follower traffic are responses bound for Web servers. Figure 4.8b shows the distribution of per-second flow rates on a per-rack basis from a single cache follower node to distinct Web server racks during a two minute period. The distributions for each of the 120 seconds are similar, and all are relatively tight, i.e., the CDFs are fairly vertical about the median of $\approx$2 Mbps. Similar patterns (albeit with different scales) can be observed for other services as well.

From the viewpoint of a single host, each second is similar to the next. However, this analysis does not take per-destination variation into consideration. It is conceivable that there could exist consistently high- or low-rate destinations that potentially could be treated differently by a traffic engineering scheme. For each host, we consider outbound traffic rates per destination rack (normalized to the median rate for that rack), and track the rate over time for each rack. Figure 4.8c plots these distributions for the outbound traffic for the same cache machine as Figure 4.8b. Each series represents a single destination; a near vertical series represents a destination rack where the rate does not deviate far from the median rate. We find that per-destination-rack flow sizes are remarkably stable across not only seconds but intervals as long as 10 seconds (not shown) as well. All of the flows are within a factor of two of their median

size in approximately 90% of the 1-second intervals—the median flow exhibits "significant change" in only 45% of the 1-second intervals according to the 20% deviation cutoff defined by Benson *et al.* [32]. Contrast this to the traffic leaving a Hadoop node—which is not load balanced—where the middle 90% of flows can vary in size by over six orders of magnitude compared to their median size in the trace (not shown).

Such stability, both over time and by destination, is the result of a combination of workload characteristics and engineering effort. To a cache system, the offered load per second is roughly held constant—large increases in load would indicate the presence of relatively hot objects, which is actively monitored and mitigated. Bursts of requests for an object lead the cache server to instruct the Web server to temporarily cache the hot object; sustained activity for the object leads to replication of the object or the enclosing shard across multiple cache servers to help spread the load. We note further that the request rate distribution for the top-50 most requested objects on a cache server is close across all cache servers, and that the median lifespan for objects within this list is on the order of a few minutes. Per-destination traffic stability is again a consequence of user request multiplexing across all available Web servers, coupled with relatively small request/response pairs. Thus, in addition to a dearth of spatial hot spots, Facebook traffic tends to lack temporal hot spots. The lack of hot spots will have consequences for outlier-analysis based partial fault detection, as we discuss in Section 4.7.

### 4.5.3 Heavy hitters

Here, we examine traffic behavior at sub-second timescales to better understand its stability and whether traffic engineering can apply. We wish to see if certain flow (or aggregated) data rates stand out, since such flows provide the largest potential network performance impact. We define a set of flows called *heavy hitters* as the minimum set of flows (or hosts, or racks in aggregate) responsible for 50% of observed traffic volume (in bytes) over a fixed time period. Intuitively, heavy hitters signify imbalances that can be acted upon—if persistent for enough time, and large enough compared other flows that treating them differently makes a difference.

**Table 4.4.** Number and size of heavy hitters in 1-ms intervals for each of flow(f), host(h), and rack(r) levels of aggregation.

| Type | | Number | | | Size (Mbps) | | |
|---|---|---|---|---|---|---|---|
| | | p10 | p50 | p90 | p10 | p50 | p90 |
| **Web** | f | 1 | 4 | 15 | 1.6 | 3.2 | 47.3 |
| | h | 1 | 4 | 14 | 1.6 | 3.3 | 48.1 |
| | r | 1 | 3 | 9 | 1.7 | 4.6 | 48.9 |
| **Cache (f)** | f | 8 | 19 | 35 | 5.1 | 9.0 | 22.5 |
| | h | 8 | 19 | 33 | 8.4 | 9.7 | 23.6 |
| | r | 7 | 15 | 23 | 8.4 | 14.5 | 31.0 |
| **Cache (l)** | f | 1 | 16 | 48 | 2.6 | 3.3 | 408 |
| | h | 1 | 8 | 25 | 3.2 | 8.1 | 414 |
| | r | 1 | 7 | 17 | 5 | 12.6 | 427 |
| **Hadoop** | f | 1 | 2 | 3 | 4.6 | 12.7 | 1392 |
| | h | 1 | 2 | 3 | 4.6 | 12.7 | 1392 |
| | r | 1 | 2 | 3 | 4.6 | 12.7 | 1392 |

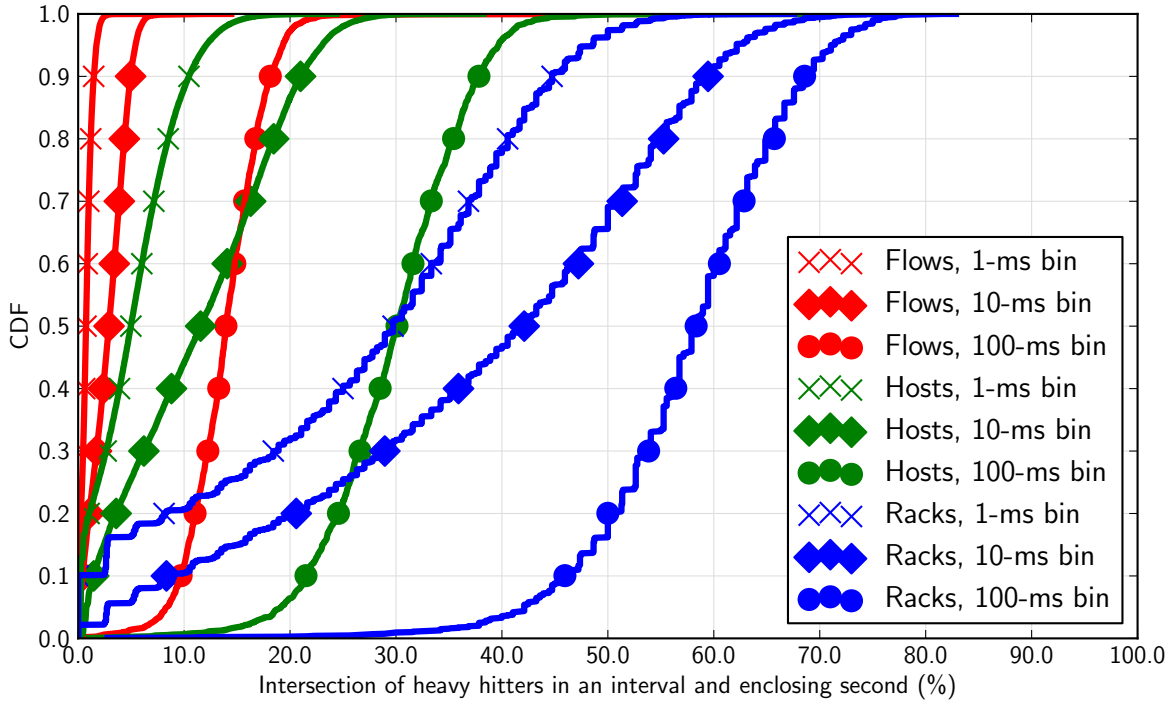
Table 4.4 shows statistics regarding the number and size of heavy hitters constituting 50% of the traffic in 1-ms intervals for each of the four server classes. Because we are interested in instantaneously large flows, we measure size in terms of rate instead of bytes sent over the flow's lifetime. Next, we consider the the lifespan of heavy hitters, aggregated by 5-tuple, destination host and rack, and measured across intervals of 1, 10 and 100 milliseconds. Figure 4.10 shows the fraction of the heavy hitters that remain in subsequent time intervals. We do not show the Hadoop nodes, as our heavy-hitter definition almost always results in the identification of 1–3 heavy hitters at each of flow, host, and rack aggregation levels across all three time intervals.

Heavy hitter persistence is low for individual flows (red): in the median case, no more than $\approx 15\%$ of flows persist regardless of interval, a consequence of the internal burstiness of flows noted earlier. Host-level aggregation (green) fares little better; with the exception of destination-host-level aggregation for Web servers, no more than 20% of heavy hitter hosts in a sub-second interval will persist as a heavy hitter in the next interval. Web servers have a higher rate over 100-millisecond periods since they have a relatively small set of caches and load balancers with which they communicate, while caches converse with many different Web servers. We discuss ramifications for outlier-analysis based partial fault detection in Section 4.7.

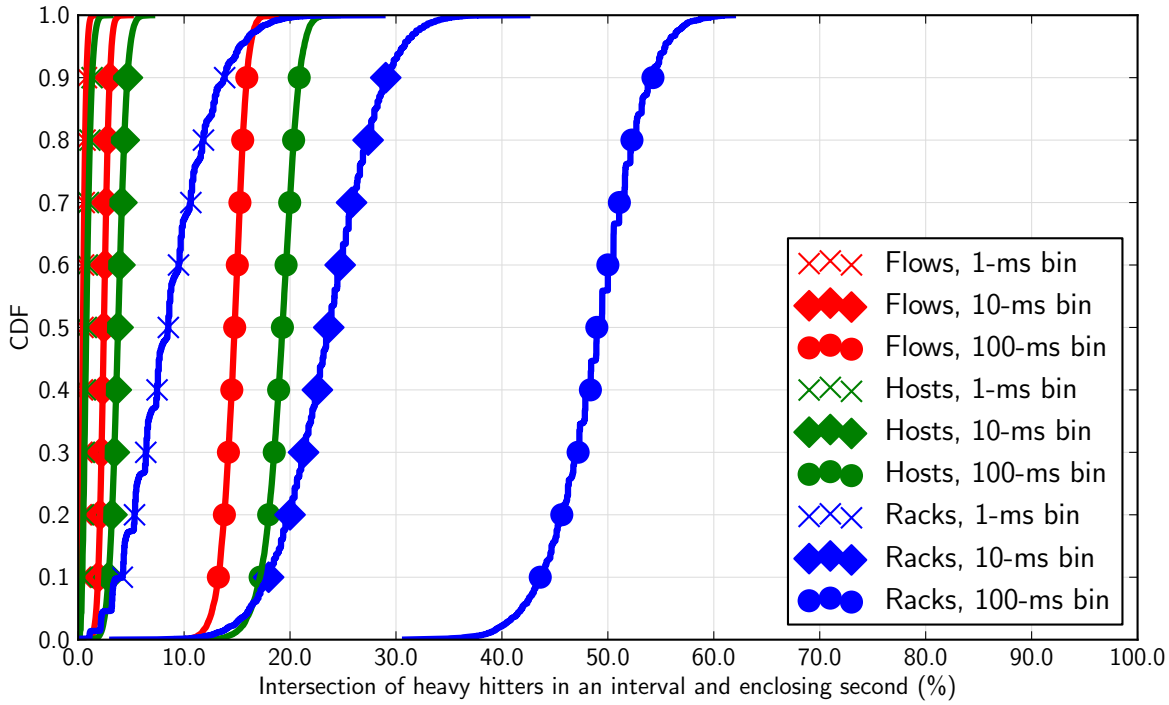
It is not until considering rack-level flows (blue) that heavy hitters are particularly stable. In the median case, over 40% of cache heavy hitters persist into the next 100-ms interval, and almost 60% for Web servers. Heavy hitters from Web servers are more stable in general, with 32% of rack-level heavy hitters persisting in the median 1-ms interval case. Even so, heavy hitter persistence is not particularly favorable for traffic engineering. With a close to 50% chance of a given heavy hitter continuing in the next time period, predicting a heavy hitter by observation is not much more effective than randomly guessing.

Even if one could perfectly predict heavy hitters on a second-by-second timescale, it remains to consider how useful that knowledge would be. We compare heavy hitters from enclosing one-second intervals to instantaneous heavy hitters from each of the subintervals within the second to see what fraction of heavy hitters in a subinterval are heavy hitters across the entire enclosing second. A limited degree of overlap implies three things: First, it establishes an upper bound on the traffic-engineering effectiveness—a significant amount of ephemeral heavy hitter traffic would go unseen and untreated by the TE scheme. Second, it serves as an indicator of prediction difficulty; if a one-second prediction interval is not sufficient, smaller timescales (consuming more resources) may be needed. Finally, this metric is an indicator of burstiness, as it indicates the presence of a large number of ephemeral heavy hitters.

Figure 4.11 plots a CDF of the fraction of a second's overall heavy hitters that are instantaneously heavy in each 1/10/100-ms interval within the second. We show results for a Web server and cache follower—cache leaders are similar. At 5-tuple granularity, predictive power is quite poor, at less than 10–15%. Rack-level predictions are much more effective, with heavy hitters remaining heavy in the majority of 100-ms intervals in the median case for both services. Host-level predictions are more useful for Web servers than cache nodes, but only the 100-ms case is more than 30% effective.




**(a)** Web server




**(b)** Cache follower

**Figure 4.11.** Intersection between heavy hitters in a subinterval with enclosing second

### 4.5.4 Implications for traffic engineering

Facebook's extensive use of connection pooling leads to long-lived flows that seem like potential candidates for traffic engineering. These same services use application-level load balancing to great effect, however, leaving limited room for in-network approaches. Many existing techniques work by identifying heavy hitters and then treating them differently (e.g., provisioning a circuit, moving them to a lightly loaded path, employing alternate buffering strategies, etc.). For any such scheme to work, however, it must be possible to first identify the heavy hitters, and then realize some benefit.

Unfortunately, it appears challenging to identify heavy hitters in a number of Facebook's clusters that persist with any frequency. Moreover, even for the timescales and aggregation levels where it is possible (e.g., rack-level flows over intervals of 100-ms or larger), it is not clear there is a great deal of benefit to be gained, as the heavy hitters are frequently not particularly heavy for the vast majority of the period. Previous work has suggested traffic engineering schemes can be effective if 35% of traffic is predictable [32]; only rack-level heavy hitters reach that level of predictability for either Web or cache servers. This somewhat counter-intuitive situation results from a combination of effective load balancing (which means there is little difference in size between a heavy hitter and the median flow) and the relatively low long-term throughput of most flows, meaning even heavy flows can be quite bursty internally.

## 4.6 Switching

Finally, we study aspects of the traffic that bear directly on top-of-rack switch design. In particular, we consider the size and arrival processes of packets, and the number of concurrent destinations for any particular end host. In addition, we examine the impact of burstiness over short timescales and its impact on switch buffering.

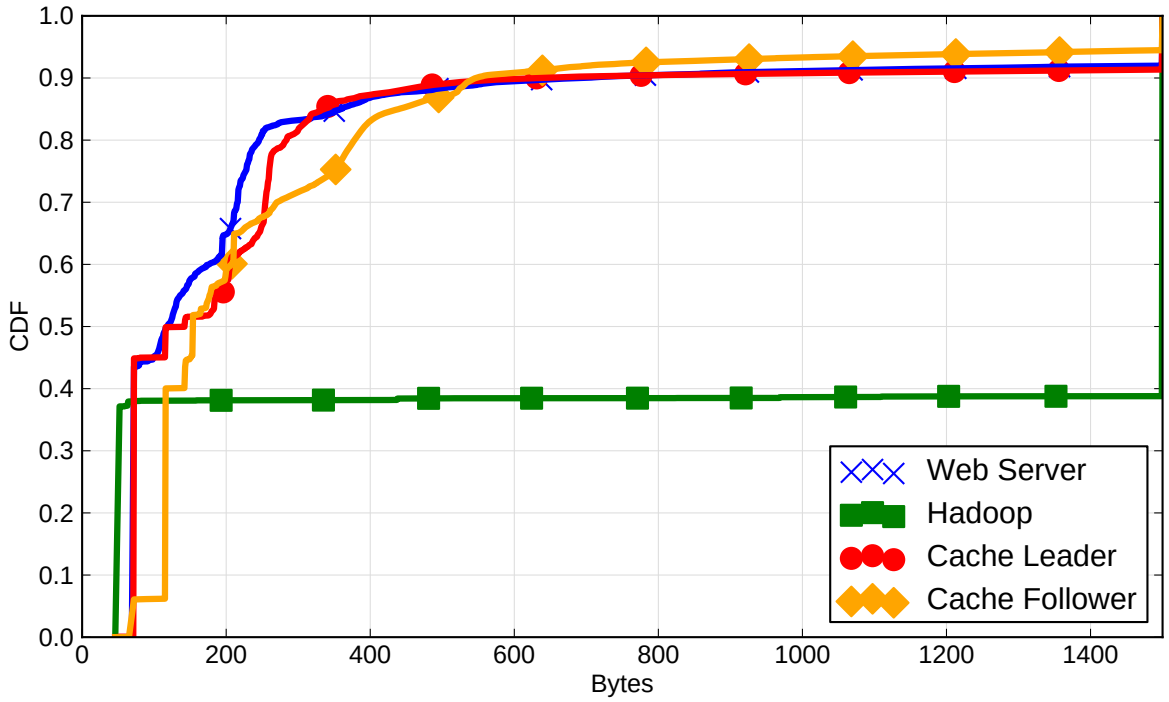
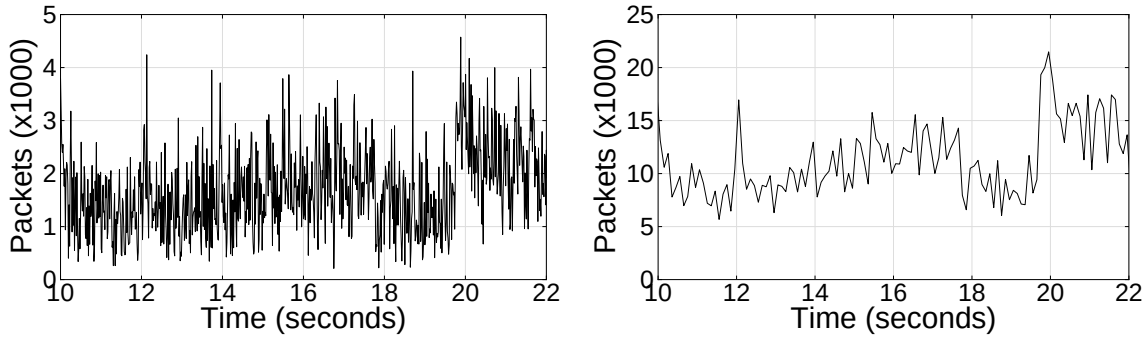



**Figure 4.12.** Packet size distribution

### 4.6.1 Per-packet features

Figure 4.12 shows the distribution of packet sizes for each of the four host types. Overall, the median packet size is approximately 250 bytes, but that is significantly skewed by Hadoop traffic. Hadoop traffic is bimodal: almost all packets are either MTU length (1500 bytes for the servers we study) or TCP ACKs. Packets for the other services have a much wider distribution, but the median packet size for all of them is significantly less than 200 bytes, with only 5–10% of the packets fully utilizing the MTU.

Thus, while link utilization is low, the packet rate is still high. For example, a cache server at 10% link utilization with a median packet size of roughly 175 bytes generates 85% of the packet rate of a fully utilized link sending MTU-sized packets. As a result, any per-packet operation (e.g., VLAN encapsulation) may still be stressed in a way that the pure link utilization rate might not suggest at first glance.




**(a)** 15 ms **(b)** 100 ms

**Figure 4.13.** Hadoop traffic is not on/off at 15 nor 100 ms

### 4.6.2 Arrival patterns

Benson *et al.* observe that packet arrivals exhibit an on/off pattern at the server level [30, 31]. Facebook servers do not exhibit this behavior, even within Hadoop clusters. Figure 4.13 shows a time series of traffic sent by a Hadoop host (arriving at a ToR port) binned by 15- and 100-ms intervals. (c.f. Benson *et al.*'s analogous graphs [31, Figure 5] and [30, Figure 6]). If one considers traffic on a per-destination host basis, on/off behavior reemerges (not shown), suggesting its disappearance may be due to a large number of concurrent destinations.

Figure 4.14 plots the outgoing TCP flow inter-arrival time CDF at each of the servers we study. While a significant amount of traffic is routed over long-lived pooled connections, as is the case for request-response traffic between Web servers and cache followers, ephemeral flows do exist. Flow inter-arrival periods from all four classes of host are shorter than those in the literature [77, Fig. 11], to varying degrees. Hadoop nodes and Web servers see an order-of-magnitude increase in flow intensity relative to previous reports—likely due at least in part to the 10× increase in link rate—with median inter-arrival times of approximately 2 ms (i.e., more than 500 flows per second). Perhaps due to connection pooling (which would decouple the arrival of external user requests from the internal flow-arrival rate), the distribution of inter-arrival times for flows at both types of cache node are similar and longer: cache leaders see a slightly higher arrival rate than followers, with median inter-arrival times of 3 and 8 ms, respectively.

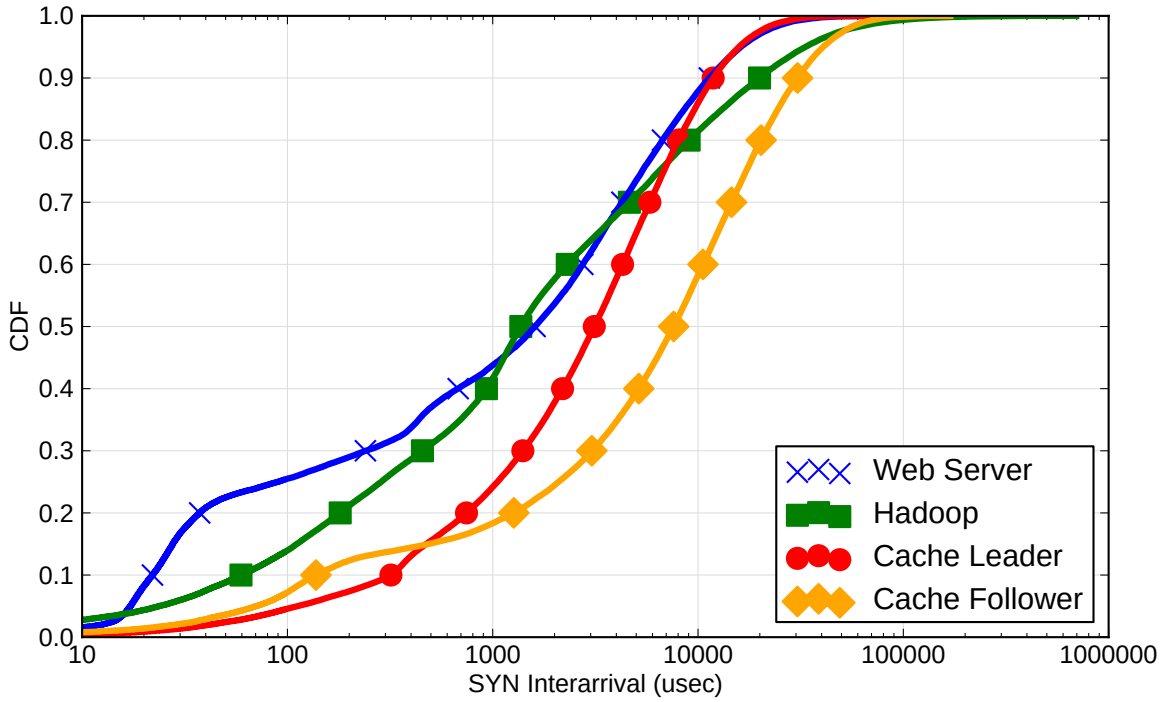
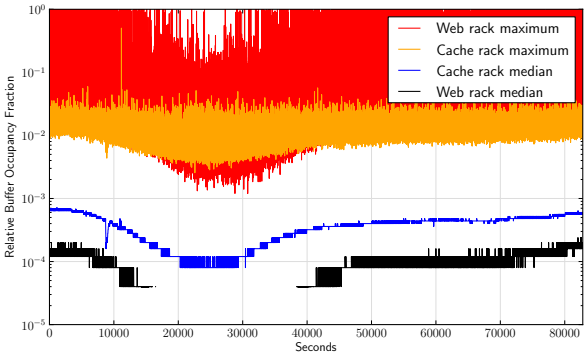


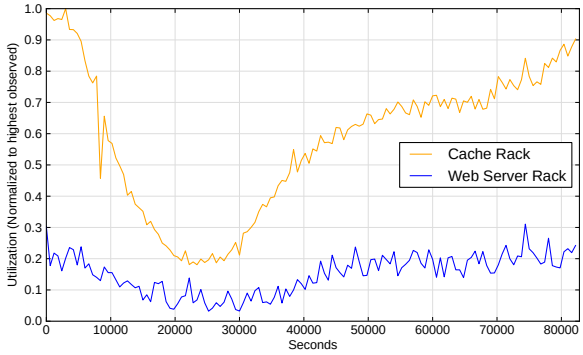

**Figure 4.14.** Flow (SYN packet) inter-arrival

### 4.6.3 Buffer utilization

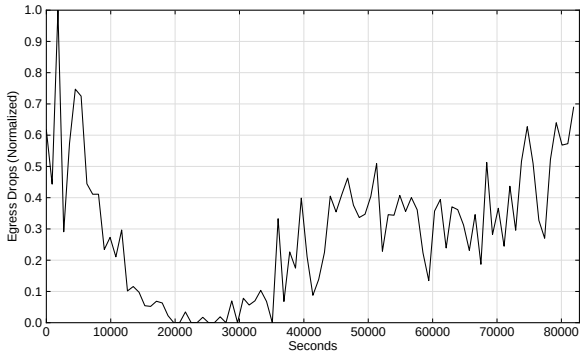
The combination of a lack of on/off traffic, higher flow intensity, and bursty individual flows suggests a potential increase in buffer utilization and overruns. Despite low average link utilization, bursty traffic can still lead to unacceptable loss rates. Recent work at Facebook has led to the development of in-house switching platforms [116], enabling us to gather buffer utilization statistics at fine granularity. In particular, we collect buffer occupancies over a 24-hour period for switches connecting Web servers and cache nodes at a 10-microsecond granularity. Figure 4.15a plots the median and maximum values per second for the entire period, normalized to the buffer size. In other words, a single point for the median series represents the $50^{\text{th}}$-percentile buffer occupancy during that second (out of 100,000 samples per second), normalized by the size of the buffer. We also plot the normalized average link utilization (Figure 4.15b) and egress drop rate (Figure 4.15c) over the same period, sourced via fbflow and SNMP counters, respectively.




**(a)** Normalized buffer occupancy, 10-microsecond resolution

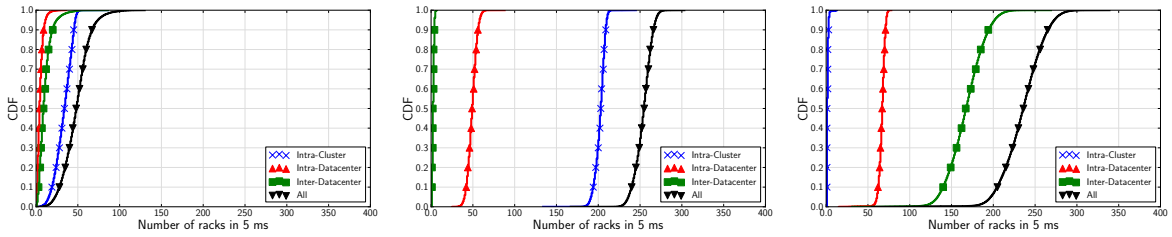



**(b)** Link utilization, 10-minute average




**(c)** Web rack egress drops, 15-minute average

**Figure 4.15.** Correlating buffer occupancy, link utilization and packet drops in Web server and Cache racks

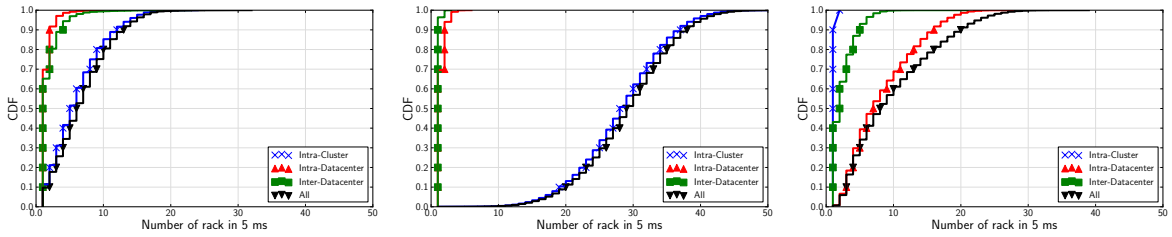



**(a)** Web server **(b)** Cache follower **(c)** Cache leader

**Figure 4.16.** Concurrent (5-ms) rack-level flows

A few trends are apparent from our results. The first is that standing buffer occupancies are non-trivial, and can be quite high in the Web-server case. Even though link utilization is on the order of 1% most of the time, over two-thirds of the available shared buffer is utilized during each 10-$\mu$s interval. Diurnal variation exists in buffer occupancies, utilization and drop rate, highlighting the correlation between these metrics over time. Even with the diurnal traffic pattern, however, the maximum buffer occupancy in the Web-server rack approaches the configured limit for roughly three quarters of the 24-hour period. While link utilization is roughly correlated with buffer occupancy within the Web-server rack, utilization by itself is not a good prediction of buffer requirements across different applications. In particular, the Cache rack has higher link utilization, but much lower buffer utilization and drop rates (not shown).

These buffer utilization levels occur despite relatively small packet sizes (Section 4.6.1). As utilization increases in the future, it might be through an increase in the number of flows, in the size of packets, or both. Either will have impacts on buffer utilization: larger packets with the same level of burstiness will use up more of the buffer, while a larger number of flows leads to a greater chance of multiple flows sending bursts of packets simultaneously. Thus, careful buffer tuning is likely to be important moving forward. High buffer utilization also has ramifications for outlier-analysis based partial fault detection, as we discuss in Section 4.7.




**(a)** Web server **(b)** Cache follower **(c)** Cache leader

**Figure 4.17.** Concurrent (5-ms) heavy-hitter racks

### 4.6.4 Concurrent flows

We consider concurrent to mean existing within the same 5-ms window (c.f. the 50-ms window considered by Alizadeh *et al.* while measuring a datacenter of hosts connected with 1-Gbps Ethernet [18]). We find that Web servers and cache hosts have 100s to 1000s of concurrent connections (at the 5-tuple level), while Hadoop nodes have approximately 25 concurrent connections on average—corresponding quite well with the findings of Alizadeh *et al.* [18, Figure 5]. That said, switches are obviously less concerned with individual connections than destination ports. If we group connections destined to the same host, the numbers reduce only slightly (by at most a factor of two)—and not at all in the case of Hadoop. The large number of concurrent flows for Web and cache hosts, and the relatively small number of concurrent Hadoop flows, will impact the time constants involved for outlier-analysis based partial fault detection; we discuss these effects more in Section 4.7 and Chapter 5.

Given the general lack of intra-rack traffic, almost all flows will traverse an up-link port. Hence, it may be more interesting to consider rack-level flows—i.e., the number of different racks with which a server is communicating. Figure 4.16 shows the number of concurrent flows sent by a single host over a 5-ms interval to different classes of destination hosts for three different host types: cache follower, cache leader, Web server. Cache followers communicate with 225–300 different racks, while leaders talk to 175–350. In the median interval, both types of caches communicate with $\approx 250$ other racks—rack location varies dramatically as discussed previously, however. Web servers communicate with 10–125 racks concurrently, 50 in the median interval.

Some proposed switch designs [65, 92] employ different technologies for large flows. Hence, we restrict our focus to the heavy hitter racks, namely those destination racks that constitute the majority of the traffic. The median number of heavy-hitter racks is 6–8 for Web servers and cache leaders with an effective max of 20–30, while the cache follower has 29 heavy hitter racks in the median case and up to 50 in the tail. Due to the differences in locality, Web servers and cache followers have very few rack-level heavy hitters of their cluster, while the cache leader displays the opposite pattern. Even considering only heavy hitter racks, the number of concurrent destinations is still significantly larger than that reported by Alizadeh *et al.* [18]. In addition, the relative impermanence of our heavy hitters suggests that, for Frontend clusters at least, hybrid circuit-based approaches may be challenging to employ.

## 4.7 Implications for partial fault localization

In Chapter 3, we motivated an outlier-analysis based approach for partial fault detection and localization, noting that under certain circumstances—specifically, a multipath network with equal-cost paths, and a large body of traffic to spread over those paths equally—that we could pinpoint faults by looking for components that exhibited significantly worse performance than normal. We demonstrated how partial faults could cause noticeable performance outliers on a testbed matching these requirements; however, we noted that real world traffic characteristics could impact the effectiveness of our methodology. Here, we discuss how various characteristics of Facebook datacenters either help or hinder our approach.

**Facebook datacenters meet the base requirements for outlier analysis**

Both the four-post topology and the newer Fabric topologies provide multiple equal-cost paths between any pair of communicating servers. The lack of rack locality means that most communicating server pairs will send traffic over the core of the datacenter network over these paths. In addition, the large number of flows (on the order of 1000s of concurrent flows in typical 5-msec intervals for just a single server) coupled with RPC-style traffic with small packet sizes

($\leq$ 200 byte median packet size for non-Hadoop services) means that ECMP routing should yield very even mixes of traffic across links within each layer of the network. Thus, performance across these links will be roughly equivalent, as we show in Chapter 5.

**High buffer occupancies suggests bursty traffic and complicates outlier analysis**

Relatively high standing buffer occupancies in spite of low overall link utilization suggest the possibility of bursty traffic and attendant loss. As such, microburst behaviour has been noticed within Facebook datacenters [131]. The presence of 'ambient' packet loss has two ramifications for outlier-analysis based detection.

First, burst-driven packet loss admits the possibility of false-positives; for example, a burst on a link over a short time period could cause it to lose more packets than other links and trigger an outlier monitor. We must take care that this does not occur. Second, even if we can filter out false-positives, ambient packet loss may place an upper bound on the sensitivity of outlier-analysis based approaches to detecting anomalous packet loss. In other words, if we can ignore false-positives caused by ambient packet loss of a given degree, we risk not noticing actual anomalous loss if it is of roughly the same degree. We argue that this may not be a large downside—if the loss was not large enough to notice over the regular behaviour of the network, it may not have a significant enough impact on application traffic to matter. In Chapter 5, we see that outlier-analysis can reliably pinpoint lossy links when the degree of loss does outstrip ambient loss, while avoiding false-positive behaviour caused by ambient loss.

**Load balancing and the lack of hot spots will allow fast pinpointing of faults**

Facebook application-level load balancing effectively squashes spatial and temporal hot spots. Any given second look like any other second for a given host, both in terms of overall volume of traffic and the amount of traffic sent to a remote host. As a consequence, no clear persistent heavy hitter behaviour emerges; even if a given host suddenly receives a burst of traffic, long-standing imbalances in traffic volume do not occur.

As we will see in Chapter 5, this allows our outlier-analysis fault detection system to successfully aggregate data and output faulty components over small timescales of ranging from a few seconds to a few tens of seconds depending on fault magnitude. While Hadoop traffic has significant amount of temporal variation, we will see that outlier-analysis still works, if we increase the timescale that we operate on. In some sense, the emphasis on load-balancing means that Facebook datacenters act as a best-case scenario for outlier-analysis based techniques. In the next chapter, we present our fault-detection methodology and evaluate it at Facebook.

# Acknowledgments

Chapter 4 is an adapted reprint of the material as it appears in "Inside the Social Network's Datacenter Network", which was published in ACM SIGCOMM 2015. The dissertation author was the primary author of this paper. The paper was co-authored with Hongyi (James) Zeng and Jasmeet Bagga of Facebook, and George Porter and Alex C. Snoeren of UC San Diego.

The original work was supported in part by the National Science Foundation through grants CNS-1314921 and CSR-1018808. We are indebted to Theo Benson, Nick McKeown, Remzi Arpaci-Dusseau, our shepherd, Srikanth Kandula, and the anonymous reviewers for their comments and suggestions on earlier drafts of the originally published manuscript. Petr Lapukhov, Michal Burger, Sathya Narayanan, Avery Ching and Vincent Liu provided invaluable insight into the inner workings of various Facebook services. Finally, and most significantly, Omar Baldonado catalyzed and facilitated the collaboration that enabled the publication of the original manuscript.

# Chapter 5

# Simplified fault detection and localization

In prior chapters, we have introduced the challenges associated with datacenter partial fault localization, developed a set of criteria for evaluating localization methodologies, and discussed prior approaches and their success and shortcomings. Additionally, we have presented an intuitive argument for why an outlier-analysis based passive monitoring approach could be a viable method for localizing partial faults, and discussed how various characteristics of production Facebook datacenters could aid the effectiveness of such an approach.

In this chapter, we present this dissertation's main contribution. Specifically, we describe how to expedite the process of detecting and localizing partial faults within datacenter networks using an server based, passive-monitoring outlier-analysis methodology generalizable to most datacenter applications. In particular, we correlate transport-layer flow metrics and network-I/O system call delay at servers with the path that traffic takes through the datacenter and apply statistical analysis techniques to identify outliers and localize the faulty link and/or switch(es). We evaluate our approach in a production Facebook front-end datacenter.

At a high level, while network statistics can be noisy and confusing to interpret in isolation, the regular topology and highly engineered traffic present within Facebook's datacenters provides an opportunity to leverage simple statistical comparison-based methods to rapidly determine where partial faults occur as they happen.

To facilitate such a comparison, we develop a lightweight packet-marking technique—leveraging only forwarding rules supported by commodity switching ASICs—that uniquely identifies the full path that a packet traverses in a Facebook datacenter. Moreover, the topological regularity of Facebook's datacenter networks allows us to use path information to passively localize the fault as well. Because each server can bin flows according to the individual network elements they traverse, we can contrast flows traversing any given link (switch) at a particular level in the hierarchy with flows that traverse alternatives in order to identify the likely source of detected performance anomalies. Operators can then use the output of our system—namely the set of impacted traffic and the network element(s) seemingly responsible—in order to adjust path selection (e.g., through OpenFlow rules, ECMP weight adjustment [108], or tweaking inputs to flow hashes [75]) to mitigate the performance impact of the fault until they can repair it.

Our specific contributions include (1) a general-purpose, server-based monitoring scheme that robustly identifies flows traversing faulty network components, (2) a methodology to discover the necessary path information scalably in Facebook's datacenters, and (3) a system that leverages both types of information in aggregate to perform network-wide fault-localization.

We demonstrate that our technique is able to identify links and routers exhibiting low levels (0.25–1.0%) of packet loss within a Facebook datacenter hosting user-servicing front-end web and caching servers within 20 seconds of fault occurrence with a minimal amount of processing overhead. We also perform a sensitivity analysis on a testbed to consider different types of errors—including those that induce only additional latency and not loss—traffic patterns, application mixes, and other confounding factors.

The rest of this chapter is organized as follows. We present a formalization of our outlier-analysis based approach by defining 'Equivalence Sets', and describe how fault localization based on comparing elements on these sets differs significantly from prior approaches. Following this, we provide an overview of our specific fault localization system, describe its detailed implementation and evaluate it within a production Facebook datacenter. We conclude by discussing the limits of its applicability, focusing both on surmountable challenges and fundamental

shortcomings of the approach. Finally, we discuss how our equivalence set based approach satisfies the partial fault localization success criteria we presented earlier in this dissertation.

## 5.1 Formalizing outlier-analysis via equivalence sets

Outlier-analysis can be confounded by network traffic pattern hot spots, unless we pick the right components—links, switches, or ports, for example—to compare with each other. In particular, we previously showed that comparing traffic performance on a core-switch by core-switch basis clearly revealed the impact of partial faults on application traffic metrics.

Here, we formalize our approach by defining 'Equivalence Sets'. We claim that comparing the equivalence-set components will allow us to find partial faults and that the characteristics of such a set are both necessary and sufficient to apply outlier analysis to do so.

### 5.1.1 Defining equivalence sets

Given a collection of network flows, an Equivalence Set is a set of network components (links, switches, ECMP groups, Link Aggregation groups, etc.) such that:

1. The flows in question are randomly split amongst the components within the set. In other words, for a given collection of network flows, any flow has an exactly equal chance of being forwarded via any of the elements in the set.
2. The number of flows is significantly larger than the number of elements in the set. Thus, the distribution of traffic characteristics for each element in the set, while arbitrary, is roughly similar to the distribution for any other member in the set.
3. Each component in the set can service network traffic equivalent performance. For example, links in a set must provide the same bandwidth and have access to the same level of buffer resources. Thus, an ECMP group of four 1-Gbps links is valid, while a set with three 1-Gbps and one 10-Gbps link is not. Or, for example, if traffic can flow via multiple network middleboxes, they must possess comparable processing resources.

We claim that components within an equivalence set, in the absence of any faults, will provide equivalent performance to traffic traversing any component in the set. In other words, the aggregate distribution of relevant performance metrics (for example, packet loss or latency) will be similar on a component-by-component basis.

Consequently, equivalence sets can be used to detect performance sapping network anomalies at the granularity of individual components where partial faults may reside when used in conjunction with server-based application and network transport level statistics. In other words, if we identify a particular component in an equivalence set where application or network protocol performance is significantly degraded compared to other elements, we can conclude that this is due to a performance sapping anomaly centered at this component only—in other words, a partial fault affects this component.

### 5.1.2 Equivalence sets underpin outlier based partial-fault localization

Being able to form equivalence sets is necessary in order to successfully leverage outlier analysis to find partial faults in datacenters. Suppose we have a hypothetical set of components that matches the requirements for an equivalence set. We argue that if any one of the conditions in the set is relaxed and left unsatisfied, that outlier analysis can fail as a result.

First, suppose flows are not randomly split amongst members in the set. In other words, one or more of the members in the set has a greater or smaller likelihood that flows will be routed through that component—ECMP imbalances detected in production datacenters are a notable example [126]. In this case, we demonstrate that the distribution of performance metrics is no longer equivalent on a per-component basis. While this effect can be used to detect erroneous cases of ECMP imbalance, where traffic ought to be evenly load balanced, it means that partial faults cannot be reliably detected when routing policy dictates an unequal split of traffic. In such a case, false-positives may occur as more heavily loaded components may provide significantly worse performance than light loaded components.

Second, if the number of flows is small, we may get significant skew in the distribution of performance metrics on a per-component basis. This skew is due to the inability to perform effective statistical multiplexing of traffic when the number of flows is small; prior studies indicate that gross differences in per-link performance can occur in such a case [15]. These differences can prevent successful usage of outlier-analysis approaches, as the more heavily loaded link presents itself as an outlier even in the absence of a fault.

Finally, if components within a candidate set do not offer equivalent performance, then outlier-analysis can be confounded. On the other hand, we demonstrate in this chapter that if equivalence sets can be formed, that they are sufficient to find faults accurately and sensitively in a manner that is root-cause and application agnostic.

### 5.1.3 Equivalence-sets compared to prior approaches

Using equivalence sets, we devise a prototype for a partial-fault detection and localization system that differs in several key ways with prior academic proposals and production systems:

1. **Full path information**: Past fault-finding systems have associated performance degradations with components and logical paths [13, 23, 39, 64] but a solution that correlates performance anomalies with specific network elements for arbitrary applications has, to the best of our knowledge, proved elusive—although solutions exist for carefully chosen subsets of traffic [133]. On the other hand, equivalence-set based outlier-analysis both requires and takes full advantage of full path information for every flow in the network; we demonstrate that we can acquire this information within Facebook datacenters and use it to pinpoint faults to the specific links or switches they occur at.
2. **Passive monitoring, not active probing**: In contrast to active probing [11, 48, 64], our method uses readily available metrics from production traffic, simultaneously increasing detection surface and decreasing network overhead, detection, and localization time.
3. **Reduced switch dependencies**: While some approaches require expanded switch ASIC features for debugging networks [66], we do not require them. Thus, network operators

may deploy our approach on commodity switches. Furthermore, our approach is resilient to bugs that may escape on-switch monitoring.

4. **No per-application modeling:** Our system leverages stable and load-balanced traffic patterns found in some modern datacenters [112] in conjunction with properties resulting from equivalent sets. Thus, it does not need to model complicated application dependencies [13, 25, 39] or interpose on application middleware [40]. Since we compare relative performance across network links, we do not require explicit performance thresholds.
5. **Rapid online analysis**: Regularity inherent in academic [14] and production [19, 117] topologies allows us to perform rapid (10–20 seconds) online detection of small magnitude ($\leq 0.5\%$ packet loss) partial faults with equivalence sets. This rapidity contrasts with prior systems that require offline analysis [13] or much larger timescales to find faults [25]. Furthermore, localizing faults is possible without resource-intensive and potentially time consuming graph analysis [48, 76, 84, 85, 86, 104].

## 5.2 System overview

In this section, we present the high-level design of a system that implements our proposed approach. To set the context for our design, we first outline several important characteristics of Facebook's datacenter environment. We then introduce and describe the high-level responsibilities of the three key components of our system, deferring implementation details to Section 5.3.

### 5.2.1 Production datacenter

Facebook's datacenters consist of thousands of servers and hundreds of switches grouped into a multi-rooted, multi-level tree topology [19]. The datacenter we consider serves web requests from a multitude of end users, and is comprised primarily of web and cache servers [112]. While WAN connections to other installations exist, we do not focus on those in this work.

### Topology and service architecture

User requests are load balanced across all web servers, while cached objects are spread across all caches. Since any web server can service any user request, there is a large fan-out of web-to-cache connections; each web server has 1000s of bidirectional flows spread evenly amongst caches [112]. Prior work notes both the prevalence of partition-aggregate workloads and the detrimental impact of packet loss and delay in this latency sensitive environment—even if they only constitute the long tail of the performance curve [129].

Web and cache servers are grouped by type into racks, each housing a few 10s of servers. A few 10s of racks comprises a pod. Each pod also contains four aggregation switches (Aggs). Each ToR has four uplinks, one to each Agg. There are a few 10s of pods within the datacenter, with cross-pod communication enabled by four disjoint planes of core switches (each with a few 10s of cores). Each Agg is connected to the cores in exactly one plane in a mesh.

Due to the effects of ECMP routing, mesh-like traffic patterns, and extensive load balancing, links at the same hierarchical level of the topology end up with a very even distribution of a large number of flows. Moreover, if we know which path (i.e., set of links) every flow traverses, it is straightforward to separate the flows into bins based on the link they traverse at any particular level of the hierarchy. Hence, we can simultaneously perform fault identification and localization by considering performance metrics across different subsets of flows.

### Operational constraints

Datacenter scale imposes some significant challenges on our system. The large number of links and switches require our system to be robust to the presence of multiple simultaneous errors, both unrelated (separate components) and correlated (faults impacting multiple links). While some errors might be of larger magnitude than others, we must still be sensitive to the existence and location of smaller errors. In addition, we must detect both packet loss and delay.

The variety of applications and workloads within the datacenter further complicate matters—an improper choice of metric can risk either masking faults or triggering false positives

(for example, the reasonable sounding choice of request latency is impacted not only by network faults but also cache misses, request size and server loads). Moreover, datacenters supporting multiple tenants clearly require application-agnostic metrics.

Furthermore, we must be able to support measurements from large numbers of servers describing the health of a large number of links, without imposing large computational or data overheads either on the servers or on the network. Overheads especially impact network switches, where relatively under-provisioned control planes are already engaged in critical tasks including BGP routing. Thus, we are limited to capabilities present in commodity switch ASIC data planes.

### 5.2.2 System architecture

Our fault localization approach involves functional components at all servers, a subset of switches, and a centralized controller, depicted in Figure 5.1. Switches mark packets to indicate network path (1). Hosts then independently compare the performance of their own flows to generate a server-local decision about the health of all network components (2). These *verdicts* are sent (3) to a central controller, which filters false positives to arrive at a final set of faulty components (4), which may be further acted upon by other systems (5). We elaborate below.

**Servers**

Hosts run production application traffic and track various per-flow metrics detailed in Section 5.3.2. In addition, the server is aware of each flow's path through the network. Periodically, servers will use collected performance data to issue verdicts for whether it considers a given subset of flows to have degraded performance, or not. By default, flow metrics are binned by the set of links they traverse. These bins are then further grouped into what we call *equivalence sets* (ESes), i.e., the set of bins that should perform equivalently, allowing us to pinpoint link-level faults. In the Facebook datacenter, the set of bins corresponding to the downlinks from the network core into a pod forms one such equivalence set. Alternative schemes can give us further resolution into the details of a fault: for example, comparing traffic by queue

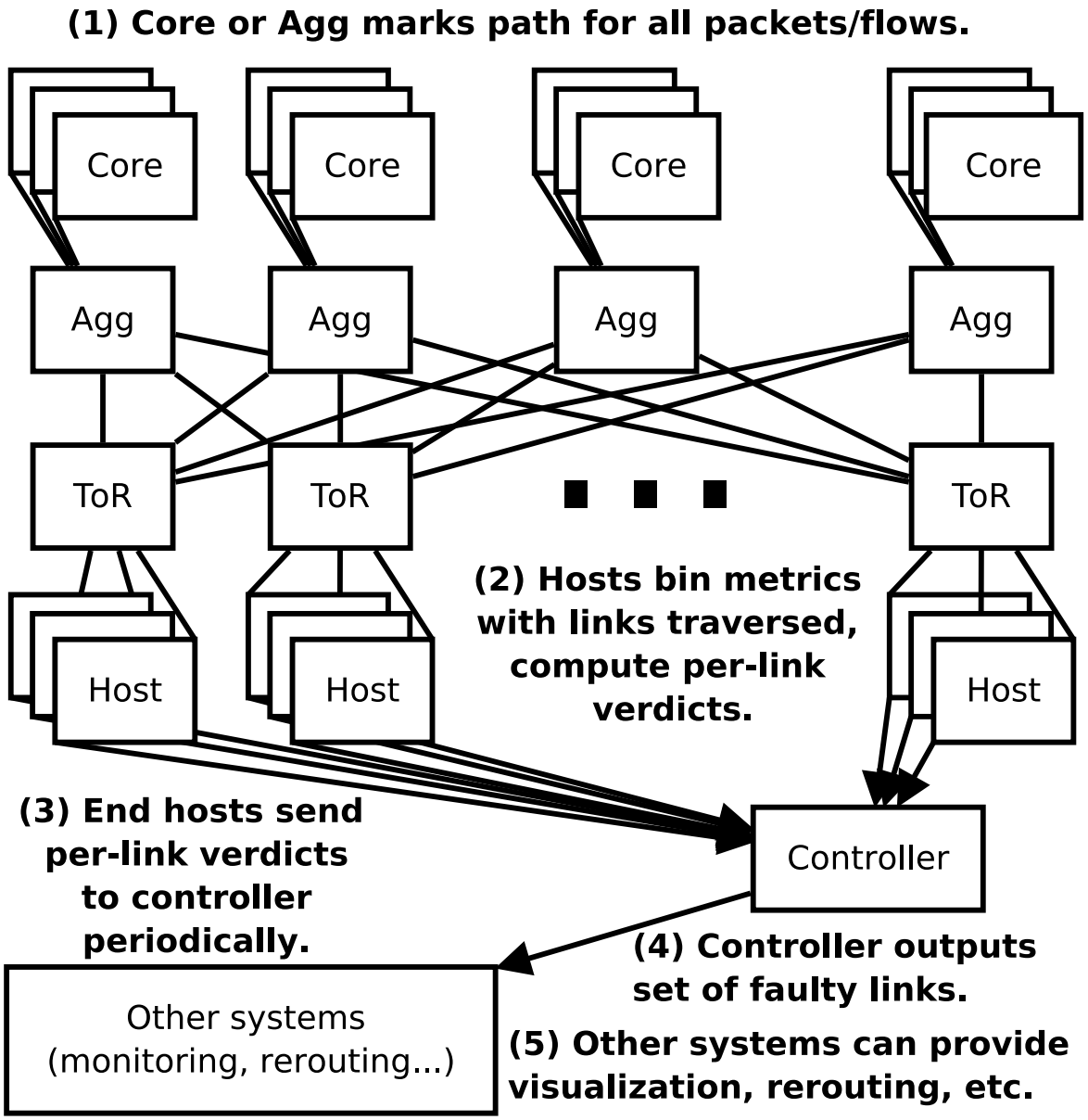



**Figure 5.1.** High-level system overview (single pod depicted).

or subnet (Section 5.4.4). We discuss the impact of heterogeneous traffic and topologies on our ability to form equivalence sets in Section 5.5.

We define a *guilty* verdict as an indication that a particular bin has degraded performance compared to others in its ES; a *not guilty* verdict signifies typical performance. We leverage path diversity and the ability to compare performance across links—if every link in an ES is performing similarly, then either none of the links are faulty, all of them are faulty (unlikely in a production network) or a fault exists but might be masked by some other bottleneck (for which we cannot account). The target case, though, is that enough links in an ES will be fault-free at any given moment, such that the subset of links experiencing a fault will be readily visible if we can correlate network performance with link traversed. Even in the absence of path diversity (e.g., the access link for a server) we can use our method with alternative binning schemes and equivalence sets to diagnose certain granular errors.

**Switches**

A subset of network switches are responsible for signaling to servers the network path for each flow; we describe details in Section 5.3.1. Once faults are discovered by the centralized controller, switches could route flows away from faulty components, relying on the excess capacity typically found in datacenter networks. We leave fault mitigation to future work.

**Controller**

In practice, there will be some number of false positives within server-generated verdicts for link health (e.g. servers flagging a link that performs poorly even in the absence of an error, possibly due to momentary congestion) confounding the ability to accurately deliver fixes to the network. Furthermore, there might be errors which do not affect all traffic equally; for example, only traffic from a certain subnet might be impacted, or traffic of a certain class. Hence, we employ a central controller that aggregates server verdicts for links (or other bins) into a single determination of which links—if any—are suffering a fault. In addition, the controller can drive

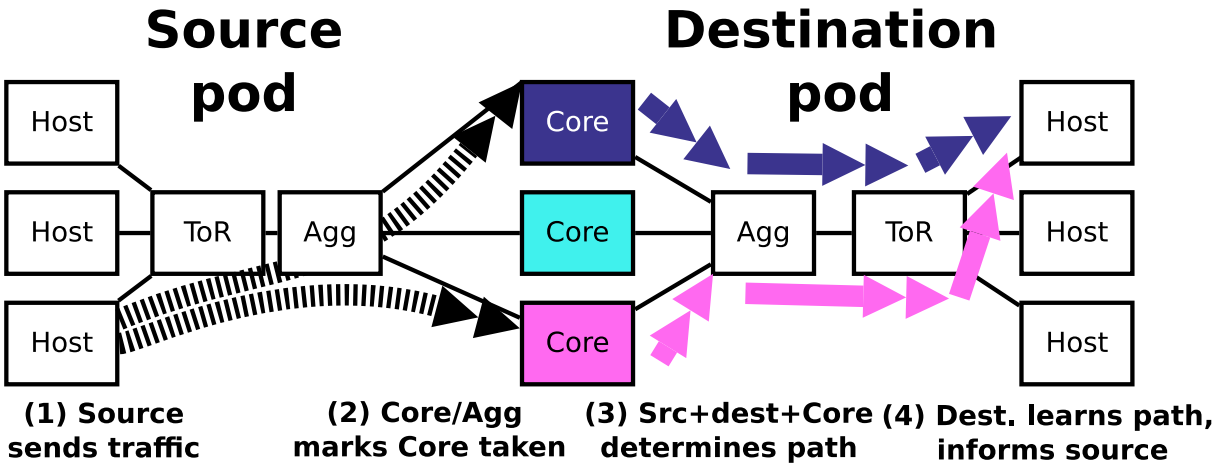



**Figure 5.2.** Determining flow network path.

other administrative systems, such as those used for data visualization/logging or rerouting of traffic around faults. Such systems are outside the scope of this work.

## 5.3 Implementation

We now present a proof-of-concept implementation that meets the constraints presented above. In order, we focus on scalable path signalling, our choice of server performance metrics and the required aggregation processing, our verdict generator for aggregated flow metrics, and the operation of our centralized controller.

### 5.3.1 Datacenter flow path discovery

Central to our approach is the ability to scalably and feasibly discover flow path information within the datacenter. While switch CPU/dataplane limits complicate this task, the regularity of the network topology aids us.

#### Topological constraints

Figure 5.2 depicts the pathfinding scheme we use in the Facebook datacenter. To aid discussion, the diagram shows a restricted subset of an unfolded version of the topology, with source servers on the left and destinations on the right; the topology is symmetric so our approach

works in either direction. Note that cross-pod traffic has only two ECMP decisions to make: which Agg switch (and thus, core plane) to transit after the ToR, and which core switch to use within the core plane. Each core plane is only connected to one Agg switch per pod, so the destination Agg switch is fixed given the core plane.

The source and destination racks and ToRs are fixed for any particular server-pair, and can be determined by examining the IP addresses of a flow. Thus, for cross-pod traffic, the choice of core switch uniquely determines the flow's path, as the Agg switches are then constrained on both sides. For intra-pod traffic it suffices to identify the Agg switch used to connect the ToRs.

In the presence of a total link error, the network attempts to forward traffic using an alternative, non-shortest path advertised as a backup route. While we do not address this case in our proof of concept, we discuss the impacts of link failures in Section 5.5.3.

**Packet marking**

We assign an ID to each core switch, that is stamped on all packets traversing the switch. Note that the stamp need not be inserted by the core switch itself—the Agg switches on either side are also aware of the core's identity and are equally capable of marking the packet. We use Linux eBPF (Extended Berkeley Packet Filter) [4] along with `bcc` (BPF Compiler Collection) [3] server instrumentation to read markings and derive flow paths. Our naive implementation imposes less than 1% CPU overhead (top row of Table 5.2), but room for optimization remains.

Several candidate header fields can be used for marking, and the best choice likely depends on the details of any given deployment. One possibility is the IPv6 flow label field; a 20-bit ID could scale to a network with over a million core switches. However, the ASIC in our Agg switches does not currently support modifying this field. Thus, for our proof of concept, we instead mark the IPv6 DSCP field, which is supported at line rate and requires only a constant number of rules (one per uplink).

While DSCP suffices for a proof of concept, its length limits the number of discernible paths. Furthermore, datacenter operators often use DSCP to influence queuing, limiting the

available bits. One alternative is to mark the TTL field[1]. A packet marked at a downward-facing Agg traverses exactly two more hops before arriving at the destination server; a server could recover an ID in the TTL field as long as the value was in the range 3–255.

**General case**

More complex topologies might preclude our ability to compute paths from a single stamp. In the event that traffic is made up of long-lived flows (as it is, for example, for the web and cache servers we tested) we can leverage the match/mark capability that many switching ASICs possess to implement a form of marking reminiscent of previous proposals for IP traceback [114].

Suppose there are $H$ hops between servers, and $C$ routing choices per hop. If we want to determine the flow path at the first hop where a routing choice must be made, the operating system can mark $C$ packets, each with the IPv6 flow label set to the possible IDs of each switch that the packets could transit for that hop. Switches would be configured with a single rule that would examine the flow label—if it matches the switch ID, the switch would set a single DSCP bit to 1. When the server receiving the packet notes the DSCP bit set to 1, it could signal the sender with the ID of the switch that was transited at that hop (that is, the flow label of the packet when the DSCP bit was set). For example, it could do this by setting a specific DSCP bit in the ACK packet while setting the flow label of the return packet to the switches ID. Thus, if the flow sends $\geq (H \cdot C)$ packets in total it can discover the entire path, at the expense of just a single rule per switch. While we have not deployed this approach in production, we validated that our switch hardware can implement it. While this method finds out path information hop by hop, partial path information can still be useful to our system. We discuss this further in Section 5.5.4.

### 5.3.2 Aggregating server metrics & path data

Given the availability of per-flow path information, we show that both transport-layer and system-call-timing metrics can be used to find under-performing links in real-world scenarios.

[1] Constraints exist on the use of TTL as well, such as in the presence of `traceroute` or eBGP session protection [42].

We consider both latency-sensitive, client/server production applications [112] and bulk-flow, large-scale computation applications [5, 46, 70]. Mixed traffic is considered in Section 5.5.2.

**Latency sensitive services**

Web servers and caches within the Facebook datacenter service user requests, where low latency is desirable [129]. Faults harm performance, where either drops or queuing delay can result in unacceptable latency increases. Since loss is often a sign of network congestion, the TCP state machine tracks various related statistics. These statistics include the number of retransmitted packets, the congestion window (`cwnd`) and the slow-start threshold (`sstresh`). Latency is also tracked using smoothed round trip time (`srtt`). When considered in isolation, these metrics are limited in usefulness; while a retransmit signifies diminished performance for a flow, it does not provide any predictive power for the underlying cause, or whether a given set of flows experiencing retransmits are doing so for the same underlying reason. Furthermore, while `cwnd` and `ssthresh` decrease with loss, the specific values depend highly on the application pattern. For example, bulk flows tend to have a larger congestion window than mouse flows. Thus, comparing any given flow against an average can be difficult, since it is unclear whether 'worse' performance than average is due to a network issue or traffic characteristics.

However, when augmented with path data, these metrics can become valuable indicators of faults. To illustrate the effectiveness of this approach, we induced a network fault impacting two servers: one web server, and one cache server. In each case, the local ToR is connected to four aggregation switches. Using `iptables` rules, each server dropped 0.5% of incoming packets that transited the link from the first aggregation switch to the ToR switch. We then measured the TCP metrics for outgoing flows, grouped by which aggregation switch (and, thus, rack downlink) was transited on the way to the server.

Rows 1–6 and 9–14 in Table 5.1 depict the `cwnd`, `ssthresh`, and retransmission count distributions grouped by inbound downlink for production cache (C) and web servers (W) respectively. While we aggregate the values for the non-faulty links into a single series (the even

**Table 5.1.** Metric for Production/Synthetic (C)ache/(W)eb/(H)adoop servers grouped by (In/Out)bound path and loss rates; syscall metrics in msec. Each color-banded row pair denotes base and impacted metrics for (aggregated) working and (unique) faulty paths.

| # | Host | Type | Metric | Path | Error | p25 | p50 | p75 | p90 | p95 | p99 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | (C) | Prod | cwnd | In | 0.5% | 7 (-30%) | 8 (-20%) | 10 (par) | 10 (par) | 10 (-41%) | 20 (-33%) |
| 2 | (C) | Prod | cwnd | In | - | 10 | 10 | 10 | 10 | 17 | 30 |
| 3 | (C) | Prod | ssthresh | In | 0.5% | 6 (-62.5%) | 7 (-63.2%) | 18 (-25%) | 24 (-64.7%) | 31 (-51.6%) | 63 (-17.1%) |
| 4 | (C) | Prod | ssthresh | In | - | 16 | 19 | 24 | 57 | 64 | 76 |
| 5 | (C) | Prod | retx | In | 0.5% | 0 (par) | 1 | 2 | 3 | 4 | 6 |
| 6 | (C) | Prod | retx | In | - | 0 | 0 | 0 | 0 | 0 | 1 |
| 7 | (C) | Prod | epoll | In | 0.5% | 0.003 (par) | 0.14 (+1.4%) | 0.47 (+10.8%) | 0.71(+30.6%) | 1.07 (+60.6%) | 2.28 (+125%) |
| 8 | (C) | Prod | epoll | In | - | 0.003 | 0.14 | 0.43 | 0.54 | 0.67 | 1.01 |
| 9 | (W) | Prod | cwnd | In | 0.5% | 8 (-63.6%) | 12 (-60%) | 17 (-74.6%) | 38 (-60%) | 121 (+23.5%) | 139 (-33.2%) |
| 10 | (W) | Prod | cwnd | In | - | 22 | 30 | 67 | 95 | 98 | 208 |
| 11 | (W) | Prod | ssthresh | In | 0.5% | 4 (-42.9%) | 12 (-40%) | 16 (-66.7%) | 19 (-75%) | 31 (-66.7%) | 117 (+19.4%) |
| 12 | (W) | Prod | ssthresh | In | - | 7 | 20 | 48 | 73 | 93 | 98 |
| 13 | (W) | Prod | retx | In | 0.5% | 0 | 0 | 1 | 3 | 4 | 6 |
| 14 | (W) | Prod | retx | In | - | 0 | 0 | 0 | 0 | 0 | 0 |
| 15 | (C) | Syn | select | Out | 2.0% | 0.56 (+25k%) | 4.22 (+717%) | 7.01 (+642%) | 37.5 (+1.6k%) | 216 (+4.3k%) | 423 (+969%) |
| 16 | (C) | Syn | select | Out | - | 0.002 | 0.516 | 0.944 | 2.15 | 4.90 | 39.6 |
| 17 | (W) | Syn | cwnd | Out | 0.5% | 2 (-80%) | 2 (-80%) | 2 (-80%) | 4 (-60%) | 7 (-30%) | 10 (par) |
| 18 | (W) | Syn | cwnd | Out | - | 10 | 10 | 10 | 10 | 10 | 10 |
| 19 | (W) | Syn | ssthresh | Out | 0.5% | 2 (-50%) | 2 (-71%) | 2 (-75%) | 2 (-78%) | 4 (-56%) | 7 (-22%) |
| 20 | (W) | Syn | ssthresh | Out | - | 4 | 7 | 8 | 9 | 9 | 9 |
| 21 | (W) | Syn | retx | Out | 0.5 | 21 | 23 | 26 | 29 | 30 | 40 |
| 22 | (W) | Syn | retx | Out | - | 0 | 0 | 0 | 0 | 0 | 0 |
| 23 | (H) | Syn | select | Out | 2.0% | 22 (+11%) | 223 (+723%) | 434 (+85%) | 838 (+32%) | 1244 (+47%) | 2410 (+39%) |
| 24 | (H) | Syn | select | Out | - | 19.5 | 27.1 | 235 | 634 | 844 | 1740 |

rows in the table), each individual non-faulty link follows the same aggregate distribution with little variation. On the other hand, the faulty-link distribution for each metric is significantly skewed—towards smaller numbers for `cwnd` and `ssthresh`, and larger for retransmits. Rows 17–22 show that `cwnd`, `ssthresh`, and retransmits provide a similarly strong signal when the link fault impacts traffic in the outbound direction instead. `srtt` is effective for detecting faults that induce latency but not loss; we defer details to Section 5.4.3.

**Bulk data processing**

Next, we consider bulk data processing workloads. Commonly used frameworks like Hadoop involve reading large volumes of data from various portions of the network; slowdowns caused by the network can have a disproportionate impact on job completion times due to stragglers. While the TCP metrics above work equally well in the case of Hadoop (not shown), the high-volume flows in Hadoop allow us to adopt a higher-level, protocol-independent metric that depends on the buffer dynamics present in any reliable transport protocol.

Consider the case of an application making a blocking system call to send data. Either there will be room present in the connection's network buffer, in which case the data will be buffered immediately, or the buffer does not have enough space and causes the application to wait. As packets are transmitted, the buffer is drained. However, if a fault induces packet drops, packets need to be retransmitted and thus the goodput of the network buffer drops. Correspondingly, the buffer stays full more of the time and the distribution of the latency of `send()` and similar blocking system calls skews larger. Delays caused by packet reordering have similar impacts. Non-blocking sends exhibit this behavior too; we can instrument either the `select()` or `epoll_ctl()` system calls to get insight into buffer behavior.

To demonstrate this effect, we consider a synthetic traffic pattern representative of the published flow-size distributions for Hadoop workloads present in Facebook's datacenters [112]. Specifically, we designate one server in our testbed (see Section 5.4.1 for details) as a sink and the remainder as sources. In addition, we induce a random packet drop impacting 1 of 9

**Table 5.2.** Server monitoring CPU utilization in production.

| Component | p25 | p50 | p75 | p95 |
|---|---|---|---|---|
| eBPF (paths) | 0.17% | 0.23% | 0.46% | 0.65% |
| TCP metrics/t-test | 0.25% | 0.27% | 0.29% | 0.33% |

testbed core switches. For a fixed period of time, each server in a loop creates a fixed number of simultaneous sender processes. Each sender starts a new flow to the sink, picks a flow size from the Facebook flow-size distribution for Hadoop servers and transmits the flow to completion, while recording the wait times for each `select()` system call.

Rows 23–24 in Table 5.1 show the distributions of `select()` latencies for flows grouped by core switch transited in our private testbed—again, the non-faulty distributions are aggregated. Faulty links on the impacted core yield a dramatically shifted distribution. Additionally, the distributions for non-faulty cores show little variation (omitted for space). This variation allows us to differentiate between faulty and normal links and devices. We found the metric to be sensitive to drop rates as low as 0.5%. Moreover, this signal can also be used for caches, due to the long-lived nature of their flows. Rows 7–8 and 15–16 in Table 5.1 show the distribution of `epoll()` and `select()` latencies for flows on faulty and non-faulty links in production and on the testbed, respectively; in both cases, the faulty link distribution skews larger.

**Metric collection computational overhead**

While the aforementioned statistics provide a useful signal, care must be taken when determining how to collect statistics. While system-call latency can be instrumented within an application, that requires potentially invasive code changes. Instead, we again leverage eBPF to track system-call latencies. For TCP statistics, we directly read `netlink` sockets in a manner similar to the `ss` command. Table 5.2 depicts the overall CPU usage of our TCP statistics collection and verdict generation agent; the cumulative CPU overhead is below 1%.

### 5.3.3 Verdict generation

Earlier, we demonstrated that path-aggregated server metrics can reveal under-performing links. Here, we develop a decision engine to generate metric-based testable hypotheses.

**Outlier detection**

Due to distribution clumping sans faults, we hypothesize that observed metric values can be treated as samples from a common underlying distribution characterizing a fault-free link given a time period and load. Moreover, a substantially different distribution applies for a faulty link for the same period. Thus, determining whether a link is faulty reduces to determining whether its sample distribution is part of the same distribution as the fault-free links.

While we cannot state with certainty the parameters that characterize the fault-free distribution because of the complexities of network interactions, we conjecture that the number of faulty datacenter links at any given instant is much lower than the number of working links. Consider the number of retransmits per flow per link, for all links, over a fixed time period. For each link, we compare its distribution to the aggregate distribution for all the other links. If there exists only one fault in the network, then there are two possible cases for any individual link: the distribution under consideration is faulty and the aggregate (excluding the single faulty link in this case) contains samples from exclusively non-faulty links, or it is non-faulty and the aggregate contains samples from 1 faulty link and $(N-1)$ working links. In the former case, the distribution under test is skewed to significantly higher values with respect to the aggregate; in the latter, it is skewed slightly lower (due to the influence of the single faulty link in the aggregate). Thus a Boolean classifier can be used: if the test distribution is skewed higher by a sufficiently large amount, as discussed below, we consider the corresponding link as potentially faulty; else, we do not. Concurrent errors across multiple links shift the aggregate distribution closer to the outlier distribution for a single faulty link. If every link is faulty, we cannot detect faulty links since we do not find any outliers. However, our method is robust even in the case where 75% of the links have faults; we examine the sensitivity of our approach in Section 5.4.4.

**Hypothesis testing**

Given our ability to collect empirical distributions of various metrics that can indicate network performance, we still need to devise a suitable methodology determining whether individual distributions are similar to others, or represent performance outliers. To do so, we leverage a well-studied branch of statistics called 'Hypothesis Testing'. Depending on the characteristics of the metric distributions we are studying, we can choose from among several well-known statistical tests to perform hypothesis testing.

**Differentiating between components in an equivalence set.**

On a high level, hypothesis testing allows us to formulate our problem in the following manner. Consider an equivalence set consisting of individual links (for the purpose of concrete exposition; our methodology generalizes to equivalence sets made up of any type of component). Suppose we are determining whether the number of retransmits per flow for a given link *TESTLINK* is higher than all of the other links in a given equivalence set, given a set of distributions corresponding to each link in the set, measured at a single server within some time period. Each empirical distribution corresponds to exactly one link and is made up several points, where each point represents the number of retransmits for a specific flow that traversed the link during the period examined. We refer to the distribution under test as L. Additionally, we create an aggregate distribution L' that contains the samples from every link $O$ where $O \neq TESTLINK$. Comparing L to L', we consider as our null hypothesis that the samples within L are from the same underlying distribution as L'. We set a significance level of $\alpha = 0.05$. The hypothesis test will output a p-value ranging from [0, 1]. If the p-value is $\leq \alpha$, we reject the null hypothesis and assume that L and L' are not sampled from the same underlying distribution—specifically, since L' represents all of the other links in the equivalence set, we consider *TESTLINK* to exhibit outlier performance and thus potentially host a partial fault, from the viewpoint of the server running this hypothesis test. On the other hand, if the null hypothesis is asserted, we do not consider *TESTLINK* as hosting a partial fault—again, from the point of view of this server.

The specific choice of test will depend on the metric. While the underlying distributions may not be known, we do sample a very large number of data points in every distribution—one for each flow, across 10s of 1000s of active flows. Furthermore, since we are considering the entire population of flows in every examined time period, we can compute the standard deviation across the entire population for each metric. Additionally, due to our equivalence set formulation, we assume that (in the no-error case) the distributions for each link are close to identical, and that each sample (corresponding to a single flow) is independent of the others (since we assume flow performance is uncorrelated). Thus, the central limit theorem allows us to assume that the average computed across all flows in a distribution is approximately normally distributed.

Given these conditions, for TCP retransmit, system-call latency, and TCP-over-IPv6 flow-label relabelling (discussed in Section 5.6) metrics, we use the single-tailed 1-sample Student's t-test. The t-test compares a sample mean (from *TESTLINK*, the link under test, with the distribution L) to a population mean (from the distribution L'), rejecting the null hypothesis if the sample mean is larger—in our case, if the tested link has more retransmits or higher system-call latency than the other links in aggregate. In every time period, each server checks if the t-statistic is greater than 0 and the $p$-value $\leq 0.05$. If so, we reject the null hypothesis, considering the link to be faulty.

Due to the large volume of traffic, even small time intervals contain a large number of samples. Thus, we have to carefully implement our outlier test to avoid significant server computational overhead. Our approach computes the t-statistic for each link over successive fixed-length (10 seconds by default) sampling periods. We can compute the t-statistic using a low-overhead and constant-space streaming algorithm. To do so we need the average and standard deviation per distribution, and the aggregate average—all of which are amenable to streaming computation via accumulators, including standard deviation via Welford's Algorithm [124].

Each server generates verdicts using the t-test for links associated with its own pod. Thus, for each flow, there are a few tens of possible core-to-aggregation downlinks, and four possible aggregation-to-ToR downlinks. Having acquired TCP metrics and path information for

all flows via our collection agent, we bin each sample for the considered metrics into per-link, per-direction buckets. Thus, each metric is binned four times: into the inbound and outbound rack and aggregation (up/down)links traversed.

This approach limits the computational overhead at any individual server since it only needs to track statistics for its own `select()/epoll()` calls and TCP statistics. The bottom row of Table 5.2 depicts the CPU utilization at a single production web server for this approach over 12 hours. The t-test computation and `netlink` TCP statistics reader uses roughly 0.25% of CPU usage. This result is roughly independent of how often the t-test is computed; the majority of the CPU usage is incrementing the various accumulators that track the components of the t-statistic. The `bcc`/eBPF portion, however, has periods of relatively high CPU usage approaching 1% overall, due to the need to periodically flush fixed-sized kernel structures that track flow data.

For `cwnd`, `ssthresh` and `srtt` TCP statistics, the student's t-test may be inapplicable; in particular, `cwnd` and `ssthresh` samples for a given flow across a fixed time period are not independent of each other as they evolve over time. Furthermore, analyzing TCP state-machine metrics is fraught with complication due to the complex interactions that can occur depending on different TCP versions, application demands, and traffic pattern characteristics. Since we do not make any assumptions on empirically-measured TCP state machine metric distributions, we fallback to using non-parametric tests for these metrics. Fundamentally, we seek to find the 'distance' between two distributions and determine if this distance is 'large', which is precisely the goal of the 2-sample Kolmogorov-Smirnov (KS-2) test. We apply the KS-2 test on two down-sampled distributions: the 99-point {p1, p2, …, p99} empirical distribution for the link under consideration, and a similarly defined distribution for the other links in aggregate. We downsample specifically because the large number of points in the original distributions makes the test too specific—downsampling allows us to compare distributions that have the same rough shape as the original (and thus, are qualitatively similar) without being unnecessarily sensitive. While the KS-2 test is applicable in our scenario, other non-parametric tests may suffice as well.

**Understanding underlying distributions and drawing more powerful inferences.**

Thus far we have adopted a relatively circumspect approach, testing only the differences between (metrics associated with the) components in an equivalence set. Due to the applicability of the central limit theorem in some cases, or the availability of non-parametric hypothesis tests in other cases, we have not needed to fully understand the underlying empirical distributions we are testing. Yet, a a deeper understanding of the empirical distributions can unlock the ability to draw more descriptive inferences. For example, a per-server, per-link packet-loss-probability confidence interval may be estimated using TCP retransmits as a proxy for packet loss (noting that in edge cases, retransmits may occur without loss, or a single loss may trigger more than one retransmit). Here, we discuss whether we can do so for the metrics we study.

A key question to answer is whether any of the metrics we collect are normally distributed. If they were, it could reduce our dependence on the central limit theorem, potentially allowing us to sample per-flow metrics from a smaller subset of flows. Confidence intervals can be easily computed across normal distributions as well, though methodologies often exist for other underlying distributions if we understand their parameters.

However, for most metrics, normality may not hold. For example, TCP Congestion windows depend heavily on application demands, and for Facebook cache server traffic the distribution is close to uniform in the absence of packet loss. For retransmits, we may argue that if a fault causes uncorrelated packet loss (that is, any given packet is equally likely to be dropped), then the loss count across a large number of packets can be approximated by a normal distribution. Thus, since retransmits approximate the number of lost packets, they too will be approximately normally distributed. However, it may be that a partial fault causes bursts of errors and drops trains of packets, where packets immediately following a dropped packet are more likely to be dropped. In such a case, normality may not apply. Thus, we conservatively assume that our raw metric distributions are not normal. Furthermore, we do not possess enough insight into TCP state-machine interactions to properly characterize the parameters associated with TCP state-machine metric distributions, especially given the hysteresis inherent within these metrics.

**What we cannot do.**

We may be tempted to use statistical techniques to root-cause (or at least, more fully characterize) partial faults after localization, in the vein of prior work [23]. We argue that this particular goal may be limited by our methodology for a few reasons. First, we make no claims about underlying distributions. Second, we are not privy to all of the confounding effects and interactions for the metrics examined. Third, we are passively monitoring metrics. While an active-probe injection approach might give us further insight into relevant partial-fault behaviours for a specific fault, here we are limited to observable variations within aggregate traffic behaviour.

Consequently, we cannot compute confidence intervals for the degree of packet loss on a link, or speculate on the underlying mechanism for a pinpointed partial fault. In other words, we can claim that a specific link is anomalous and possibly faulty, without being able to describe exactly *why*. Neither can we comment on the magnitude of the partial fault we have localized; while we demonstrate in Section 5.4 that we can rapidly pinpoint a 0.1% packet-loss-rate inducing partial fault or a 0.2% packet-loss-rate inducing partial fault, we cannot concretely state what the loss rate is for a given fault that we have localized. In particular, we are unable to distinguish between loss contributed by the fault vs. loss caused by regular congestion across the faulty link. That said, we can still use the output of our localization system to trigger active probe mechanisms [59, 133] to perform this task.

For pinpointing partial faults, however, we demonstrate that all we require is determining that metric distributions are different in an equivalence set. It is not the specific value that matters; only whether the value is worse than that of the other components in the specific equivalence set.

### 5.3.4 Centralized fault localization

While individual servers can issue verdicts regarding link health, doing so admits significant false positives. Instead, we collate server verdicts at a centralized controller that attempts to filter out individual false positives to arrive at a network-wide consensus on faulty links.

**Controller processing**

Assuming a reliable but noisy server-level signal, we hypothesize that false positives should be evenly distributed amongst links in the absence of faults. While some networks might contain hot spots that skew metrics and violate this assumption, traffic in the Facebook datacenter is evenly distributed on the considered timescales, with considerable capacity headroom.

We use a centralized controller to determine if all links have approximately the same number of guilty (or not guilty) verdicts, corresponding to the no-faulty-links case. Hosts write link verdicts—generated once per link every ten seconds—to an existing publish-subscribe (pub-sub) framework used for aggregating log data. The controller reads from the pub-sub feed and counts the number of guilty verdicts per link from all the servers, over a fixed accumulation period (10 seconds by default). The controller flags a link as faulty if it is a sufficiently large outlier. We use a chi-squared test with the null hypothesis that, in the absence of faults, all links will have relatively similar numbers of servers that flag it not-guilty. The chi-square test outputs a $p$-value; if it is $\leq 0.05$, we flag the link with the least not-guilty verdicts as faulty. We iteratively run the test on the remaining links to uncover additional faults until there are no more outliers.

**Computational overhead**

The controller has low CPU overhead: a Python implementation computes 10,000 rounds for 10s of links in $<$1 second on a Xeon E5-2660, and scales linearly in the number of links. Each server generates two verdicts per link (inbound/outbound) every 10 seconds, with O(1000s) servers per pod. Each verdict consists of two 64-bit doubles (t-stat and $p$-value) and a link ID. This yields a streaming overhead of $<$ 10 Mbps per pod, well within the pub-sub's capabilities.

## 5.4 Evaluation

We now evaluate our approach within two environments: the Facebook datacenter described in Section 5.2.1, and a small private testbed. First, we describe our test scenario in

each network, and provide a motivating example for real-world fault detection. We then consider the speed, sensitivity, precision, and accuracy of our approach, and conclude with experiences from a small-scale, limited-period deployment at Facebook.

### 5.4.1 Test environment

Within one of Facebook's datacenters, we instrumented 86 web servers spread across three racks with the monitoring infrastructure described in Section 5.3. Path markings are provided by a single Agg switch, which sets DSCP bits based on the core switch from which the packet arrived. (Hence, all experiments are restricted to the subset of traffic that transits the instrumented Agg switch, and ignores the remainder.) To inject faults, we use `iptables` rules installed at servers to selectively drop inbound packets that traversed specific links (according to DSCP markings). For example, we can configure an server to drop 0.5% of all inbound packets that transited a particular core-to-Agg link. This methodology has the effect of injecting faults at an arbitrary network location, yet impacting only the systems that we monitor.[2]

Our private testbed is a half-populated $k = 6$ fat tree, consisting of 3 (of 6) pods of 9 servers each, connected via 9 core switches. Each core has three links, one to each pod. To inject faults, we use a 'bump in the wire' to perturb packets on a link. For example, consider a faulty core that drops (or delays) random packets traversing a link to an Agg. To inject a fault at that link, we replace the link connecting the core to the Agg with a link connecting it to a network bridge, which is in turn connected to the Agg. The bridge is a packet-forwarding Linux server that randomly drops (or delay) packets in a given direction. We implement ECMP using source routing; the 5-tuple therefore allows us to determine the paths packets traverse in our testbed.

### 5.4.2 Motivating example

Over a five-minute interval in the Facebook datacenter, we induced faults on links from three different core switches to the instrumented Agg, denoted $A$, $B$ and $C$. In particular, we

[2]Note that while all monitored servers will see the same loss *rate* across the link, the actual packets dropped may vary because `iptables` functions independently at each server.

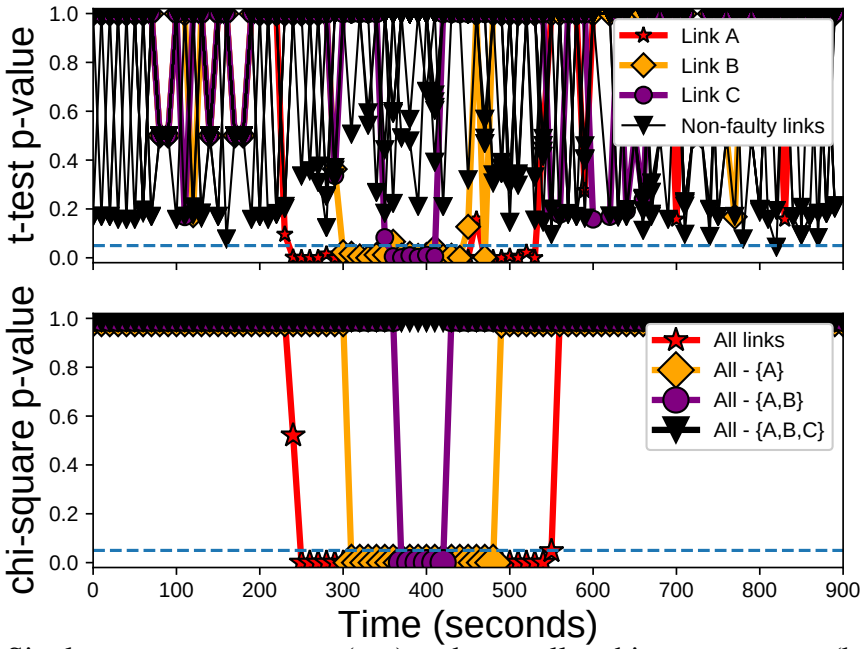
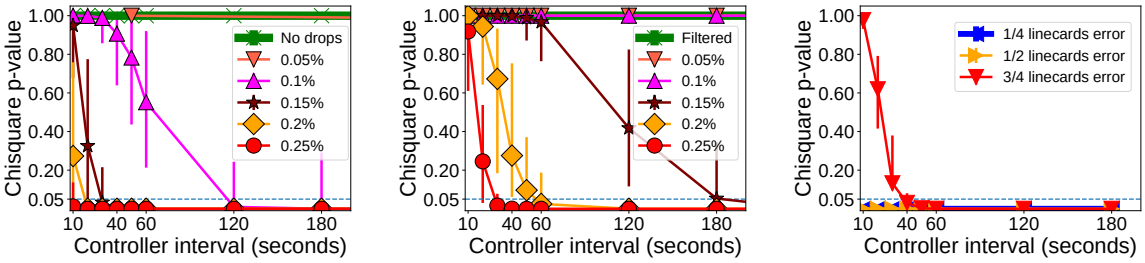



**Figure 5.3.** Single server t-test output (top) and controller chi-square output (bottom) for three separate link faults.

induce faults in the order $A$, $B$, $C$, with a one-minute gap; we then removed the faults in reverse order with the same gap. Each fault is a random 0.5% packet drop (1 in 200 packets). In aggregate, this corresponds to an overall packet loss rate of $< 0.02\%$.

The top portion of Figure 5.3 depicts the t-test output for a single server. Flows are grouped according to incoming core downlink, and TCP retransmission statistics are aggregated into ten-second intervals. For a single server, for every non-faulty link (every series in black, and the faulty links before/after their respective faults) the t-test output is noisy, with $p$-values ranging from 0.15 to 1.0. However, during a fault event the $p$-value drops down to near 0. Close to 100% of the servers flag the faulty links as guilty, with few false positives.

These guilty verdicts are sent to our controller. The controller runs the chi-squared test every 10 seconds using each core downlink as a category; it counts the number of *non-guilty* verdicts from the servers as metric and flags an error condition if the output $p$-value $\leq 0.05$. Note that this flag is binary, indicating that there exists at least one active fault; the guilty verdict count must be consulted to identify the actual guilty link. The bottom portion of Figure 5.3 depicts the controller output using this mechanism. We depict the output of the test for all paths (in red), and




**(a)** Loss rate sensitivity. **(b)** Masked fault sensitivity. **(c)** Linecard faults.

**Figure 5.4.** Controller chi-square $p$-value convergence for various faults vs. controller interval length.

for the set of paths excluding faulty paths $\{A\}$, $\{A,B\}$ and $\{A,B,C\}$ (in yellow, purple and black, respectively). These results indicate that the controller will find a fault as long as there is at least one faulty link in the set of paths under consideration, thus supporting an iterative approach.

### 5.4.3 Speed and sensitivity

For our mechanism to be useful, it must be able to rapidly detect faults of small impact. Moreover, we must detect faults that increase packet latency, not just those that cause loss.

#### Loss rate sensitivity

We performed a sensitivity analysis on the amount of packet loss we can detect in the datacenter. While loss rates $\geq 0.5\%$ are caught by over 90% of servers, we see a linear decrease in the number of guilty verdicts as the loss decreases past that point—at 0.1% drop rate, only $\approx 25\%$ of servers detect a fault in any given 10-second interval, reducing our controller's effectiveness. However, we can account for this by prolonging the interval of the controller chi-square test. Figure 5.4a depicts the distribution of $p$-values outputted by the controller for a given loss rate and calculation interval. Each point with error bars depicts the median, p5 and p95 $p$-values outputted by the controller during a fault occurrence; each loss rate corresponds to a single series. We see that while a 0.25% loss is reliably caught with a 20-second interval, a 0.15% loss requires 40 seconds to be reliably captured; lower loss rates either take an unreasonably large (more than

**Table 5.3.** `srtt_us` distribution vs. additional latency and request size. The no-additional-latency case is aggregated across all non-impacted links, while the others correspond to a single (faulty) link.

| Request bytes | Latency msec | p50 | p75 | p95 | p99 |
|---|---|---|---|---|---|
| 100 | - | 1643 | 1680 | 2369 | 2476 |
| 100 | 0.1 | 3197 | 3271 | 4745 | 4818 |
| 100 | 1.0 | 10400 | 10441 | 19077 | 19186 |
| 8000 | - | 4140 | 4778 | 619 | 7424 |
| 8000 | 0.1 | 6809 | 7510 | 9172 | 11754 |
| 8000 | 0.5 | 11720 | 14024 | 18367 | 21198 |

a minute) period of time to be caught or do not get detected at all. Note that in the no-fault case, no false positives are raised despite the increased monitoring interval.

### High latency detection

In our private testbed, we induced delays for traffic traversing a particular core switch. To do this, we used Linux `tc-netem` on our 'bump-in-the-wire' network bridges to add constant delay varying from 100 microseconds to 1 millisecond (a typical 4-MB switch buffer at 10 Gbps can incur a maximum packet latency of roughly 3 milliseconds before overflowing). We then ran client/server traffic with a single pod of 9 HHVM [6] servers serving static pages, and two pods (18 servers) configured as web clients running Apache Benchmark [1]. Each server handled 180 simultaneous clients and served either small (100-B) or medium (8-KB) requests.

Table 5.3 depicts TCP `srtt_us` distributions as a function of induced latency and request size. As before, the distributions of non-impacted links are similar, while the distribution of the latency heavy link is clearly differentiable. No drops were detected in the 100-B case; due to our use of 1-Gbps links, a handful of drops comparable to the no-fault case were detected in the 8-KB case. The modified KS-2 test operating over a 10-second interval correctly flags all intervals experiencing a latency increase, while avoiding false positives.

### 5.4.4 Precision and accuracy

Next, we demonstrate the precision and accuracy of the system in the presence of concurrent and correlated faults, as well as the absence of false positives.

**Concurrent unequal faults**

A large network will likely suffer concurrent faults of unequal magnitude, where the largest fault may mask the presence of others. While we surmise that the most visible fault will be easily diagnosable, ideally it would be possible to parallelize fault identification in this case.

We consider a scenario in the Facebook datacenter with two concurrent faults on distinct core-to-Agg switch links: one causing a packet loss rate of 0.5%, and one with a rate that varies from 0.25% to 0.15% (which we can easily identify in isolation). Using TCP retransmit statistics, the high-loss fault was flagged by almost all the servers the entire time, while the flagging rate for the lower-impact fault depends roughly linearly on its magnitude. However, the drop-off in guilty verdicts is steeper in the presence of a masking, higher-impact fault. As a result, the 10- and 20-second controller intervals that flag the low-loss-rate faults in isolation no longer suffice.

Figure 5.4b depicts the controller chi-square $p$-value outputs for the set of paths *excluding* the one suffering from the readily identified larger fault; each series corresponds to a different loss rate for the smaller fault. The interval needed to detect such "masked" faults is longer; a 0.25% loss rate requires a 40-second interval to reliably be captured vs. 20 seconds in the unmasked case (Figure 5.4a), while a 0.15% rate requires over three minutes.

**Large correlated faults**

So far, we have considered faults impacting small numbers of uncorrelated links. However, a hardware fault can affect multiple links. For example, each Agg switch contains several core-facing linecards providing cross-pod capacity. A linecard fault would affect several uplinks. Similarly, a ToR-to-Agg link might be impacted, affecting 25% of the uplinks for that rack's servers. Our approach relies on the student's t-test picking outliers from a given average, with the

**Table 5.4.** TCP congestion window and retransmit distributions when binning by remote rack with a faulty rack inducing a 0.5% drop rate.

| Metric | Error | p50 | p90 | p95 | p99 |
|---|---|---|---|---|---|
| retx | - | 0 | 0 | 1 | 2 |
| retx | 0.5% | 1 | 3 | 4 | 5 |
| cwnd | - | 10 | 18 | 26 | 39 |
| cwnd | 0.5% | 9 | 10 | 16 | 28 |

assumption that the average represents a non-faulty link. However, certain faults might impact vast swaths of links, driving average performance closer to that of the faulty links'.

To test this scenario, we induce linecard-level faults on 25%, 50%, 75% and 100% of the uplinks on the instrumented Agg switch in the Facebook datacenter. The per-link loss rate in each case was 0.25%. With 100% faulty links, our method finds no faults since no link is an outlier—a natural consequence of our approach. However, in all other cases our approach works if the servers declare paths faulty when the $p$-value $\leq 0.1$. Figure 5.4c shows the controller performance for various linecard-level faults as a function of interval length. A 10-second interval captures the case where 25% of uplinks experience correlated issues, but intervals of 20 and 40 seconds, respectively are required in the 50% and 75% cases.

**False positives**

The longer our controller interval, the more sensitive we are to catching low-impact faults but the more likely we are to be subject to false positives. We ran our system in production in the absence of any (known) faults with intervals ranging from 10 seconds to an hour. Even with 30-minute intervals, the lowest $p$-value over 42 hours of data is 0.84; only one-hour intervals generated any false positives ($p \leq 0.05$) in our data. We note, however, that we need not support arbitrarily large intervals. Recall that an interval of roughly three minutes is enough to get at least an intermittent fault signal for a 0.1% loss rate.

**Granular faults and alternative binnings**

By default, our approach bins flow metrics by path. In certain cases, however, a fault may only impact a specific subset of traffic. For example, traffic from a particular subnet might exhibit microburst characteristics, periodically overflowing switch buffers and losing packets.

Alternative binnings can be employed to identify such "granular" faults. To illustrate, we induced a fault at a single cache server, in which packets from exactly one remote rack are dropped at a rate of 0.5%. We then binned traffic by remote rack. Table 5.4 depicts the distribution of congestion window and retransmit by remote rack; as before, the distributions for non-impacted bins are all close to each other. The KS-2 test and t-test successfully pick out the fault without false positives using the `cwnd` and retransmissions metrics respectively. Note such alternative binning can help diagnose faults even if there is no path diversity—in this case, the alternatives are provided by application load balancing.

### 5.4.5 Small-scale deployment experience

While our experiments focus on injected failures, we are obviously interested in determining whether our system can successfully detect and localize network anomalies "in the wild". Thus, we examine the performance of our system over a relatively long time period (in the absence of any induced failures) to answer the following questions: "Does our system detect performance anomalies?" "Do non-issues trigger false positives?" "Do we notice anomalies before or after Facebook's existing fault-detection services catch it?"

To answer these questions, we deployed our system on 30 servers for two weeks in early 2017. As an prototype, deployment was necessarily restricted; our limited detection surface thus impacted our chance of detecting partial faults. Another large-scale datacenter operator suggests that, in their experience, roughly 10 partial faults a day occur in a network containing O(1M) servers [97]. It is unsurprising, then, that our two-week trial on only 30 servers did not uncover any faults. We were, however, able to derive useful operational experience that we relate below.

**Response to organic failure**

On January 25th, 2017, the software agent managing a switch linecard that our system was monitoring failed. The failure had no immediate impact on traffic; the dataplane continued to forward traffic according to the most recent ruleset installed by the agent. Thus, the initial failure is invisible to and explicitly outside the scope of our tool, which focuses only on faults that impact traffic.

Roughly a minute later, however, as the BGP peerings between the linecard and its neighbors began to time out, traffic was preemptively routed away from the impacted linecard. Thus, applications saw no disruption in service despite the unresponsive linecard control plane. Yet, we observe that as traffic was routed away from the failed linecard, the distributions of TCP's `cwnd` and `ssthresh` metrics for the traffic remaining on the linecard's links rapidly diverged from the values on other links in the equivalence set. Figure 5.5 depicts the per-link mean congestion window measured by every server, aggregated per-linecard and averaged across every corresponding server, with the afflicted linecard colored red.

Deviations are immediate and significant, with mean `cwnd` dropping over 10% in the first interval after most traffic diverts, and continually diverging from working links thereafter. Furthermore, measured flow volume at each server traversing the afflicted linecard rapidly drops from O(1000s) to O(10s) per link. By contrast, one of Facebook's monitoring systems, NetNORAD [11], took several minutes to detect the unresponsive linecard and raise an alert.

It is important to note that in this case, we did not catch the underlying software fault ourselves; that honor goes to BGP timeouts. However, we do observe a sudden shift in TCP statistics in real time as traffic is routed away, as our system was designed to do. With respect to our stated goal—to pinpoint the links responsible for deleterious traffic impact—our system performs as expected. Thus, this anecdote shows that our system can compliment existing fault-detection systems, and provide rapid notification of significant changes in network conditions on a per-link or per-device basis.

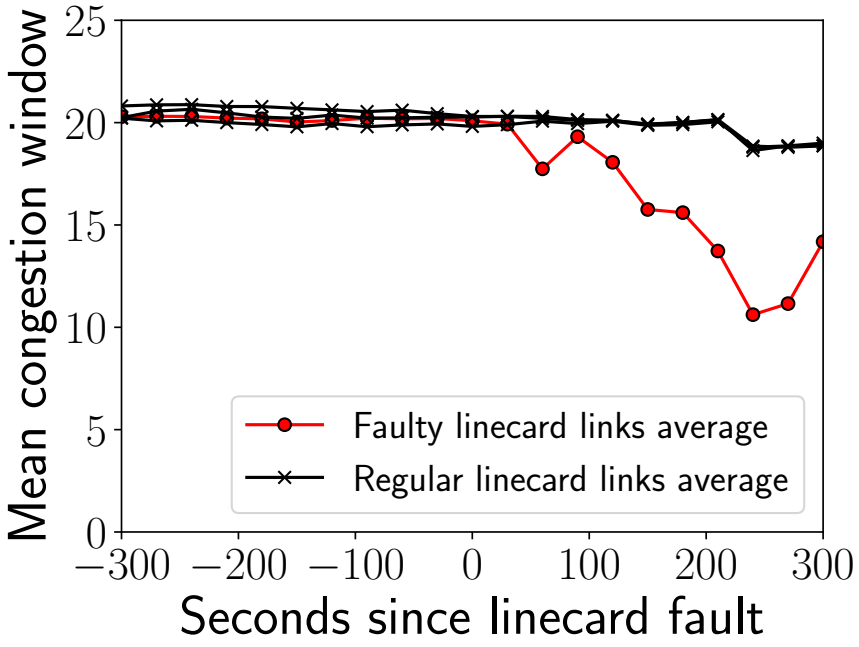



**Figure 5.5.** Mean `cwnd` per (server,link) during linecard fault.

### Filtering normal congestion events

During the monitoring period, no other faults were detected by the system. While a small number of false positives were generated in every interval, the controller filters out these indications since the noise is spread across the monitored links. However, we noticed that the number of false positives had periodic local maxima around 1200 and 1700 GMT. Furthermore, these were correlated with the raw (i.e., independent of flow/path) TCP retransmit counts tracked by the servers. Given that they occurred at similar times each day and were evenly spread across all the monitored servers, we surmise that these retransmits were not due to network faults, but organically occurring congestion. This experience provides some confidence that our system effectively distinguishes between transient congestion and partial faults.

## 5.5 Applicability

Here, we consider the applicability of our approach in various challenging scenarios, e.g., datacenters with heterogeneous traffic patterns, topologies less amenable to single-marking path discovery (either by design or due to failed links re-routing traffic), virtualized multi-tenant environments, and more. We first list some conditions to which our approach is resilient. Subsequently, we clarify the extent to which traffic homogeneity, link failures and topology impact the suitability of our approach. We conclude with a discussion on known limitations.

### 5.5.1 Surmountable issues

While we have access to only one production environment, we have conducted sensitivity analyses in our testbed to consider alternative deployments. Due to space limits, we summarize our findings here, but provide more extensive discussion in Appendix 5.8.

1. **VMs and high CPU load**: Our datacenter tests were on bare metal; our system works equally well in a VM-serving environment, even when the hardware CPU is fully taxed.
2. **Mixed, over-subscribed and uneven traffic**: The Facebook datacenter lacks saturated links and uneven load. We consider a mixed workload with latency sensitive and core-saturating bulk traffic, where server load ranges from 1×—16×.
3. **TCP settings**: Datacenter servers employ NIC offload features such as TSO. Turning these optimizations off does not obscure TCP or timing-based signals; neither does varying buffer sizes across three orders of magnitude (16 MB—16 KB).

### 5.5.2 Traffic homogeneity

Facebook traffic is highly load balanced [112], aiding our outlier-based detection approach. We are optimistic, however, that our method is also applicable to datacenters with more heterogeneous and variable workloads. That said, outlier analysis is unlikely to succeed in the presence of heterogeneous traffic if we do not carefully pick the links or switches that we compare against each other—specifically, if we fail to form a valid ES.

In the case of our experiments at Facebook, the ES we used was the set of downlinks from the network core into a pod. ECMP routing ensured that these links did form an ES; all cross-pod flows had an equal chance of mapping to any of these links, and the path from these links down to the servers were equal cost. This characteristic notably holds true regardless of the specific mix of traffic present. Thus, we hypothesize that on any network where such an ES can be formed, our approach works regardless of traffic homogeneity. To demonstrate this, we ran our fat-tree testbed with a worst case scenario of heterogeneous traffic: running synthetic bulk transfer and latency sensitive RPC traffic, with heavy traffic skew (with per server-load ranging from 1–16$\times$ the minimum load). Furthermore, we overloaded the network core by artificially reducing the number of links. Even in this case, our t-test classifier operating on the `select()` latency metric was able to successfully differentiate the outlier link.

### 5.5.3 Link failures

In addition to being able to detect and localize partial faults, our system must be able to account for total link failures, which can confound our ability to determine a flow's path through the network due to re-routing. Consider the outcome of a total link failure on the fate of traffic routed via that link. There are three possible outcomes for such traffic: (1) traffic is redirected at a prior hop to a working alternative path, (2) traffic is re-routed by the switch containing the dead-end link to a backup non-shortest path, and (3) traffic is black holed and the flow stops (the application layer might restart the flow).

Cases (1) and (3) do not affect our approach. ECMP routing will ensure that flows are evenly distributed among the surviving links, which still form an ES (albeit one smaller in size than before the failure). Case (2) can impact our approach in two ways. First, traffic taking a longer path will likely see worse performance compared to the rest of the traffic that traverses links on the backup path—harming the average performance on that path. Moreover, backup path performance might drop due to unfair loading as more flows join. Presuming rerouting is not silent (e.g., because it is effected by BGP), the former effect can be accounted for; traffic

using backup routes can be marked by switches and ignored in the server t-test computation. The latter can be mitigated by careful design: rather than loading a single backup path unfairly, the load can be evenly distributed in the rest of the pod. Even if an imbalance cannot be avoided, two smaller ESes can yet be formed: one with links handling rerouted traffic, and one without.

### 5.5.4 Topologies

Our system leverages the details of Facebook's datacenter topology to obtain full path info with a single marking identifying the transited core switch. The topology also allows us to form equivalence sets for each pod by considering the core-to-pod downlinks. Other networks might provide more challenging environments (e.g. middleboxes or software load balancers [105] might redirect some traffic; different pods might have varying internal layouts; links might fail) that confound the ability to form equivalence sets. In an extreme case, a Jellyfish-like topology [118] might make it extremely difficult to both extract path information and form ESes.

In certain cases, though, even networks with unruly layouts and routing can be analyzed by our approach. Consider a hybrid network consisting of a Jellyfish-like subset. For example, suppose a single switch in the Jellyfish sub-network is connected to every core switch in a multi-rooted tree, with identical link bandwidths. While we cannot reason about traffic internal to the Jellyfish, we can still form an ES for the links from the single switch connecting to the regular topology, for all traffic flowing into the regular sub-network. No matter how chaotic the situation inside the Jellyfish network, the traffic should be evenly distributed across core switches in the regular sub-network, and from then on the paths are equivalent.

Note that here, we only consider the subset of the path that lies within the regular topology. As long as an ES can be formed, the path behind it can be considered as a black box. Thus, we argue that even on topologies where we cannot find the full path, or where inequalities in path cost exist, we can run our approach on subsets of the topology where our requirements do hold.

### 5.5.5 Limitations

On the other hand, there are cases where our current approach falls short. To begin, we presume we are able to collect server metrics that reflect network performance. While we believe our current metrics cover the vast majority of existing deployments, we have not yet explored RDMA-based applications or datacenter fabrics. Our current metrics also have limits to their precision; while we can detect 0.1% drop rates in the datacenter we studied, past a certain point we are unable to discern faults from noise. Moreover, the production datacenter is well provisioned, so fault-free performance is stable, even in the tail. Hence, we do not consider the inability to detect minor impairments to be a critical flaw: If the impacts of a fault are statistically indistinguishable from background network behavior, then error severity may not be critical enough to warrant immediate response. Datacenters operating closer to capacity, however, may both exhibit less stable fault-free behavior, as well as require greater fault-detection sensitivity.

Despite extensive use of ECMP and application load-balancing, datacenters with mixed workloads may include links that see more traffic and congestion than others. That said, we have not encountered enough network load variability to trigger enough per-server false positives to confound the accuracy of our method in either production or under our testbed, and thus cannot yet quantify under what circumstances this degradation in detection performance would occur. Furthermore, for links lacking alternatives—such as an server to top-of-rack access link in the absence of multihoming—we cannot pick out an outlier by definition since there is only one link to analyze. We can, however, still perform our analysis on different groupings within traffic for that link; for example, when traffic to a particular subnet is impacted.

## 5.6 Applicability for high-variability bulk-traffic workloads

Outlier-analysis-based partial fault detection appears well-suited to web and cache server traffic at Facebook. In particular, the high degree of temporal stability and load-balancing, characteristic to both Facebook web and cache (henceforth, front-end) servers enables rapid

partial fault detection. To find lossy links servicing front-end servers, our approach requires only a few tens of seconds with loss rates on the order of 0.1%. However, not all services possess such favourable characteristics; instead, traffic can be spatially imbalanced, or possess temporal hot spots on timescales of 10s of seconds, confounding our approach. Here, we examine such traffic patterns at Facebook—in particular, we determine the applicability of our partial-fault localization system to Hadoop and WarmStorage [51] (henceforth, bulk-traffic) servers.

Hadoop [5] is a well-known batch data processing system, characterized by relatively few bulky flows in comparison to Facebook front-end servers. On a per-second basis, traffic is highly variable compared to both front-end servers, as well as more rack-local. WarmStorage [51] is a storage disaggregation system targeted at fixing various shortcomings present in HDFS (the Hadoop Filesystem) when used at Facebook's scale. Traditionally, Hadoop aims to leverage data locality, running computational tasks at servers where relevant data is known to reside [69] to avoid network utilization. However, WarmStorage discards this as a desired goal, and instead performs disaggregation of computation and storage—all storage access occurs remotely over the network. Since WarmStorage is aimed at supplanting HDFS, however, the notion of bulky and likely imbalanced flows applies to it as well.

### 5.6.1 Fundamental challenge: load balancing and temporal behaviour

Bulk-traffic workloads exhibit greater load imbalance and higher temporal variance compared to front-end servers, which can confound our ability to form equivalence sets. For example, consider a workload with high temporal variance, such as the type exhibited by Hadoop servers at Facebook [112]. Within a given 5-millisecond period, Hadoop servers have on average 25 concurrent active flows (defined by 5-tuple), though within the median-case millisecond, just 2 of these flows account for more than half the traffic sent in terms of volume in bytes. Consider also that the lifespan of individual flows within Hadoop servers range from approximately 10 seconds (in the roughly median case) to around 100 seconds (in the $95^{\text{th}}$ percentile). Furthermore, only a subset of these flows will be leaving the server rack.

Thus, if we consider inter-rack traffic sent by the Hadoop server within a 10-second interval, unlike the front-end servers, it is likely that the traffic will not be evenly spread amongst the top of rack switch uplinks within that period. This effect is exacerbated at higher levels of the network topology. A similar argument can be made for traffic destined to a Hadoop server within a 10-second period, as 99.8% of Hadoop server traffic is destined to other Hadoop servers [112]. Consequently, we cannot call the set of ToR uplinks or Agg switch uplinks an equivalence set at this timescale—they do not handle equivalent amounts of traffic during the period. Here, we consider effects like retransmits to be more likely to occur for links more traffic.

**Solution: Waiting out temporal instability.**

Despite second-by-second Hadoop traffic variability, there are sufficiently large time periods over which traffic does balance out—in particular, no inter-rack hot spots emerge for Hadoop clusters over a 24-hour period. This result suggests that outlier analysis may be possible if we consider a longer time interval than for front-end servers. One option for limiting the length of the required interval is to aggregate data from multiple servers together; instead of considering per-server data for the T-Test, we can aggregate information on a per-rack basis.

In Section 5.6.3 we see that performing a per-rack T-Test over 1-minute intervals or more (rather than 10-second intervals on a per-server basis as in the front-end case) allows us to continue to successfully use outlier analysis to find partial faults. However, simply increasing time intervals for applying our hypothesis tests were not enough, due to certain implementation challenges related to finding a suitable measurement metric for our tests given our path identification method. We discuss these challenges and how we overcame them next.

### 5.6.2 Implementation challenge: path information and metric choice

Earlier, our testbed experiments in Section 3.3.2 correlated outbound flow and application performance metrics with outbound path. Thus, if an outbound packet was lost, it would trigger an outbound retransmit which we correlated with links on the outbound path. This effect allowed us to use TCP metrics (retransmits, smoothed round trip time, etc.) in our testbed, for both

synthetic front-end server style traffic and for Hadoop. A subtlety applies within our Facebook testing, where we measured outbound metrics but correlated it with *inbound* path information to pinpoint per-link packet loss in the *inbound* direction. Here, we explain why this correlation is suitable for front-end traffic but *not* bulk-traffic. To continue supporting outlier-based partial fault localization for bulk-traffic servers, we discuss an alternative metric that we use instead.

At Facebook, our path-recovery methodology does not reveal path information for outbound flows. Due to our packet marking methodology, servers instead only have access to inbound flow path. However, TCP metrics collected at a server refer to performance for outbound flows, not inbound. Despite this, though, we are able to correlate outbound TCP flow metrics with the path taken by their corresponding inbound flows,and are able to use these correlations to detect inbound packet loss on a link by link basis, as we demonstrated in Section 5.4.

At first glance, this is a surprising result. Since outbound and inbound flows likely traverse different paths through the network, it appears strange that outbound flow performance should correspond with inbound flow path. We speculate that this is a consequence of the RPC-style nature of web and cache server traffic. Suppose a web server sends a request to a cache server. Prior studies suggest single-packet requests of approximately 200 Bytes [112] and typically, single-packet responses of an average of around one to a few hundred bytes [34]. Before a response is sent, however, the request packet may be acknowledged by a bare TCP-ACK packet. Suppose, further, that a single link in the path taken by the returned ACK exhibits partial-fault behaviour; in particular, it randomly loses packets.

Due to this fault, the returned ACK packet may be lost. At the web server sending the original request, the ACK is thus never received, and TCP interprets this as a loss of the original request packet. Thus, it resends the request and increments a retransmission counter for the flow. This *outbound* retransmit is thus caused by lossy behaviour for a link on the *inbound* path for the reverse-direction inbound flow. Consequently, we can correlate outbound TCP metrics like retransmit and congestion window with the links taken by the corresponding inbound flow, to find packet loss for inbound traffic on links.

What happens if we have a bulk-flow traffic pattern instead? Suppose, instead of sending a single packet, our flow must send several packets, as would be the case for data transfers in Hadoop and WarmStorage. In this case, an inbound ACK may still get lost, as before. However, consider that TCP stacks typically ACK every other packet within bulky flows. Thus, even if a particular ACK packet was lost, a subsequent ACK packet is likely to be received within a short period of time anyways, and thus no outbound retransmit occurs. Empirically, we notice that outbound packet retransmits and congestion window no longer appear to respond to inbound packet loss. Despite outbound retransmits correlating with inbound flow path for front-end servers, no such correlation appears for bulk-traffic servers. Thus, these metrics do not suffice as a proxy for inbound packet loss as was the case for front-end servers.

**Recovering packet loss information with flow-label churn.**

Since we cannot correlate outbound TCP metrics with inbound flow performance for bulk-traffic, we have two options: either we attempt to recover outbound path information for correlation with TCP metrics, or find a suitable inbound flow performance metric. To find outbound path information, one possible approach is to set up signalling between a communicating server pair, where the destination conveys the path taken by a received flow to the sender. However, this would require significant implementation effort and roll-out to be useful at Facebook's network scale—a large fraction of servers would need to participate in order to derive enough information for outlier analysis. While this is feasible to do, it requires more engineering effort than was available for a small scale prototype partial-fault localizer.

Thus, we must use an alternative metric that correlates with packet loss for inbound traffic. Since Facebook traffic is mostly TCP over IPv6 on Linux, an unexpected opportunity presents itself in the form of IPv6 packet header churn. The authors of Flowbender [75] proposed that, when faced with degraded performance, server network stacks may consider changing bits within a packet header in an attempt to influence packet routing through the network. Flowbender assumes that the bits in question are used by mechanisms such as ECMP routing, and that the network does possess multiple alternative paths—both of which are true at Facebook. Hopefully,

if a particular link or switch was responsible for degraded performance, the new path may avoid that particular component and thus benefit flow performance.

Recent versions of the Linux kernel possess a similar feature; specifically, when a certain number of retransmit timeouts occur for TCP over IPv6, the kernel network stack will modify the 20-bit wide IPv6 flow-label field in the hope that this field is consulted by network switches when making forwarding decisions within ECMP groups.

Thus, suppose a Hadoop or WarmStorage server is the recipient of a bulk-flow from another server running a recent Linux kernel. As in the front-end server case, the Hadoop or WarmStorage server can associate individual flows with every link the flow traversed through the network. It can then look for changes in the IPv6 flow-label field, and associate the resulting churn with the link the flow traversed when the churn occurred. The intuition here is that if the flow suffers from loss, the sending host will notice this loss and attempt to reroute the flow to avoid this loss, in a way that is visible to the destination server with the path information. Then, if a particular link has a higher than normal amount of associated flow-label churn, it could be due to a partial fault centered at that link.

Note that this is a significantly more sparse signal than retransmits; in particular, while a retransmit occurs for every lost packet, the flow-label will be modified only if several retransmit timeouts occur. By their nature, these are more rare than regular packet loss. While at least one retransmit occurs for any lost packet, a retransmit timeout only occurs if the packet at the end of a send window was lost. If a mid-window packet is lost, mechanisms such as duplicate ACK signalling will enable the sending server to resend a packet without suffering from a retransmit timeout, and thus, without changing the flow label.

In order to demonstrate this effect, we instrumented several Hadoop and WarmStorage servers with a collection agent that would correlate per-flow instances of flow-label churn with the specific link the flow was on when the churn occurred. We induced intermittent packet loss on one of the downlinks from either the datacenter core layer to a single pod aggregation switch serving the servers in question (WarmStorage), or from a single pod aggregation switch to the

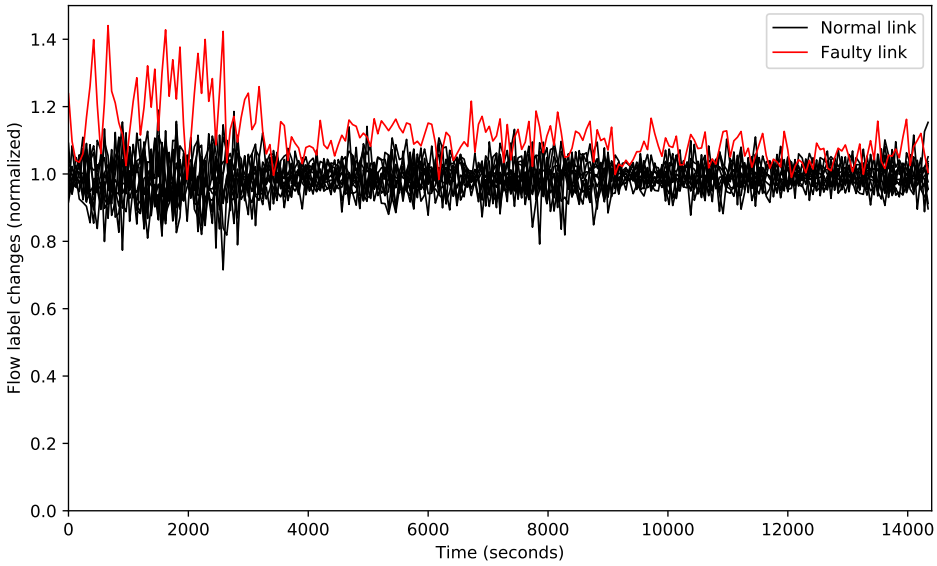



**Figure 5.6.** Normalized flow-label churn per link for WarmStorage.

ToR switch (Hadoop). To verify that flow-label churn responds to packet loss, we measured four hours of data and tracked the number of flow-label changes per link using 300-second intervals.

Figure 5.6 depicts the number of flow-label changes per link for each 300-second interval, normalized to the median number of flow-label changes across all surveyed links within the interval. While effect magnitude varies by time and network load, we see that the induced-fault-carrying link has a higher number of flow-label changes throughout the four-hour period.

Given that flow-label churn on a link-by-link basis can signal the presence of packet loss, the rest of our fault localization system simply uses this signal as input to the per-server T-Test as in the front-end server case. Specifically, by examining flow-label churn, we generate a timeline of flow-label churn (and thus, suspected packet loss for the sender) on a per link basis. Suppose we wanted to run the T-Test once every 60 seconds. We generate a distribution for each link and for each 60 second period by taking the following steps:

1. We arbitrarily subdivide the 60-second period into 100-millisecond chunks.
2. For each chunk, for each link, we count the number of flow-label changes across all flows. We discard the chunks where none of the links have any flow label change events.

3. For the remaining chunks, at least one link had a flow-label change. For links where no such change occurred within that 100-millisecond period, the value is 0. These counts are used to form a per-link distribution for the 60-second period.

4. We apply the T-Test for each link in the equivalence set; comparing its distribution to the aggregate distribution of the other links, as we did previously for front-end servers. T-test execution can occur, as mentioned, on a per-server level or on a per-rack level. The results can be processed, as before, by a centralized controller running the Chi-Square test.

### 5.6.3 Evaluation

Having described our updated mechanism for detecting partial faults at bulk-traffic servers, we evaluate it within a datacenter pod hosting WarmStorage servers. We focus primarily on the continued applicability of the per-server or per-rack T-test. We focus on the precision (low false positives) and recall (low false negatives) of the T-Test when operating at different frequencies, and for different intermittent packet loss rates.

#### Experiment setup

We instrument three racks of WarmStorage servers in a single pod, comprising 60 servers in total. We install our packet-marking rules onto a single pod switch, revealing to each server the path of all inbound traffic. Of the links connecting the Agg. switch to the network core, we induce partial fault behaviour affecting traffic transiting exactly one link by installing firewall rules at servers which randomly drops a small percentage of packets carried by the link.

Having thus induced intermittent packet loss, we configure the servers to track packet IPv6 flow-label fields for each flow over time, and generate per-link distributions for per-server and per-rack T-tests as described earlier. We compute T-test results over various intervals over an overall period of several hours.

To characterize the precision and recall of the T-test with the flow-label churn signal, we vary the degree of the packet loss from 0% (no loss) to 0.5%, and the frequency with which the

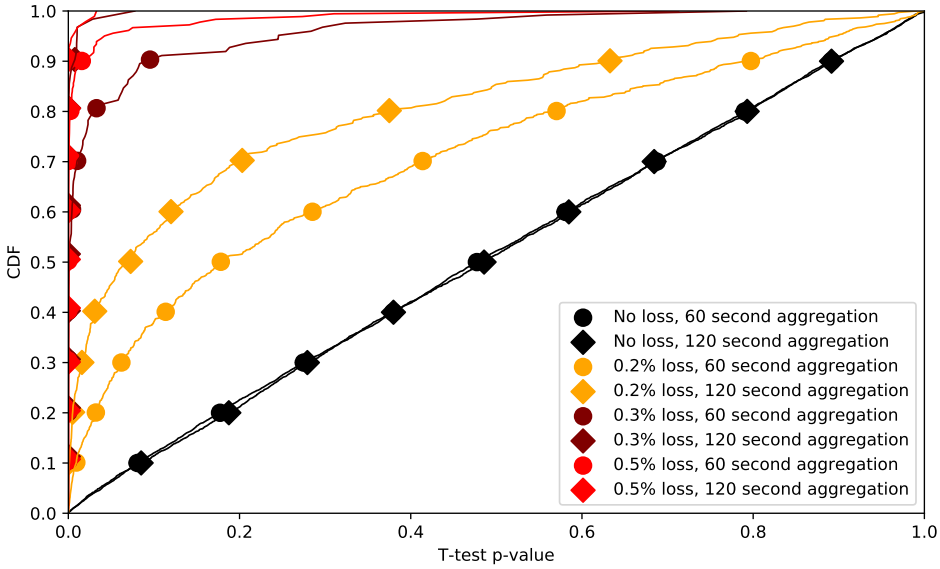



**Figure 5.7.** T-test p-value distribution as a function of T-test aggregation interval and packet loss rate for WarmStorage racks.

T-test is run from 60 seconds to 600. We also compute the T-test on a per-server and per-rack basis, as well as across all 60 servers combined to determine the degree to which aggregating measurements across servers is helpful or not.

### T-Test precision and recall

Due to the sparseness of the flow-label churn derived signal, we find that the T-test hit rate, or recall, *on a per-server basis*, is low when calculating the T-test every 60 seconds. For example, for a 0.5% drop rate, the T-Test yields a $p \leq 0.05$ approximately 15% of the time (c. f. nearly 90% of the time for front-end servers with a 10 second interval). For 0.2% drop rates, the distribution of T-test p-values is indistinguishable from the no-error case.

However, this does not mean that we are unable to leverage this methodology—rather than treat each server individually, we still have the ability to aggregate collected data and treat each server rack as a 'large' server. Figure 5.7 depicts, for a variety of packet-loss rates and T-test aggregation frequencies, the distribution of the T-test results from the three measured racks over the four-hour runtime of the experiment. For example, if we consider the case where the

T-test runs every 60 seconds, we receive 3 data points (one for each rack) per 60 second interval, and thus 3*240 = 720 data points over the four-hour period. A typical pod consists of a few tens of racks and could collectively generate the same volume of data points within a few minutes.

Each series in Figure 5.7 uses colouration to depict loss rate, where red, purple and orange depict 0.5%, 0.3% and 0.2% respectively. Each marker style depicts the T-test aggregation period, where circle markers correspond to 60 seconds and diamond markers correspond to 120 seconds. As an example, if we consider a 0.3% packet loss and run the T-test once every 60 seconds, then close to 80% of the resulting T-test p-values are $\leq 0.05$, giving us a hit-rate of close to 80%. Recall that for front-end servers under a 0.25% loss, the T-test had a hit-rate of roughly 75%; across 90 servers worth of data, this provided enough confidence for a centralized controller running the Chi-square test to correctly deduce the presence of a partial fault.[3] Thus, given a 0.3% or higher loss rate, per-rack aggregation and a 60-second T-test interval, outlier-analysis-based partial fault localization appears to be applicable to WarmStorage workloads.

However, a 0.2% loss has a significantly lower hit rate of $\approx 20\%$ with a 60-second test interval. Increasing the test interval to 120, 300 and 600 seconds does boost hit rate to $\approx 40\%$, 70% and 90% respectively, however. In the front-end scenario, these hit-rates did allow the Chi-Square test to deduce the presence of a faulty link, though a 40% hit-rate required the controller to aggregate multiple rounds of T-test results for detection to occur. Thus, even for a 0.2% loss rate, outlier analysis is applicable for partial-fault detection. However, the signal was not strong enough to catch a 0.1% packet loss rate; at that point, the signal was not strong enough to rise above the noise of the usual incidence of flow-label churn we see in the no-error case.

That said, the behaviour of the T-test p-value distribution in the no-error case does suggest an alternative approach. In particular, the no-error case yields a roughly uniform distribution of p-values regardless of T-test interval. On the other hand, for a 0.2% loss when using a 60-second T-test, we observe a decidedly non-uniform distribution. For any T-test interval, for a 0.2% or higher loss rate, we see a marked shift to the left for the p-value distribution.

[3]For front-end servers, though, the T-test only had to run for 10 seconds instead of 60.

Thus, instead of running the Chi-Square test by bucketizing by link and counting the number of T-test hits per link as we did for front-end servers (where a T-test hit is a p-value $\leq$ 0.05 and we look for outlier links/Chi-square buckets), we could run a separate instance of the Chi-Square test per link and determine whether the p-value distribution was uniform or not, and use that to determine if a fault was present. This approach would hypothetically allow us to catch the 0.2% loss fault within 60 seconds, if we formed a distribution using every rack in the pod.

## 5.7 Alternative methods

Alternative methods abound for investigating datacenter faults. One option is to couple switch counters with programmable switches. For example, suppose a switch is configured such that when a packet is dropped, either due to error or queuing pressure, it is probabilistically sampled to the switch CPU (note that with switches shifting millions of packets a second, examining all packets leads to unacceptable CPU overhead). Additionally, some switches can be configured to sample packets from high occupancy queues. Thus, counters could indicate a fault, while sampled packets could be used to build a picture of which traffic is impacted.

While this method can alert network operators to faults, it is slow in determining impacted traffic. Suppose a link carrying 2,000,000 packets per second develops a 0.5% drop rate, leading to 10,000 drops per second. Such a link might be carrying 10s of 1000s of flows, however. With a relatively high 1 in 100 sampling rate (100 packets per second), and with just 10,000 flows carried (a large underestimation) it would take around 100 seconds to determine all the flows if the sampling was perfect and captured a different 5-tuple each time. Furthermore, sampling is subject to bias; a lightweight flow carrying control traffic might lose packets but fly under the radar of sampled dropped packets. One heavy handed potential approach could be to disable the entire link; however, consider the case of a partial fault only affecting a subset of traffic (for example, a misconfigured routing rule). In such a case, disabling the entire link would penalize all traffic routed through that link without providing any insight to the underlying problem.

For contrast, our proposed system can quickly identify all the flows impacted by a link error in a less than a minute in several tested cases. In the case of granular errors affecting only a subset of traffic (by source or destination subnet, application port, queuing policy, etc.) our mechanism can still detect outliers, enabling servers to reroute around damaged links for afflicted traffic as a stopgap measure. Note that switch counters can be used in conjunction with our methods; features of sampled dropped or latency enduring packets could be used to guide our system's binning of flows in order to converge on which traffic is affected by link faults even quicker. Furthermore, our proposed system is ASIC agnostic since we do not rely on the features of any given chipset. Finally, it is robust to unreliable reporting by switches, as well as the uncertainty of counters that might arise within environments using cut-through routing.

## 5.8 Metric robustness

Here, we demonstrate the effectiveness of our approach in the presence of pathological confounding factors. These experiments are performed on our private testbed, since we could not induce harmful configuration changes in production traffic. We focus on syscall latency metrics, which are less obviously robust to many of these factors. To save space, we omit similarly strong results using TCP statistics.

### 5.8.1 CPU utilization

Datacenter networks run a variety of applications, frequently with stringent CPU requirements and high utilization. To be general, our approach needs to cope with high utilization figures. Furthermore, while some datacenters run applications on bare metal hardware, a significant number of installations use virtual machines.

To ascertain the impact of virtual machine hardware and high CPU utilization on our approach, we set up an experiment where each server in our private testbed runs two virtual machines, `cpuvm` and `netvm`. Each instance of `cpuvm` runs a variety of simultaneous CPU intensive tasks: specifically, a fully loaded `mysql` server instance (with multiple local clients), a

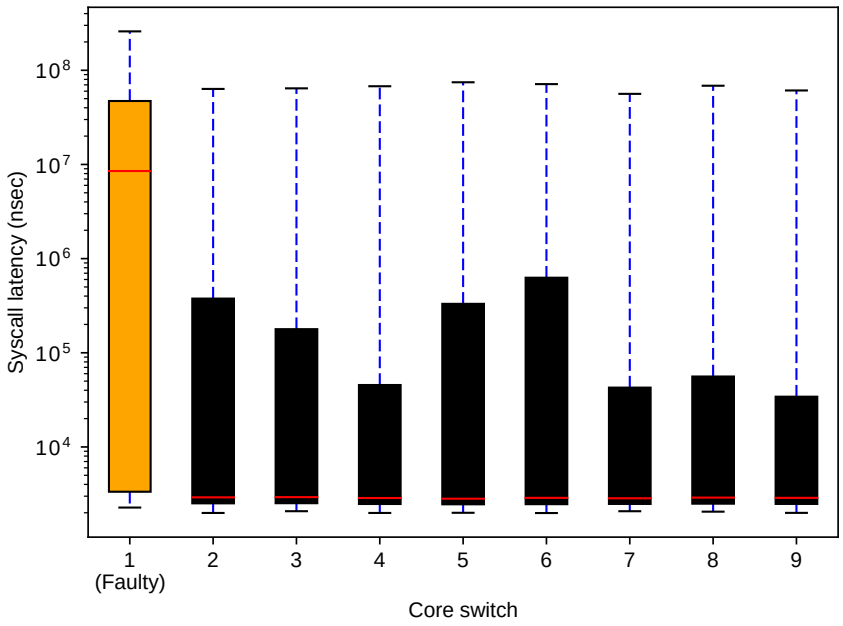



**Figure 5.8.** `select()` latencies per core switch for the two-VM CPU stress test.

scheduler-intensive multithreading benchmark, and an ALU-operation-intensive math benchmark. These tasks cause the CPU utilization of the bare metal server to go to 100%. Meanwhile, each instance of `netvm` runs a bidirectional client-server communication pattern using flow size distributions prevalent in certain datacenters [112]. This situation is analogous to the case where one VM is shuffling data for a map-reduce job, while another is actively processing information, for example. Figure 5.8 depicts the distribution of `select()` latencies across each core switch in the presence of this extreme CPU stress. The distribution corresponding with paths through the faulty switch are clearly distinguishable, with a median value over three orders of magnitude greater than the no-error case. These variations enable a 100% accuracy rate of flagging the faulty link using our chi-squared test over 1-minute intervals for a 5-minute long test.

### 5.8.2 Oversubscription and uneven load

Thus far, all our experiments have networks without saturated core links. While this is a desirable quality in real deployments, it might not be feasible depending on the amount of

oversubscription built into the network. Furthermore, in heterogeneous environments such as multi-tenant datacenters, it is conceivable that not every server is participating in monitoring traffic metrics. Finally, not every system that is being monitored is subject the same load.

To determine how our approach deals with these challenges, we devised an experiment in our private testbed where 1/3rd of our 27 servers are no longer participating in monitoring. Instead, these servers suffuse the network with a large amount of background traffic, with each so-called 'background server' communicating with all other background servers. Each background server sends a large amount of bulk traffic; either rate limited to a fraction of access link capacity, or unbound and limited only by link capacity. In conjunction with this, our network core is reduced to 1/3rd of normal capacity: from 9 core switches to 3. Thus, the background servers can, by themselves, saturate core capacity. Our remaining 18 servers continue to run the bidirectional client-server network pattern, with one important difference: rather than all servers having the same amount of load, they are partitioned into three classes. The first class chooses flow sizes from the Facebook cache server flow size distribution [112]; the other two classes choose flow sizes from larger $4\times$ and $16\times$ multiples of the Facebook distribution. In other words, the $4\times$ distribution is simply the regular distribution, except every value for flow size being multiplied by 4. Thus, the link utilizations of the $4\times$ and $16\times$ distributions are correspondingly higher than normal. While the regular distribution has a link utilization of roughly 100 Mbps in this case, the $4\times$ case and $16\times$ cases bring us to roughly 40% and 100% utilization respectively.

Figure 5.9 depicts the `select()` latency distribution for this scenario as a multi-series CDF. Each series represents the distribution for a single network path; there is one faulty path and two working paths in this case. The $x$-axis depicts nanosecond latency. Despite the combination of confounding factors, the faulty-path distribution remains significantly higher (skewed higher in the figure) than non-faulty paths (possessing similar distributions). Correspondingly, select() latency signal remains useful. Using 1-minute intervals, the chi-squared test output remained close to $p = 0$ when we considered all paths, and close to $p = 1$ when the faulty path was removed from the set of candidate paths; thus, we successfully find the faulty link in this scenario.

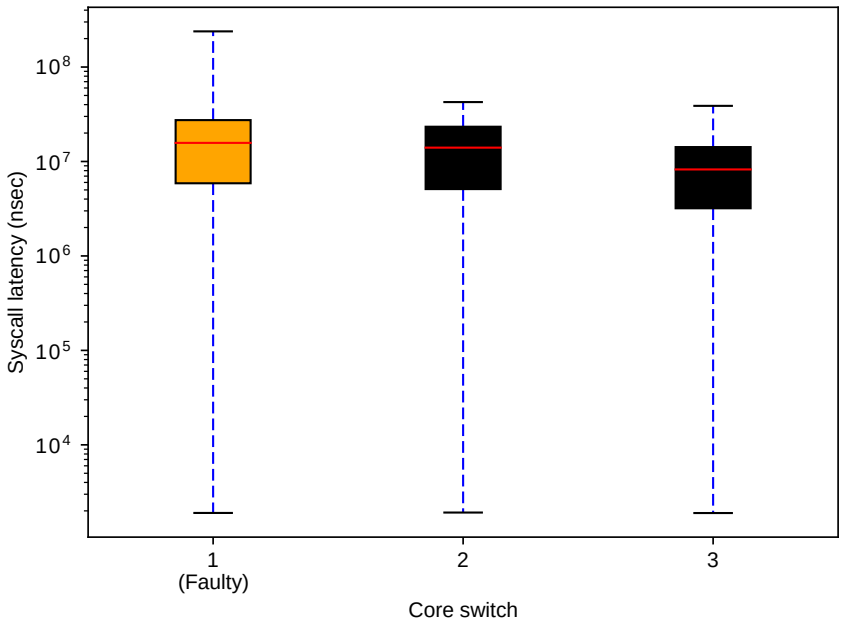



**Figure 5.9.** `select()` latencies per core switch for the oversubscribed background traffic uneven server test.

### 5.8.3 Sensitivity to TCP send buffer size

Our system call latency approach uses the latency of `select()`/`epoll()` calls by servers that are sending data over the network as a signal. As a reminder, when an application calls `select()`/`epoll()` (or `send()`) on a socket, the call returns (or successfully sends data) when there is enough room in the per-flow socket buffer to store the data being sent. Thus, the distribution of results is fundamentally impacted by the size of the per-flow socket buffer. Note that with the advent of send-side autotuning in modern operating systems, the size of this buffer is not static. In Linux, the kernel is configured with a minimum size, a default (initial) size and a maximum size, with the size growing and shrinking as necessary.

By default, our testbed systems are configured with a maximum send buffer size of 4 MB. Values for median buffer size vary from 1.5–2.5 MB for servers and 0.8–1.2 MB for clients in our bidirectional traffic pattern. To determine the sensitivity of our approach to buffer size,

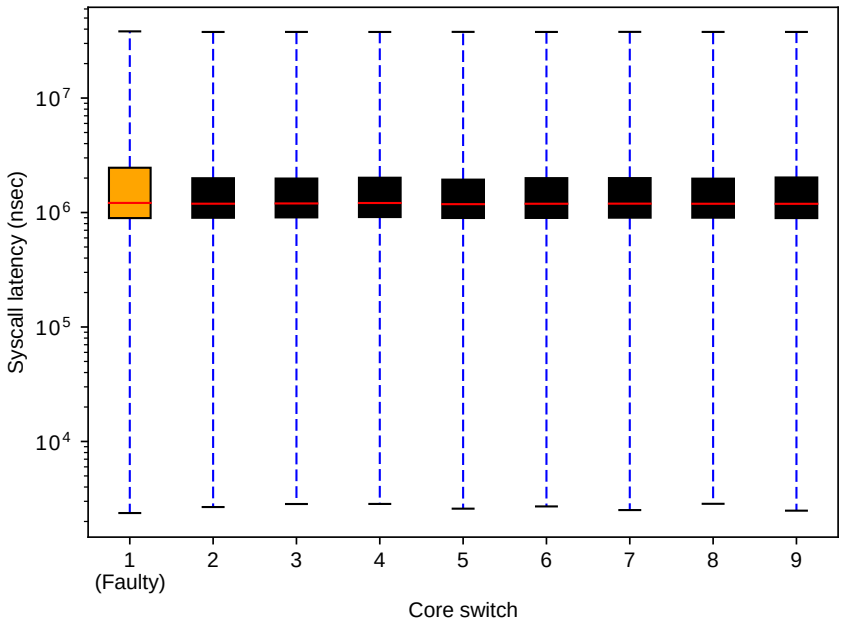



**Figure 5.10.** `select()` latencies per core switch for the 10-Gbps 16-KB socket send buffer test.

we first increased the buffer size to values of 8 and 16 MB. Then, we decreased the buffer size significantly, to values of 52, 26 and 16 KB (note that 16 KB is the default initial size). We noticed that TCP autotune prevents the buffer from growing too large even if the maximum is set significantly higher than default; accordingly, raising the maximum had little to no impact.

Reducing the maximum buffer size, however, is expected to have a larger impact on our metrics. Intuitively, a smaller buffer should correspond with higher overall buffer occupancy; thus, a larger percentage of `select()` and `send()` calls would need to wait until the buffer is drained. Buffer reductions should have the two-fold effect of shifting the `select()` latency distributions to the right—since more calls to `select()` wait for longer—and decreasing the distance between a faulty-path distribution and the normal case.

To test this case, we ran our bidirectional client-server traffic pattern with 1-Gbps NICs on our $k = 6$ testbed, and with 10-Gbps NICs on our three system setup. We ran a low-intensity (roughly 50 and 300 Mbps link utilization per client and server, respectively) and a high-intensity test (roughly 1–2 and 5–7 Gbps for clients and servers respectively in the 10-Gbps case, and

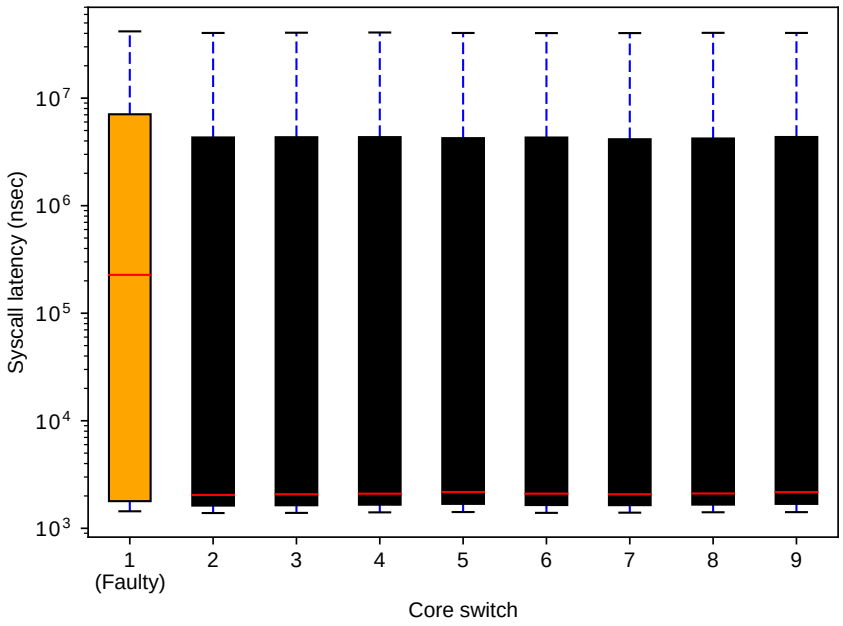



**Figure 5.11.** `select()` latencies per core switch for the no-offload test with CPU stress.

saturated server links in the 1-Gbps case). We noted that in all the low-intensity test cases, and in the 1-Gbps high-intensity cases, we were able to derive a clear signal and successfully pick out the faulty link using 1-minute intervals. In our worst-case scenario of 10-Gbps links and a 16-KB send buffer size, we note a significant skew to larger values by $\approx 3$ orders of magnitude, as predicted, as depicted in Figure 5.10. While the faulty distribution is closer to the non-faulty distributions, there is enough divergence to differentiate it from the pack, albeit requiring our statistical tests to run over longer time intervals. We note, however, that such small send socket buffer sizes are unlikely in real-world deployments.

### 5.8.4 The effect of NIC offloads

Contemporary NICs have a variety of offload features designed to cut down on server CPU utilization (which may be heavily utilized in datacenter environments). For example, TCP discretizes a continuous stream into individual packets determined by path MTU; a feature called ”TCP Segmentation Offload” allows the NIC to perform this discretization, saving CPU effort.

NICs typically implement three common offload features: TCP Segmentation Offload, Generic Segmentation Offload and Generic Receive Offload. All our previous experiments had these features turned on in every server. To test the sensitivity of our approach to the availability and state of these features, we re-ran the bidirectional client-server traffic pattern with all three offload features turned off. Figure 5.11 depicts the `select()` latency distribution for two clients and one server within our private testbed with all offloads turned off and a CPU utilization of 100% using the combination of programs described in the CPU stress test earlier. This scenario represents a sort of worst case for our approach due to the high CPU utilization required to push multiple gigabits of traffic without offload; despite this, our approach is still able to correctly flag the faulty path over a 30-second interval.

## 5.9 General-case topologies

The characteristics of the production datacenter we examined in this work lends itself to relatively simple diagnosis of which link is faulty based on per-link performance metric binning; however, alternative scenarios may exist where either traffic is not as evenly load balanced, or a different topology complicates the ability to subdivide link by hierarchy for the purposes of comparison (for example, different pods in a network might have different tree depths).

Our method fundamentally depends on finding roughly equivalent binnings for flow metrics, such that outliers can be found. In the case of a network with hot spots or complicated topologies, which can confound this process, we can still make headway by considering links at every multipath decision point as a single equivalency group. If one of these links exhibits outlier performance, the flows traversing the link can be marked individually as faulty rather than the link as a whole, and the flow and the traversed path can be submitted as input to a graph-based fault localization algorithm such as SCORE [84] or Gestalt [104]. We leave the evaluation of such a scenario to future work.

## 5.10 Summary

Having presented our primary contribution, we now revisit the partial-fault localization success criteria we presented earlier to determine how our methodology reaches and falls short of our goals. We claim considerable success on reaching every criteria described:

1. **Speed of detection** Traffic temporal stability plays a key part in determining partial-fault detection speed. Facebook web and cache server traffic possesses per-second stability—in other words, every second of traffic possesses similar aggregate characteristics compared to the seconds before and after—and allows us to successfully detect a partial fault with as little as 0.5% randomized packet loss in 10 seconds. Even as drop rates diminish to 0.1%, it is still possible to detect the partial fault responsible within two minutes.

   For temporally-variable traffic like Hadoop and WarmStorage, timescales around 1 (0.5% packet loss) or 2 (0.2% packet loss) minutes were sufficient, despite the heavy variations noted in traffic on a minute-by-minute basis [112]. In either case, we note a significant improvement over manual-intervention methods that can take hours [126].

2. **Sensitivity to minuscule faults and false negatives** Sensitivity to minuscule faults, and thus our false-negative rate, also depends on traffic. When examining Facebook web and cache traffic, our approach pinpointed packet-loss rates of 0.1%; when examining bulk Hadoop and WarmStorage we detected losses of 0.2% or greater. These rates are comport with studies revealing 0.1-0.2% ambient loss occurring in Facebook ToR switches [131]. We hypothesize that higher standing switch buffer utilization associated with Hadoop [131], coupled with greater temporal instability [112] are responsible for the greater ambient packet loss and thus reduced partial fault sensitivity by our methodology.

3. **Accuracy and false positives in the face of network variability** Due to centralized filtering of per-server fault indications and looking for consistent outliers, we did not encounter any false-positive events during the weeks that our prototype was active.

4. **Ability to detect varied faults** Our methodology detects faults due to their impact on perceived packet loss and latency, thus making it agnostic to the underlying cause of the fault. Furthermore, we were able to detect changing aggregate traffic characteristics due to routing anomalies caused by a control-plane fault, thus indicating that any event with a performance impact should be detectable by our system.

5. **Ability to pinpoint fault location and correlate application impact** By construction, our system is able to both localize faults to a particular link, switch or port and allow users to determine specifically what traffic is impacted by the fault located at each component.

6. **Resilience against faulty monitors** Since our method does not depend against switch-based monitoring and since it does not require every server to successfully detect a fault in order to pinpoint the fault, we claim that it is wholly resilient to faulty or unreliable switch-based monitoring and partially resilient to unreliable servers that incorrectly fail to diagnose faults. Note, however, that we do not claim resilience against malicious and coordinated incorrect reporting of faults.

7. **Computation, storage and network overheads** Our methodology minimizes network-switch CPU requirements (since packet marking uses ASIC support) and servers (empirically measured). It also imposes low network and storage overheads for aggregated data that is sent to the centralized controller that performs the actual localization step.

8. **Need to modify hardware and application software** Finally, our methodology requires no user application or server kernel changes (assuming, of course, utilization of recent Linux kernels) and leverages existing switch features for path marking.

Thus, we consider our equivalence-set based approach as providing an effective and realizable solution for quickly and accurate localizing partial faults within contemporary data-center environments, in a manner that is fundamentally agnostic of the root-cause of the fault and that applies to varying kinds of applications. Next, we will consider how our approach may be impacted within so-called ‘Cloud’ or ‘Tenant’ centric datacenters.

# Acknowledgments

Chapter 5, not including Section 5.6, is an adapted reprint of the material as it appears in "Partial datacenter fault detection and localization", which was published in USENIX NSDI 2017. The dissertation author was the primary author of this paper. The paper was co-authored with Hongyi (James) Zeng and Jasmeet Bagga of Facebook, and Alex C. Snoeren of UC San Diego.

Section 5.6 contains material currently being prepared for submission to IEEE Transactions on Networking. The dissertation author was primary author for this material. The work is co-authored with Hongyi (James) Zeng and Jasmeet Bagga of Facebook, and Rajdeep Das and Alex C. Snoeren of UC San Diego.

The original work as published in USENIX NSDI 2017 was supported in part by the National Science Foundation through grants CNS-1422240 and CNS-1564185. We are indebted to Aaron Schulman, Ariana Mirian, Danny Huang, Sunjay Cauligi, our shepherd, Dave Maltz, and the anonymous reviewers for their feedback on the originally published manuscript. Alex Eckert, Alexei Starovoitov, Daniel Neiter, Hany Morsy, Jimmy Williams, Nick Davies, Petr Lapukhov and Yuliy Pisetsky provided invaluable insight into the inner workings of Facebook services. Finally, we thank Omar Baldonado for his continuing support.

# Chapter 6

# Monitoring virtualized cloud datacenters

Previously, we focused on private 'first-party' datacenters that are owned and operated by a single entity that controls all of the infrastructure and traffic within the datacenter. Of ever-increasing importance, however, is the so-called 'cloud' datacenter, whose operators lease computing and networking infrastructure to third-party tenants.

Web service operators and IT-dependent enterprises alike are increasingly relying on cloud providers to address their computational needs. Cloud tenants simultaneously expect high reliability and performance in order to deliver quality service for their own users. Performance and reliability expectations are typically codified through service level agreements (SLAs). It is critical for cloud providers to detect and mitigate infrastructure faults in a timely fashion.

Certain cloud datacenter characteristics complicate fault localization. To support potentially conflicting customer use-cases, cloud providers employ extensive virtualization: host virtualization provides isolation and resource multiplexing so that multiple tenants can share the same server. Similarly, network virtualization allows tenants to operate within cloud datacenters without undue complexity or contention. Rather than sharing IP addresses and logical network topology with other customers, tenants typically operate inside of virtual networks within a software-defined network (SDN) overlay provided by the datacenter operator.

Much is known about managing physical datacenter infrastructure faults. On the other hand, comparatively little has been published regarding the operational realities of managing

virtualized, multi-tenant networks utilizing logical network overlays networks to provide strong isolation in multi-tenant datacenters. While the end goal of fault monitoring is the same, virtualized networks impart both additional management powers and monitoring challenges in the form of indirection layers, black-box tenants, and additional infrastructure complexity.

As cloud datacenters increase in scale, complexity and popularity, it is important to consider reliability and fault localization in the context of cloud-specific challenges. This includes characterizing the additional complexities incurred by virtualized tenant datacenters, the types of faults that may occur (above and beyond those that affect non-virtualized private datacenters) and the effectiveness of cloud datacenter monitoring.

We present a first look into the nuances of monitoring these "virtualized" networks through the lens of Microsoft Azure, a large-scale cloud provider, focusing on tenant virtual network (VNET) monitoring. Specifically, we describe VNET Pingmesh, a VNET liveness and latency monitoring system, focusing on the challenges associated with building a fault monitoring system in a cloud environment. In addition, we present a preliminary study of the data it has collected in production, allowing us to answer these high-level questions:

1. How can cloud operators (as opposed to tenants) effectively monitor VNET performance, given black-box tenants? Can physical network monitoring tools [64, 113, 133] be usefully adapted to virtualized networks? How does virtualization impact monitoring precision?
2. What additional risks—beyond those also present in non-cloud datacenters—do virtual environments pose? What additional classes of faults occur, and how might they impact tenant performance?

While VNET Pingmesh has yielded several monitoring and fault diagnosis wins in production, we find that subtle interactions between SDN network virtualization, tenant behavior, and non-network infrastructure can reduce monitoring precision—complicating fault diagnosis by either raising alerts that do not correspond with customer impact, or potentially overshadowing the impact of actual network faults.

We highlight two specific interactions involving local disk I/O and tenant-deployed middleboxes. In both cases, VNET Pingmesh measurements suggested network performance anomalies yet customers were not complaining. Moreover, physical network monitoring tools indicated the underlying datacenter network was performing well. We diagnose the specific mechanisms that cause these measurements, finding that interactions involving both SDN infrastructure and tenant behavior together can induce false-positive indications of poor network performance despite a lack of actual customer impact. Finally, we describe how we account for each effect. Our preliminary experience suggests two takeaway lessons:

*VNET monitoring must be subject to ongoing validation to ensure precision.* While confounding effects can be handled when found, the ever-increasing search space of cross-layer interactions coupled with ever-expanding network feature sets suggests that handling all possible effects *a priori* is infeasible, turning VNET performance validation into an ongoing task.

*End-host behavior can induce networking anomalies and complicate mitigation.* End-host participation in the network plane, including the use of software virtual switches, means that faults or performance anomalies hitherto contained within individual end hosts can now affect disparate servers across disjoint virtual networks. Even when faults are diagnosed, mitigation is not always straightforward: while an operator can route around damage in a physical network, she may not be able to "route around" virtual switches or reboot misbehaving servers as easily.

While we do not evaluate this dissertation's main contribution—specifically, our partial-fault localization system—at Microsoft Azure, we do speculate on how the additional challenges imposed by cloud datacenters may impact our system's operation.

## 6.1 Datacenter fault monitoring

Contemporary large-scale, multi-path datacenter network fabrics have been subject to considerable scrutiny; various studies have provided both taxonomies of common network failures [64, 117, 126, 131, 133], networked application performance [112, 126], and methods for

pinpointing the cause and location of performance-sapping faults [22, 23, 113, 126]. In addition to the contribution made by this dissertation, network performance monitoring and anomaly detection has received significant study. Contemporary approaches include active-probing techniques, where specially crafted traffic is injected into the network in order to ascertain liveness and adequate performance [11, 59, 64, 130]. Tracing infrastructures correlate network behavior with packet network paths, in case a fault is deep within the network core [22, 133]. Server statistics [23, 100, 112] and switch counters [38] are also used to monitor networks.

VNETs may be impacted by the same ailments as physical networks, and accordingly tools such as Pingmesh [64] do see use in multi-tenant datacenter networks. However, they are not always easily applied; customer applications may run within a VM, where neither application metrics, OS level metrics or the ability to run code are available to the network operator. Some strategies can still be applied, however; VMs often use a server-based virtual switch to access the network that the network operator does control, allowing the recovery of some transport-level statistics [57] or other potentially useful indications like Vswitch drop counters.

Microsoft Azure leverages a monitoring system that can be deployed by a datacenter operator without tenant involvement, yet accurately reflects customer experience. Moreover, in order to meet stringent SLA requirements, it is lightweight enough to be run continuously. After describing the tool below, we examine the statistics gathered from the deployment of this system on a large cloud providers network and discuss what they reveal about the impact of VNET overlays on network performance.

## 6.2 Azure VNET overview

Microsoft Azure consists of datacenters within geographic regions across the world. Each region can contain several datacenters, each of which can contain several clusters. A cluster contains multiple racks aggregating (physical) servers. Azure VMs are multiplexed across these servers, and are organized into isolated, non-interfering VNETs.

### 6.2.1 VNET addressing and packet handling

VNETs are topologically flat, L3-addressed IP network overlays built atop the physical topology. VNETs can aggregate thousands of individual VMs, each with one or more virtual NICs. Each NIC has a customer-chosen virtualized "customer IP address" or "CA". Customer applications on a VM will address other VMs in the VNET using CAs.

VM network access is provided via a bespoke physical-server-based virtual switch ("Vswitch") called VFP [54]. Each VFP instance has several virtual ports; one is connected to a physical NIC and the others to VM virtual NICs. An outbound VM NIC packet is transformed by per-port processing layers, each with distinct tasks (like metering traffic or implementing customer ACLs [54]). One layer translates CAs to an IP "physical address" ("PA") that is routable on the underlying network, providing VNET isolation. CA $\Rightarrow$ PA mappings are dynamic; actions like creating or deleting a VNET or VM can change mappings. Mappings reside in a reliable distributed directory. A per-server userspace agent receives updated mappings from this directory as network allocation state evolves. When a VM starts a network flow to a given CA, the CA $\Rightarrow$ PA mapping for the flow is queried from the userspace agent and cached in the kernel datapath; subsequent packets leverage this cache. Cached mappings are evicted after inactivity timeouts.

### 6.2.2 Monitoring via VNET Pingmesh

Reliable, low-latency inter-VM connectivity is essential to network performance. Ping-based liveness testing aids physical network monitoring [11, 64], suggesting one strategy may be to uplift such systems and measure liveness between VMs within VNETs. Thus, we devise VNET Pingmesh to measure VNET-layer network quality from the tenant's point of view. For VNETs comprising $\geq 1$ CA, we track full-mesh CA$\Rightarrow$CA ping latencies (cf. Pingmesh [64], where full-mesh statistics are impractical due to scaling considerations). While straightforward in principle, several cloud-specific subtleties emerge that we must account for, above and beyond existing challenges inherited from physical-layer systems like physical Pingmesh.

Fundamentally, we need to measure VM-observed network health without VM access (preventing VM deployment of Pingmesh or monitoring VM TCP statistics). Experience shows that VM-host monitoring alone does not suffice due to the potential for VNET-specific performance anomalies. To measure VNETs, we leverage Vswitch-level control and inject specially crafted TCP-based pings that appear to the network as if they were generated by the VM itself. The receiving Vswitch transparently responds to these pings; VMs never see these ping packets, enabling us to measure observed VNET latency without disturbing customers.

While physical Pingmesh measures end-to-end latency from a userspace agent on the physical server OS, VNET Pingmesh latency is measured from the kernel of the physical server where VFP operates. Thus, VNET Pingmesh probes—when they hit in the mapping lookup cache—are not subject to context switching and scheduler variation, and thus can deliver lower (likely closer to actual) latency measurements than physical Pingmesh. On the other hand, unlike physical-layer tools, we have to anticipate and account for tenant-environment and SDN-specific confounding behaviors. Private datacenters are advantaged because, at a high level, a single entity drives all network behaviors and configuration. Similarly, non-SDN datacenters are relatively uncomplicated and well-understood. In contrast, cloud networks are driven by the network operator and tenants and possess additional moving parts. Thus, if unaccounted for, these qualities may reduce monitoring precision.

Some interactions are easily foreseeable and accounted for by the initial VNET Pingmesh deployment. Other interactions, however, were unexpected and discovered only during validation of production statistics; we discuss these statistics and the cases they reveal in Section 6.3. Both classes of impacts result from VNET Pingmesh being impacted by tenant actions and reporting measurement artifacts that do not correspond with actual customer impact. While the fixes may be straightforward (or not) in either case, accounting for every possible effect is difficult in a complex system with ever-increasing scale and feature sets.

**Interactions with customer rulesets**

While full intra-VNET VM connectivity is desirable, customers can install ACL rules to support firewalls and gateways which can interfere with VNET Pingmesh. To avoid ACL traps (and avoid metering pings as customer traffic), we simply inject pings after ACL and meter VFP layers [54]. Similarly, customers can install "User-Defined Routes" ("UDRs") to support middleboxes. While ACL interactions were easy to foresee and account for, UDRs resulted in unanticipated interactions with VNET Pingmesh. We discuss UDRs further in Section 6.3.2.

**Interactions with customer VM lifecycle**

Tenant networks possess significant churn due to unpredictable creation or destruction of VMs and VNETs. If a VM is shutdown or disconnected, VFP disconnects the corresponding Vswitch virtual port, and pings to the port are ignored. Thus, if a VM is destroyed, one can expect transient ping failures until the VM is delisted. However, rather than only causing occasional loss spikes, this effect can cause severe, persistent and cyclical false-positive loss indications within certain VNETs. Specifically, some clients shut down multiple VMs during non-business hours to save costs. Unaccounted for, this can manifest as large-scale loss.

Despite not anticipating how common VM shutdowns would be in practice, we foresaw the need to filter statistics. We correlate VM and VM-NIC liveness from separate monitoring systems with VNET Pingmesh and filtered out connectivity loss indications from shutdown VMs, and are investigating Vswitch mechanisms that can disambiguate between a failed VM and an administratively disabled one.

### 6.2.3 Successful monitoring outcomes

VNET Pingmesh indicates good overall VNET performance; in a randomly chosen period, across all Azure, $\geq$ 94% of VNETs meet or exceed latency requirements $\geq$99.999% of the time when considering per-VM 5-minute averages. Furthermore, measurements indicate that Azure achieves just under five-9's connectivity across every VNET in a typical hour. Despite

favorable high-level metrics, large network scale means that faults do occur. Here, we discuss representative success stories before focusing on confounding effects for monitoring.

In March 2017, a datacenter incident prompted various clients to raise support requests complaining about high VM-to-storage latency. Initially, storage was suspected, however, a correlated VNET Pingmesh latency spike suggested a network cause. No physical network or 'Canary' (VMs deployed in each rack in the datacenter that periodically initiate connections on various common ports, monitor, and alert on these connection's failures) alarms were triggered. Later correlation with physical Pingmesh along with address resolution failures during the affected time-period confirmed a network fault. Detailed investigation revealed a linecard configuration and attendant congestion as the root cause. Another incident involved customer VM connectivity issues, confirmed through a transient but significant drop in VNET Pingmesh connectivity. Correlation with other monitoring systems identified a ToR reboot as the root cause.

In both cases, VNET Pingmesh, coupled with time-based correlation with other monitoring systems proved invaluable in incident diagnosis. In these instances, VNET Pingmesh alarms were true positives: its latency metrics correlated strongly with customer-reported impacts. On other occasions, however, VNET Pingmesh's measurements frustrated operators, both due to wasted effort diagnosing alerts with no customer impact, and by failing to identify actual performance issues. We discuss these realities in the next section.

## 6.3 Measuring Azure VNETs

In a network comprising O(1M) virtual servers, incidents are inevitable. Diagnosing incidents with VNET Pingmesh depends on monitoring precision, which cloud-network-specific effects can impact. Previously, we described several confounding effects that we anticipated and accounted for during VNET Pingmesh's initial design. Ongoing validation of our statistics revealed further interactions that, while simply understood with the benefit of hindsight, were not apparent before large-scale rollout. Here, we describe the metrics captured by VNET

Pingmesh and certain unforeseen interactions that yielded false-positive indications of poor network performance, despite no actual impact to customers.

While we have adapted VNET Pingmesh to account for these interactions, we highlight high-level similarities in these cases that suggest that accounting for every possible confounding effect *a priori* is unrealistic. First, each issue was influenced by actions driven by non-network actors: either another layer of Azure infrastructure, or by tenant network usage. Second, each issue was an edge case, which both complicated initial analysis and can be hard to speculate about. Third, the continued growth of Azure's feature set both increases the future likelihood of such interactions and complicates the task of searching the state space of all possible interactions *a priori*. Thus, even though accounting for a given issue is relatively simple, it may have to be a reactive process as tenant networking continues to evolve.

### 6.3.1 Server utilization and fake latency spikes

Physical network latency can be explained by differences in locality (inter-pod latency is higher than intra-pod), queuing delay (causing relatively small increases in latency) and packet loss (larger, multi-millisecond increases in TCP latency). Deviations from baseline performance have been considered as evidence of network faults [11, 64, 113]. While VNET latency anomalies often correlate with physical network faults, we discovered interactions between VNET infrastructure and other parts of the Azure infrastructure could confound our ability to determine if a high-latency indication was an actual customer issue or a false-positive.

Pingmesh [64] leverages physical network structure and locality to present a 2-D heatmap network view, where specific failures yield characteristic visual patterns. VNETs discard that structure due to the locality-independent flat address space model. However, we can project VNET latency data onto the underlying physical network. Figure 6.1 depicts latency measured in a single cluster over one hour, projected to physical rack. Every cell depicts latency between a source (x-axis) and destination (y-axis) rack, measured as the average latency of all VMs in all VNETs present in all servers in the source rack. Several interesting features emerge:

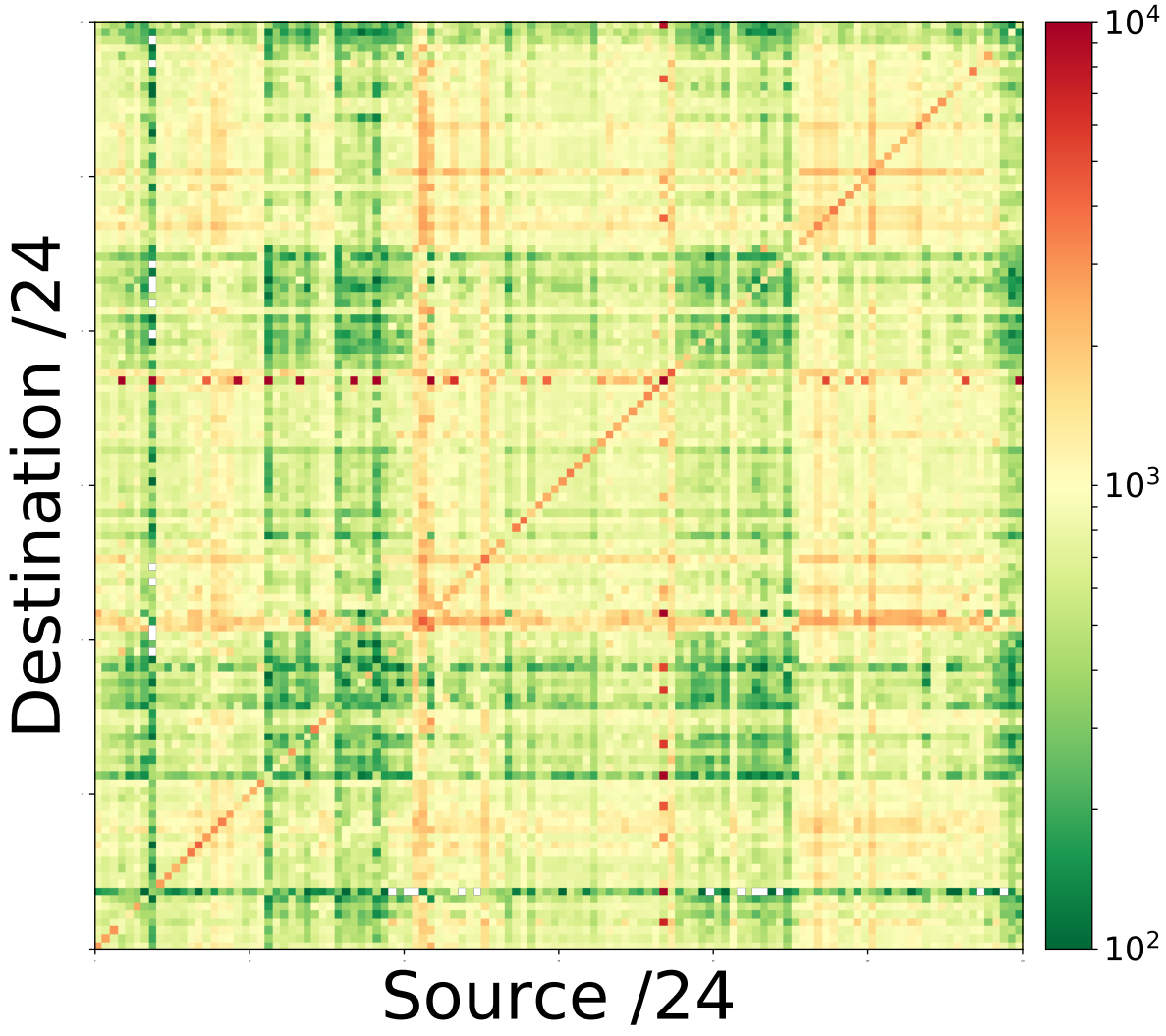



**Figure 6.1.** Rack-level heatmap

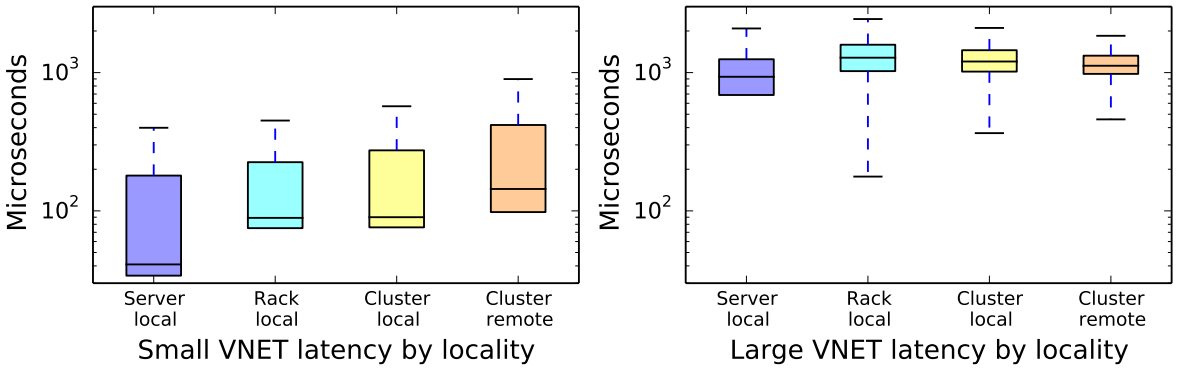



**Figure 6.2.** VNET latency vs. locality

1. Latency averages are on the order of hundreds of microseconds to a millisecond, higher than expected for datacenters. Intra-rack latency is $\geq 2.5\times$ inter-rack latency on average.
2. Several racks have an average latency of $\geq 1.5\times$ the median rack, delineated by darkly-colored intersecting horizontal and vertical stripes. On the other hand, some racks experience ping latency averages an order-of-magnitude smaller than other servers.
3. While omitted for space reasons, other clusters contain servers with outbound latency (pings generated by the server) $\geq 2\times$ inbound latency.

These effects are due to misses in the kernel CA $\Rightarrow$ PA mapping cache. Since VNET Pingmesh iterates through CAs consecutively, large VNETs with sparse traffic matrices may lead to cache misses for a given ping, unless the VM was actively communicating with the pinged VM. High average latencies can thus be explained by large VNETs—as a large fraction of measured latencies are due to large VNETs, their performance dominates. High intra-rack latency can also be explained by large VNETs; since VM placement within a cluster is random, pinging a VM in the same rack is more likely for large VNETs. Cold stripes correspond with either servers handling VMs within small VNETs exclusively (if the VNET size is small enough, mappings will always be within the cache due to VNET Pingmesh) or VMs in large VNETs that constantly communicate with other VMs.

Thus, for small VNETs, VNET Pingmesh measures actual network latency from the local Vswitch to the remote server's Vswitch, and comports with physical Pingmesh behavior. Figure 6.2 depicts the latency for small and large VNETs, bucketed by the locality of the destination. Small VNET latency values are on the order of 10s of microseconds at most for intra-cluster destinations, and are correlated with the locality of the destination server. Large VNET latency is higher, cluster-dependent, and noisier. Latencies range from 100s of microseconds to a millisecond depending on cluster, with a deviation on par with the total latency for small VNETs. Locality and latency trends vanish—instead, mapping-lookup context-switch jitter dominates.

**Table 6.1.** Probability that physical layer Pingmesh $\geq 1.5\times$ cluster average if VNET latency $\geq 2\times$ average for a VM.

| Category | % incidents | % physical anomalous |
|---|---|---|
| **Total** | 100 | 40.90 |
| **Large VNETs only** | 59.00 | 23.50 |
| **Small VNETs only** | 22.60 | 55.00 |
| **Large and small VNETs** | 18.40 | 79.00 |
| **Server->ToR packet loss** | 2.20 | 92.60 |

Meanwhile, a stubborn minority of servers yielded consistently high latency (i.e. $\geq 2\times$ other servers for the same VNET) for large VNETs, despite low average resource utilization and without any corresponding issues for co-located small VNETs or physical latency. Over a randomly picked hour with no active networking alerts, we considered servers hosting $\geq 1$ VM with an average outbound ping latency of $\geq 2\times$ the average within its corresponding VNET. We categorize each server into one of three categories, depending on whether all of the VMs onboard with poor latency belong to large VNETs only, small VNETs only, or a mix of large and small VNETs. Table 6.1 depicts the probability that physical Pingmesh reports a latency anomaly on a server, given that at least one VM on the server has $\geq$ 2x the average latency for its VNET, grouped by the kinds of VNET impacted. We call physical latency anomalous if the average server ping latency is $\geq$ 1.5x the cluster average.

Almost 20% of the time, both large and small VNETs are impacted. These seem to correlate highly with network fabric based latency woes—nearly 80% of the time, physical Pingmesh is also impacted in these cases. On the other hand, around 60% of high latency indications impact large VNETs only. A relatively small fraction of this number includes VNETs that may also be impacted by physical network fabric issues (only 23.5% of the incidences coincide with physical Pingmesh latency anomalies) suggesting that these false-positives may confound detection if not accounted for.

Thus, as initially deployed, VNET Pingmesh complicates large VNET latency analysis in two ways. First, mapping lookup jitter and magnitude may wash out actual network latency spikes. Second, two effects confound interpreting latency: traffic patterns can induce cold spots, and mapping lookups can cause hot spots. Thus, VNET Pingmesh may not provide an accurate picture of customer-experienced latency; some high latency indications are not performance concerns, but still generate occasional monitoring alerts.

Some VNET-only latency spikes do impact customer performance, however, so effective diagnosis is critical. Does the blame rest within VNET, or an external factor?

**Finding the root cause.**

The breakthrough was a VNET latency alert correlating with an OS-update roll-out performed by an internal team. Detailed investigation showed that sustained high disk I/O was interfering with user-space address mapping lookups due to a logging statement blocked on disk I/O. Thus, while small VNETs and existing (mapping-cache hitting) connections were not impacted, connections requiring a mapping lookup from user-space were. (Client disk traffic was not impacted due to storage disaggregation.) Subsequent analysis of high VNET latency indications across Azure showed a correlation between server I/O utilization and high (false-positive) latency indications.

To quantify this effect, we examined the prevalence of latency alerts as a function of server utilization in terms of both disk I/O and CPU. Figure 6.3 plots the likelihood that average VNET latency for a server exceeds 2 milliseconds, as a function of disk and CPU utility. Both graphs are normalized to the baseline probability at nominal utilization levels. We see a clear correlation between utilization and latency; as average disk utility passes 10%, we see a sharp increase in likelihood that latency is past acceptable boundaries.

We surmise that the same should be true for CPU utilization; in particular, that a heavily loaded CPU may result in effects such as delayed scheduling for crucial VNET management processes. One confounding factor is that CPU use can be bursty; since we measure utilization as a 5-minute average, we cannot distinguish between a server CPU constantly at 5% utilization

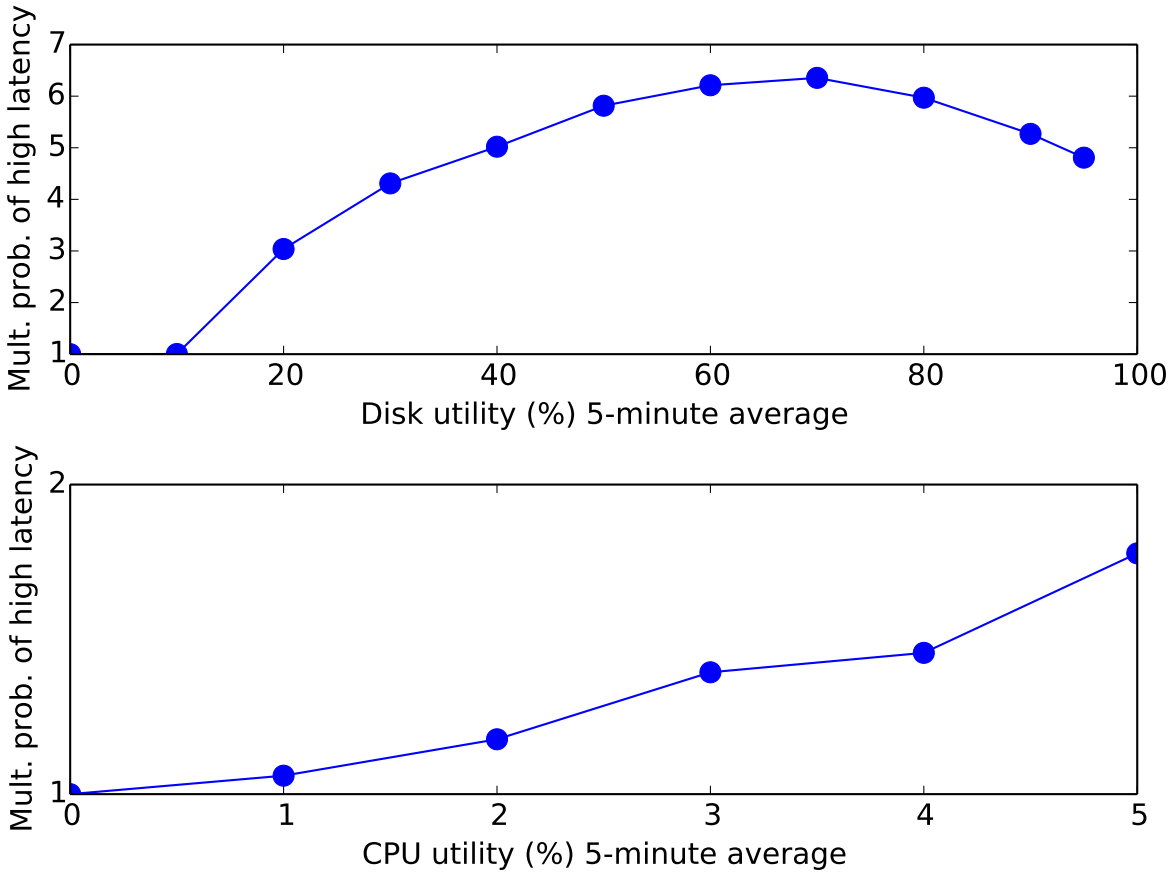



**Figure 6.3.** Multiplicative likelihood of high VNET Pingmesh latency vs. server utilization.

(unlikely to incur significant increases in VNET latency) or at 100% utilization for roughly 5% of the time—which would significantly impact latency during that period. Even so, a (weaker) correlation between CPU utilization and high latency emerges.

**Solution.**

Having diagnosed the mechanism for VNET Pingmesh false positives, recovering meaningful latency measurements for large VNETs is simple—we modify VNET Pingmesh to ping more than once, and consider latencies after the first ping to show network latency while the first ping can reveal mapping latency for large VNETs. In addition, modifying the mapping-lookup user-space agent removes the interaction with disk utilization.

These results suggest that heuristically, we can use a simple classification for latency anomalies—if only large VNET mapping latency is impacted, we may investigate VNET infrastructure and server metrics first; if multiple VNETs of differing sizes suffer, we may suspect a physical network root cause. Experience with physical network Pingmesh shows that the amount of latency elevation can suggest the cause: small increases or jitter can correspond to high buffer occupancy, while large increases may correspond to packet loss. Unfortunately, the same is not true with VNET Pingmesh: the complexity of address mapping lookups means that a variety of contending effects can increase latency by unpredictable amounts (server utilization can drown out latency caused by the network) or decrease latency (traffic pattern), making it harder to reason about differences in latency measurements over time.

### 6.3.2 Middleboxes and fake connectivity loss

Connectivity measurements also revealed persistent subnet-level connectivity loss, within several VNETs. Despite alarming connectivity statistics for these VNETs, no customer issues were raised. Investigating confirmed it was a false-positive indication of connectivity loss related to a relatively subtle monitoring bug. Specifically, 'User-Defined Routes' (or 'UDRs', commonly supporting middleboxes like gateways and firewalls) were, for some VNETs, stealing Pingmesh ping responses, manifesting as a subnet-level loss in connectivity for VNETs with tunneling

rules. Diagnosis was confounded by the infrequent incidence of the problem; a relative minority of VNETs possess UDRs, a minority of which actually triggered the packet loss. Furthermore, the behavior of the UDRs (and thus VNET connectivity) were subject to tenant modification, providing limited windows of diagnosis opportunity for each VNET.

Once diagnosed, fixing the issue was simple; we simply modified VNET rulebase generation to prevent UDRs from processing VNET Pingmesh packets, fixing the connectivity metrics for the affected VNETs. In effect, this issue is akin to a routing black-hole, a classic failure mode for physical networks. However, certain differences applied for VNETs that complicated diagnosis. Physical routing black holes tend to have a large and obvious impact as application traffic ceases entirely; here, though, we were faced with an edge-case black hole only impacting monitoring traffic. Furthermore, while the network operator is responsible for physical routing, tenant networks have the additional difficulty of user-influenced black holes forming.

## 6.4 Implications for partial fault localization

While we have not adapted or evaluated this dissertation's primary contribution at Microsoft Azure, we speculate here on how virtualized tenant datacenters provide additional and significant challenges that can complicate our system's task.

1. **'Network' partial faults may be caused by servers.** Because servers are packet forwarding devices in VNETs, it naturally follows that server performance anomalies will now manifest, both to customers and initially to network operators, as network anomalies. Thus, fault diagnosis can gain complexity over first-party datacenters; servers may run more black-box software contending for resources than network switches, possibly complicating root cause analysis.
2. **Server-based partial fault mitigation may be harder.** Server-driven, network-impacting partial fault mitigation can be daunting since, unlike the physical network case, datacenter operators can neither route around faults (it is not obvious how to avoid impacted Vswitches

or VMs) nor reboot or replace the server without potentially disrupting customers, though VM live-migration will help alleviate this issue when deployed.

3. **Path information is harder to recover.** Middleboxes, load balancers, and user-defined routing prevent us from unambiguously recovering full flow network path using a single-switch marking as in the Facebook case. In particular, we can no longer rely on topology characteristics to lock down flow path. Note that this is a surmountable challenge using techniques like traceroute [22] or the alternative strategies we discuss in Section 5.3.1.

4. **Equivalent sets may be harder to form.** In the Facebook scenario, each server possessed thousands of active flows each minute traversing the network core. Within Azure, different tenants display varying network utilization and traffic patterns. While some VNETs may allow formation of per-server measured equivalence sets due to sufficient traffic pattern volume and spread, others may possess too sparse a traffic matrix to satisfy the requirements for forming an equivalence set.

5. **Analyzing application performance impact is more difficult.** Since tenant VMs are black boxes to the datacenter operator, metrics like TCP retransmits are harder to acquire, though still possible via Vswitch interposition techniques [57]. Application metrics are no longer available due to privacy concerns. Active-probing techniques are usable, but requires care. Specifically, repurposing physical-network targeting techniques [64] for monitoring virtual network overlays requires understanding the additional complexities and modes of operation that the overlay imposes on network packet handling. Without doing so, unexpected interactions may lead to inaccurate connectivity statistics (tunnels stealing probe packets) or latency statistics (we may be monitoring either network fabric latency or server mapping latency depending on the VNET in question).

Despite these challenges, we surmise that we may be able to form equivalence sets at a higher level of the topology. Rather than examine traffic collected at a single VM or server, we may possess enough traffic volume to form equivalence sets if we consider aggregate traffic at

ToR switches. For example, one possible equivalence set could be the uplinks connecting the ToR to the pod aggregation switches. We defer a full evaluation, however, to future work.

# Acknowledgments

Chapter 6 is an adapted reprint of material currently being prepared for submission to ACM IMC 2018. The dissertation author was primary author for this material. The work is co-authored with Deepak Bansal, David Brumley, Harish Kumar Chandrappa, Parag Sharma, Rishabh Tewari and Behnaz Arzani of Microsoft, and Alex C. Snoeren of UC San Diego.

Nimish Aggarwal, Milan Dasgupta, Abhishek Ellore Sreenath, Alex Fu, Daniel Firestone, Shefali Garg, Deven Jagasia, Abhishek Kumar, Neeraj Motwani, Jae Park, Chaitanya Raje, Pranjal Shrivastava, Abhishek Shukla, Qaseem Zuhair, Michal Zygmunt, and the entire VNET and VFP teams at Microsoft provided valuable insight into Azure Networking.

# Chapter 7

# Conclusions and future work

This dissertation asserts that the fundamental design goals of contemporary datacenter networks, and the resulting characteristics of datacenter topologies and application traffic loads, enabled an outlier-analysis based partial-fault localization methodology. In particular, it claimed that the combination of large-scale load-balanced multipath topologies and high-volume datacenter traffic enables simple, low-overhead and application and root-cause agnostic partial fault localization via passive link-by-link outlier analysis of application network performance.

For a variety of different applications, within real-world datacenters and within artificially constrained testbeds with pathologically compromised conditions, this claim bore fruit—if we could create 'Equivalence Sets' out of components such as links or switches in the network, we could interpret (poor) performance outliers with faulty components with a high degree of precision (low false-positives) and recall (low false-negatives).

At the same time, we characterized the cases where this approach fell short for physical network fabrics; for example, non-homogeneous network fabrics where we could not form equivalence sets due to effects such as unequal link capacity or buffering. Even so, this was borne of an inability to form equivalence sets, not due to a failing in the overall concept.

However, we may be unable to form equivalence sets in certain scenarios; in particular, the advent of virtualized, third party tenant 'Cloud' datacenters means we cannot expect server-by-server similarities found in more homogeneous Facebook datacenters. While a case may be

made for applying our approach to the underlying physical network fabric in a cloud datacenter (assuming we can still correlate flow performance to network component; they do, after all, still use equal cost paths and aggregate large amounts of traffic), it does not commonly apply to server-by-server measurements that are commonly tracked within cloud datacenters.

However, it is still imperative for cloud datacenter operators to be able to detect and pinpoint failures within virtualized network overlays. While there is no shortage of physical network monitoring tools that can be adapted for use within cloud networks, these tools have an uphill battle. While virtual network overlays do provide the illusion of an isolated, regular physical network to tenant applications, maintaining this facade means that the network overlays perform significant amounts of interposition on customer traffic; for example, remapping virtualized packet addresses or handling broadcast traffic for protocols like ARP. The net result is that server-based performance problems can now manifest as network performance problems, across a wide variety of private virtual networks. In particular, they can manifest in addition to, and with similar symptoms as physical-network partial faults, randomly delaying or dropping network traffic. From the point of view of a tenant network, the outcome is the same.

Thus, the fundamental problem to be solved is still one of attribution. In a physical network, we seek to determine which link or switch causes loss or delay. In a virtual network, we want to know if a given issue is due to the physical network or not; if it is not, we seek to determine exactly which portion of the virtual network overlay is at fault. Some differences do apply, however. In a physical network, it is often the case that after a fault is pinpointed, it can be routed around and thus mitigated; faulty components can be replaced with relatively low impact. In the virtual network, however, pinpointing a fault is merely the beginning.

Rather, since partial fault behaviour may be due to server-level anomalies, and since servers are both network-plane devices and hosts to customer services and traffic, we cannot easily mitigate faults by rebooting servers or routing around virtualization infrastructure due to the high business costs of disrupting third-party customer traffic. Diagnosing faults and solving them in-place can be complicated affairs due to the presence of varied, black-box style

applications competing for the same resources as networking infrastructure. Greater visibility and the ability to rapidly pinpoint a fault is not enough.

While enhancing passive monitoring capabilities for customer traffic likely has value for cloud networks, one possible future direction for cloud networks might be architectural—rather than enhancing the ability for network operators to find performance anomalies, it may be worth architecting virtualized networks in a way that disaggregates virtualization infrastructure from customer software, perhaps in a manner similar to existing storage disaggregation. Examining such possibilities is in keeping with an old maxim that 'an ounce of prevention is worth a pound of cure'. Whether this is necessary—that is, determining the impact that virtualized overlays has on real world customer traffic patterns—also remains an open line of inquiry.

# Bibliography


[1] ab - Apache HTTP server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html.

[2] Airline on-time performance. http://stat-computing.org/dataexpo/2009/.

[3] BPF Compiler Collection (BCC). https://github.com/iovisor/bcc/.

[4] Extending extended BPF. https://lwn.net/Articles/603983/.

[5] Hadoop. http://hadoop.apache.org/.

[6] HHVM. http://hhvm.com.

[7] Introduction to Redis. http://redis.io/topics/introduction.

[8] An Open Network Operating System. http://onosproject.org.

[9] Scribe (archived). https://github.com/facebookarchive/scribe.

[10] L. Abraham, J. Allen, O. Barykin, V. Borkar, B. Chopra, C. Gerea, D. Merl, J. Metzler, D. Reiss, S. Subramanian, J. L. Wiener, and O. Zed. Scuba: Diving into Data at Facebook. *Proceedings of the VLDB Endowment*, 6(11):1057–1067, Aug. 2013.

[11] A. Adams, P. Lapukhov, and H. Zeng. NetNORAD: Troubleshooting networks via end-to-end probing. https://code.facebook.com/posts/1534350660228025/netnorad-troubleshooting-networks-via-end-to-end-probing/, 2016.

[12] V. K. Adhikari, Y. Guo, F. Hao, V. Hilt, Z.-L. Zhang, M. Varvello, and M. Steiner. Measurement Study of Netflix, Hulu, and a Tale of Three CDNs. *IEEE/ACM Transactions on Networking*, 23(6):1984–1997, Dec. 2015.

[13] M. Aguilera, J. Mogul, J. Wiener, P. Reynolds, and A. Muthitacharoen. Performance Debugging for Distributed Systems of Black Boxes. SOSP 2003, Bolton Landing, NY, USA, 2003. ACM.

[14] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. SIGCOMM 2008, Seattle, WA, USA, 2008. ACM.

[15] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. NSDI 2010, San Jose, California, USA, 2010. USENIX Association.

[16] A. Alameldeen, M. Martin, C. Mauer, K. Moore, X. Min, M. Hill, D. Wood, and D. Sorin. Simulating a $2M commercial server on a $2K PC. *IEEE Computer*, 36(2):50–57, Feb. 2003.

[17] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese. CONGA: Distributed Congestion-aware Load Balancing for Datacenters. SIGCOMM 2014, Chicago, Illinois, USA, 2014. ACM.

[18] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data Center TCP (DCTCP). SIGCOMM 2010, New Delhi, India, 2010. ACM.

[19] A. Andreyev. Introducing data center fabric, the next-generation Facebook data center network. https://code.facebook.com/posts/360346274145943, 2014.

[20] Apache Software Foundation. Apache Hadoop 3.0.0 - Overview. https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/Metrics.html#datanode.

[21] G. Appenzeller, I. Keslassy, and N. McKeown. Sizing Router Buffers. SIGCOMM 2004, Portland, Oregon, USA, 2004. ACM.

[22] B. Arzani, S. Ciraci, L. Chamon, Y. Zhu, H. H. Liu, J. Padhye, B. T. Loo, and G. Outhred. 007: Democratically Finding the Cause of Packet Drops. NSDI 2018, Renton, WA, USA, 2018. USENIX Association.

[23] B. Arzani, S. Ciraci, B. T. Loo, A. Schuster, and G. Outhred. Taking the Blame Game out of Data Centers Operations with NetPoirot. SIGCOMM 2016, Florianopolis, Brazil, 2016. ACM.

[24] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload Analysis of a Large-scale Key-value Store. SIGMETRICS 2012, London, England, UK, 2012. ACM.

[25] P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. A. Maltz, and M. Zhang. Towards Highly Reliable Enterprise Network Services via Inference of Multi-level Dependencies. SIGCOMM 2007, Kyoto, Japan, 2007. ACM.

[26] D. Banerjee, V. Madduri, and M. Srivatsa. A Framework for Distributed Monitoring and Root Cause Analysis for Large IP Networks. IEEE International Symposium on Reliable Distributed Systems 2009, Niagara Falls, NY, USA, 2009. IEEE Computer Society.

[27] L. A. Barroso, J. Clidaras, and U. Hölzle. *The Datacenter as a Computer:An Introduction to the Design of Warehouse-Scale Machines*. Morgan & Claypool, 2nd edition, 2013.

[28] M. Becchi. From Poisson Processes to Self-Similarity: a Survey of Network Traffic Models. http://www1.cse.wustl.edu/~jain/cse567-06/ftp/traffic_models1/, 2008.

[29] A. Bechtolsheim, L. Dale, H. Holbrook, and A. Li. Why Big Data Needs Big Buffer Switches. https://www.arista.com/assets/data/pdf/Whitepapers/BigDataBigBuffers-WP.pdf, 2016.

[30] T. Benson, A. Akella, and D. A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. IMC 2010, Melbourne, Australia, 2010. ACM.

[31] T. Benson, A. Anand, A. Akella, and M. Zhang. Understanding Data Center Traffic Characteristics. *SIGCOMM Comput. Commun. Rev.*, 40(1):92–99, Jan. 2010.

[32] T. Benson, A. Anand, A. Akella, and M. Zhang. MicroTE: Fine Grained Traffic Engineering for Data Centers. CoNEXT 2011, Tokyo, Japan, 2011. ACM.

[33] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[34] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. TAO: Facebook's Distributed Data Store for the Social Graph. USENIX ATC 2013, San Jose, CA, USA, 2013. USENIX Association.

[35] M. Calder, R. Miao, K. Zarifis, E. Katz-Bassett, M. Yu, and J. Padhye. Don't Drop, Detour! SIGCOMM 2013, Hong Kong, China, 2013. ACM.

[36] C. Cardenas. Arista's Big Buffer B.S. http://packetpushers.net/aristas-big-buffer-b-s/, 2016.

[37] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, V. Jacobson, and A. Vahdat. TCP BBR congestion control comes to GCP your Internet just got faster. https://cloudplatform.googleblog.com/2017/07/TCP-BBR-congestion-control-comes-to-GCP-your-Internet-just-got-faster.html, 2017.

[38] J. D. Case, M. Fedor, M. L. Schoffstall, and J. Davin. RFC 1157: Simple Network Management Protocol (SNMP), 1990.

[39] M. Y. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer. Path-based Failure and Evolution Management. NSDI 2004, San Francisco, California, USA, 2004. USENIX Association.

[40] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem Determination in Large, Dynamic Internet Services. Dependable Systems and Networks 2002, Bethesda, MD, USA, 2002. IEEE Computer Society.

[41] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica. Managing Data Transfers in Computer Clusters with Orchestra. SIGCOMM 2011, Toronto, Ontario, Canada, 2011. ACM.

[42] Cisco. BGP Support for TTL Security Check. http://www.cisco.com/c/en/us/td/docs/ios/12_2s/feature/guide/fs_btsh.html.

[43] CNBC. Here's how billions of people use Google products, in one chart. https://www.cnbc.com/2017/05/18/google-user-numbers-youtube-android-drive-photos.html, 2017.

[44] CNN. Facebook hits 2 billion monthly users. http://money.cnn.com/2017/06/27/technology/facebook-2-billion-users/index.html, 2017.

[45] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. DevoFlow: Scaling Flow Management for High-performance Networks. SIGCOMM 2011, Toronto, Ontario, Canada, 2011. ACM.

[46] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. OSDI 2004, San Francisco, CA, USA, 2004. USENIX Association.

[47] C. Delimitrou, S. Sankar, A. Kansal, and C. Kozyrakis. ECHO: Recreating Network Traffic Maps for Datacenters with Tens of Thousands of Servers. IEEE International Symposium on Workload Characterization 2012, San Diego, CA, USA, 2012. IEEE Computer Society.

[48] N. Duffield, F. L. Presti, V. Paxson, and D. Towsley. Network Loss Tomography Using Striped Unicast Probes. *IEEE/ACM Transactions on Networking*, 14(4):697–710, Aug. 2006.

[49] N. Dukkipati, M. Kobayashi, R. Zhang-Shen, and N. McKeown. Processor Sharing Flows in the Internet. International Workshop on Quality of Service 2005, Passau, Germany, 2005. Springer-Verlag.

[50] D. Ersoz, M. S. Yousif, and C. R. Das. Characterizing Network Traffic in a Cluster-based, Multi-tier Data Center. International Conference on Distributed Computing Systems 2007, Toronto, Canada, 2007. IEEE Computer Society.

[51] Facebook. Facebook Warm Storage - next generation storage for Data Warehouse in Hadoop ecosystem. https://events.static.linuxfound.org/sites/events/files/slides/WarmStorage%20for%20Conference.pdf.

[52] N. Farrington and A. Andreyev. Facebook's Data Center Network Architecture. IEEE Optical Interconnects Conference 2013, Santa Fe, New Mexico, USA, May 2013. IEEE.

[53] N. Farrington, G. Porter, S. Radhakrishnan, H. H. Bazzaz, V. Subramanya, Y. Fainman, G. Papen, and A. Vahdat. Helios: A Hybrid Electrical/Optical Switch Architecture for Modular Data Centers. SIGCOMM 2010, New Delhi, India, 2010. ACM.

[54] D. Firestone. VFP: A Virtual Switch Platform for Host SDN in the Public Cloud. NSDI 2017, Boston, MA, USA, 2017. USENIX Association.

[55] G. Forman, M. Jain, M. Mansouri-Samani, J. Martinka, and A. C. Snoeren. Automated Whole-System Diagnosis of Distributed Services Using Model-Based Reasoning. IFIP/IEEE Workshop on Distributed Systems: Operations and Management 1998, Newark, DE, Oct. 1998.

[56] J. Gettys and K. Nichols. Bufferbloat: Dark Buffers in the Internet. *Queue*, 9(11):40:40–40:54, Nov. 2011.

[57] M. Ghasemi, T. Benson, and J. Rexford. Dapper: Data Plane Performance Diagnosis of TCP. SOSR 2017, Santa Clara, CA, USA, 2017. ACM.

[58] D. Ghita, K. Argyraki, and P. Thiran. Network Tomography on Correlated Links. IMC 2010, Melbourne, Australia, 2010. ACM.

[59] A. Greenberg. PingMesh + NetBouncer: Fine-grained path and link monitoring for data centers. https://atscaleconference.com/videos/pingmesh-netbouncer-fine-grained-path-and-link-monitoring-for-data-centers/, 2016.

[60] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. SIGCOMM 2009, Barcelona, Spain, 2009. ACM.

[61] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: Towards an Operating System for Networks. *SIGCOMM CCR*, 38(3), July 2008.

[62] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu. BCube: A High Performance, Server-centric Network Architecture for Modular Data Centers. SIGCOMM 2009, Barcelona, Spain, 2009. ACM.

[63] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu. Dcell: A Scalable and Fault-tolerant Network Structure for Data Centers. SIGCOMM 2008, Seattle, WA, USA, 2008. ACM.

[64] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, Z.-W. Lin, and V. Kurien. Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis. SIGCOMM 2015, London, United Kingdom, 2015. ACM.

[65] D. Halperin, S. Kandula, J. Padhye, P. Bahl, and D. Wetherall. Augmenting Data Center Networks with Multi-gigabit Wireless Links. SIGCOMM 2011, Toronto, Ontario, Canada, 2011. ACM.

[66] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown. I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks. NSDI 2014, Seattle, WA, USA, 2014. USENIX Association.

[67] K. He, E. Rozner, K. Agarwal, W. Felter, J. Carter, and A. Akella. Presto: Edge-based Load Balancing for Fast Datacenter Networks. SIGCOMM 2015, London, United Kingdom, 2015. ACM.

[68] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving High Utilization with Software-driven WAN. SIGCOMM 2013, Hong Kong, China, 2013. ACM.

[69] IBM. What is HDFS? Apache Hadoop Distributed File System. https://www.ibm.com/analytics/hadoop/hdfs.

[70] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. EuroSys, Lisbon, Portugal, 2007. ACM.

[71] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a Globally-deployed Software Defined Wan. SIGCOMM 2013, Hong Kong, China, 2013. ACM.

[72] V. Jalaparti, P. Bodik, S. Kandula, I. Menache, M. Rybalkin, and C. Yan. Speeding Up Distributed Request-response Workflows. SIGCOMM 2013, Hong Kong, China, 2013. ACM.

[73] V. Jeyakumar, M. Alizadeh, Y. Geng, C. Kim, and D. Mazières. Millions of Little Minions: Using Packets for Low Latency Network Programming and Visibility. SIGCOMM 2014, Chicago, Illinois, USA, 2014. ACM.

[74] V. Jeyakumar, M. Alizadeh, C. Kim, and D. Mazières. Tiny Packet Programs for Low-latency Network Control and Monitoring. HotNets-XII, College Park, Maryland, USA, 2013. ACM.

[75] A. Kabbani, B. Vamanan, J. Hasan, and F. Duchene. FlowBender: Flow-level Adaptive Routing for Improved Latency and Throughput in Datacenter Networks. CoNEXT 2014, Sydney, Australia, 2014. ACM.

[76] S. Kandula, D. Katabi, and J.-P. Vasseur. Shrink: A Tool for Failure Diagnosis in IP Networks. MineNet 2005, Philadelphia, Pennsylvania, USA, 2005. ACM.

[77] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. The Nature of Data Center Traffic: Measurements & Analysis. IMC 2009, Chicago, Illinois, USA, 2009. ACM.

[78] R. Kapoor, A. C. Snoeren, G. M. Voelker, and G. Porter. Bullet Trains: A Study of NIC Burst Behavior at Microsecond Timescales. CoNEXT 2013, Santa Barbara, California, USA, 2013. ACM.

[79] D. Katabi, M. Handley, and C. Rohrs. Congestion Control for High Bandwidth-delay Product Networks. SIGCOMM 2002, Pittsburgh, Pennsylvania, USA, 2002. ACM.

[80] N. P. Katta, J. Rexford, and D. Walker. Incremental Consistent Updates. HotSDN 2013, Hong Kong, China, 2013. ACM.

[81] P. Kazemian, G. Varghese, and N. McKeown. Header Space Analysis: Static Checking for Networks. NSDI 2012, San Jose, CA, USA, 2012. USENIX Association.

[82] M. Khabbaz, K. Shaban, C. Assi, and L. Qu. Prioritizing Deadline-constrained Data Flows in Cloud Datacenter Networks. Symposium on Applied Computing 2016, Pisa, Italy, 2016. ACM.

[83] R. Kohavi and R. Longbotham. Online Experiments: Lessons Learned. http://exp-platform.com/Documents/IEEEComputer2007OnlineExperiments.pdf, 2007.

[84] R. R. Kompella, J. Yates, A. Greenberg, and A. C. Snoeren. IP Fault Localization via Risk Modeling. NSDI 2005, Boston, MA, USA, 2005. USENIX Association.

[85] R. R. Kompella, J. Yates, A. Greenberg, and A. C. Snoeren. Detection and Localization of Network Black Holes. INFOCOM, Anchorage, AK, USA, 2007. IEEE Computer Society.

[86] R. R. Kompella, J. Yates, A. Greenberg, and A. C. Snoeren. Fault Localization via Risk Modeling. *IEEE Trans. Dependable Secur. Comput.*, 7(4), Oct. 2010.

[87] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: A Distributed Control Platform for Large-scale Production Networks. OSDI 2010, Vancouver, BC, Canada, 2010. USENIX Association.

[88] A. Kuznetosv. ss - another utility to investigate sockets. https://linux.die.net/man/8/ss.

[89] M. Kuzniar, P. Peresini, and D. Kostić. Providing Reliable FIB Update Acknowledgments in SDN. CoNEXT 2014, Sydney, Australia, 2014. ACM.

[90] W. E. Leland, M. S. Taqqu, W. Willinger, and D. V. Wilson. On the Self-similar Nature of Ethernet Traffic (Extended Version). *IEEE/ACM Transactions on Networking*, 2(1):1–15, Feb. 1994.

[91] A. Likhtarov, R. Nishtala, R. McElroy, H. Fugal, A. Grynenko, and V. Venkataramani. Introducing mcrouter: A memcached protocol router for scaling memcached deployments. https://code.facebook.com/posts/296442737213493, Sept. 2014.

[92] H. Liu, F. Lu, A. Forencich, R. Kapoor, M. Tewari, G. M. Voelker, G. Papen, A. C. Snoeren, and G. Porter. Circuit Switching Under the Radar with REACToR. NSDI 2014, Seattle, WA, USA, 2014. USENIX Association.

[93] V. Liu, D. Halperin, A. Krishnamurthy, and T. Anderson. F10: A Fault-tolerant Engineered Network. NSDI 2013, Lombard, IL, USA, 2013. USENIX Association.

[94] K. Lougheed and Y. Rekhter. RFC 1105: A Border Gateway Protocol (BGP), 1989.

[95] L. Ma, T. He, A. Swami, D. Towsley, K. K. Leung, and J. Lowe. Node Failure Localization via Network Tomography. IMC 2014, Vancouver, BC, Canada, 2014. ACM.

[96] R. Mack. Building Timeline: Scaling up to hold your life story. https://www.facebook.com/note.php?note_id=10150468255628920, Jan. 2012.

[97] D. Maltz. Private communication, Feb. 2017.

[98] K. Mandakolathur. Network Disaggregation Does Your Switch have the Right Packet Buffer Architecture? http://www.mellanox.com/blog/2017/09/network-disaggregation-switch-right-packet-buffer-architecture-part-2/, 2017.

[99] V. Mann, A. Vishnoi, and S. Bidkar. Living on the edge: Monitoring network flows at the edge in cloud data centers. COMSNETS 2013, Bangalore, India, 2013. IEEE.

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

[101] P. Mockapetris. RFC 1035: DOMAIN NAMES - IMPLEMENTATION AND SPECIFICATION, 1987.

[102] M. Moshref, M. Yu, A. Sharma, and R. Govindan. Scalable Rule Management for Data Centers. NSDI 2013, Lombard, IL, USA, 2013. USENIX.

[103] J. Moy. RFC 2328: OSPF Version 2, 1998.

[104] R. N. Mysore, R. Mahajan, A. Vahdat, and G. Varghese. Gestalt: Fast, Unified Fault Localization for Networked Systems. USENIX ATC 2014, Philadelphia, PA, USA, 2014. USENIX Association.

[105] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, C. Kim, and N. Karri. Ananta: Cloud Scale Load Balancing. SIGCOMM 2013, Hong Kong, China, 2013. ACM.

[106] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, and S. Shenker. Extending Networking into the Virtualization Layer. ACM HotNets 2009, New York, NY, USA, Oct. 2009. ACM.

[107] L. Popa, S. Ratnasamy, G. Iannaccone, A. Krishnamurthy, and I. Stoica. A Cost Comparison of Datacenter Network Architectures. CoNEXT 2010, Philadelphia, Pennsylvania, USA, 2010. ACM.

[108] S. Radhakrishnan, M. Tewari, R. Kapoor, G. Porter, and A. Vahdat. Dahu: Commodity Switches for Direct Connect Data Center Networks. ANCS 2013, San Jose, California, USA, 2013. IEEE Press.

[109] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving Datacenter Performance and Robustness with Multipath TCP. SIGCOMM 2011, Toronto, Ontario, Canada, 2011. ACM.

[110] P. Reynolds, J. L. Wiener, J. C. Mogul, M. K. Aguilera, and A. Vahdat. WAP5: Black-box Performance Debugging for Wide-area Systems. WWW 2006, Edinburgh, Scotland, 2006. ACM.

[111] M. Roughan, T. Griffin, Z. M. Mao, A. Greenberg, and B. Freeman. IP Forwarding Anomalies and Improving Their Detection Using Multiple Data Sources. ACM SIGCOMM Workshop on Network Troubleshooting 2004, Portland, Oregon, USA, 2004. ACM.

[112] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the Social Network's (Datacenter) Network. SIGCOMM 2015, London, United Kingdom, 2015. ACM.

[113] A. Roy, H. Zeng, J. Bagga, and A. C. Snoeren. Passive Realtime Datacenter Fault Detection and Localization. NSDI 2017, Boston, MA, USA, 2017. USENIX Association.

[114] S. Savage, D. Wetherall, A. Karlin, and T. Anderson. Practical Network Support for IP Traceback. SIGCOMM, Stockholm, Sweden, 2000. ACM.

[115] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar. Can the Production Network Be the Testbed? OSDI 2010, Vancouver, BC, Canada, 2010. USENIX Association.

[116] A. Simpkins. Facebook Open Switching System (FBOSS) and Wedge in the open. https://code.facebook.com/posts/843620439027582/facebook-open-switching-system-fboss-and-wedge-in-the-open/, Mar. 2015.

[117] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, and A. Vahdat. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. SIGCOMM 2015, London, United Kingdom, 2015. ACM.

[118] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey. Jellyfish: Networking Data Centers Randomly. NSDI 2012, San Jose, CA, USA, 2012. USENIX Association.

[119] D. Sommermann and A. Frindell. Introducing Proxygen, Facebook's C++ HTTP framework. https://code.facebook.com/posts/1503205539947302, 2014.

[120] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: A Warehousing Solution over a Map-reduce Framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, Aug. 2009.

[121] D. Turner, K. Levchenko, J. C. Mogul, S. Savage, and A. C. Snoeren. On Failure in Managed Enterprise Networks. Technical report, Hewlett-Packard Labs, May 2012.

[122] D. Turner, K. Levchenko, A. C. Snoeren, and S. Savage. California Fault Lines: Understanding the Causes and Impact of Network Failures. SIGCOMM 2010, New Delhi, India, 2010. ACM.

[123] G. Wang, D. G. Andersen, M. Kaminsky, K. Papagiannaki, T. E. Ng, M. Kozuch, and M. Ryan. c-Through: Part-time Optics in Data Centers. SIGCOMM 2010, New Delhi, India, 2010. ACM.

[124] B. P. Welford. Note on a Method for Calculating Corrected Sums of Squares and Products. *Technometrics*, 4(3), 1962.

[125] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron. Better Never Than Late: Meeting Deadlines in Datacenter Networks. SIGCOMM 2011, Toronto, Ontario, Canada, 2011. ACM.

[126] X. Wu, D. Turner, C.-C. Chen, D. A. Maltz, X. Yang, L. Yuan, and M. Zhang. NetPilot: Automating Datacenter Network Failure Mitigation. SIGCOMM 2012, Helsinki, Finland, 2012. ACM.

[127] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. EuroSys 2010, Paris, France, 2010. ACM.

[128] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. NSDI 2012, San Jose, CA, USA, 2012. USENIX.

[129] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz. DeTail: Reducing the Flow Completion Time Tail in Datacenter Networks. SIGCOMM 2012, Helsinki, Finland, 2012. ACM.

[130] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. Automatic Test Packet Generation. CoNEXT 2012, Nice, France, 2012. ACM.

[131] Q. Zhang, V. Liu, H. Zeng, and A. Krishnamurthy. High-resolution Measurement of Data Center Microbursts. IMC 2017, London, United Kingdom, 2017. ACM.

[132] X. Zhou, Z. Zhang, Y. Zhu, Y. Li, S. Kumar, A. Vahdat, B. Y. Zhao, and H. Zheng. Mirror Mirror on the Ceiling: Flexible Wireless Links for Data Centers. SIGCOMM 2012, Helsinki, Finland, 2012. ACM.

[133] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, and H. Zheng. Packet-Level Telemetry in Large Datacenter Networks. SIGCOMM 2015, London, United Kingdom, 2015. ACM.