Understanding the Limits of Passive Realtime Datacenter Fault Detection and Localization

Arjun Roy, Rajdeep Das, Hongyi Zeng, Jasmeet Bagga, and Alex C. Snoeren, *Senior Member, IEEE*

***Abstract*— Datacenters are characterized by large scale, stringent reliability requirements, and significant application diversity. However, the realities of employing hardware with non-zero failure rates mean that datacenters are subject to significant numbers of failures that can impact performance. Moreover, failures are not always obvious; network components can fail partially, dropping or delaying only subsets of packets. Thus, traditional fault detection techniques involving end-host or router-based statistics can fall short in their ability to identify these errors. We describe how to expedite the process of detecting and localizing partial datacenter faults using an end-host method generalizable to most datacenter applications. In particular, we correlate end-host transport-layer flow metrics with per-flow network paths and apply statistical analysis techniques to identify outliers and localize faulty links and/or switches. We evaluate our approach in a production Facebook front-end datacenter, focusing on its effectiveness across a range of traffic patterns.**

***Index Terms*— Computer network reliability, fault diagnosis.**

I. Introduction

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 [40] and beyond—so too do the workloads they support: for example, Google datacenters support literally thousands of distinct applications and services [42]. 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 efficient operation, these efforts can be frustrated by the inevitable failures that arise within the network itself.

Unfortunately, experience indicates that modern datacenters are rife with hardware and software failures—indeed, they are designed to be robust to large numbers of faults. Large scale deployment both ensures non-trivial fault rates and complicates localization. Microsoft authors describe [48] a rogue's gallery of datacenter faults: dusty fiber-optic connectors causing corrupted packets, switch software and hardware faults, incorrect load balancing, untrustworthy counters, and more. Unfortunately, faults can be intermittent and partial: rather than failing completely, a link or switch might only affect some traffic, complicating detection and diagnosis. Moreover, failures can have significant application impact. For example, NetPilot [46] describes how a single link dropping a small percentage of packets, combined with cut-through routing, resulted in degraded application performance and a multiple-hour diagnosis process to identify the faulty device.

Existing production methods for detecting and localizing datacenter network faults typically involve watching for anomalous network events (for example, scanning switch-queue drop and link utility/error counters) and/or monitoring performance metrics at end hosts. Such methods consider each event independently: "Did a drop happen at this link? Is application remote procedure call (RPC) latency unusually high at this host?" Yet, in isolation, knowledge of these events is of limited utility. There are many reasons an end host could observe poor network performance; similarly, in-network packet drops may be the result of transient congestion rather than a persistent network fault. Hence, datacenter operators frequently fall back to active probing and a certain degree of manual analysis to diagnose and localize detected performance anomalies [24], [48].

We propose an alternative approach: rather than looking at anomalies independently, we consider the impacts of faults on aggregate application performance. Modern datacenter fabrics are designed with a plethora of disjoint paths and operators work hard to load balance traffic across both paths and servers [9], [42]. Such designs result in highly regular flow performance regardless of path—in the absence of network faults [40]. 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.

To facilitate such comparisons, we develop a light-weight packet-marking technique—using features supported by commodity switch application-specific integrated circuits (ASICs)—that uniquely identify per-flow paths within Facebook datacenters. Moreover, Facebook's topological regularity allows us to use path information to also passively localize faults. Because end hosts can categorize flows according to the individual network elements traversed, we can contrast flows traversing any given link (switch) at a particular level in the hierarchy with flows that use alternatives to identify the likely source of detected performance anomalies. Operators can then use our system's output—namely the set of impacted traffic

Manuscript received July 13, 2018; revised April 7, 2019; accepted July 31, 2019; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor R. La. Date of publication September 13, 2019; date of current version October 15, 2019. This work was supported by the National Science Foundation under Grant CNS-1422240 and Grant CNS-1564185. A preliminary version of this article was presented at the USENIX Symposium on Networked Systems Design and Implementation (NSDI), Boston, MA, USA, March 2017. *(Corresponding author: Arjun Roy.)*

A. Roy, R. Das, and A. C. Snoeren are with the Department of Computer Science and Engineering, University of California at San Diego, La Jolla, CA 92093 USA (e-mail: arroy@cs.ucsd.edu; r4das@cs.ucsd.edu; snoeren@cs.ucsd.edu).

H. Zeng and J. Bagga are with Facebook Inc., Menlo Park, CA 94025 USA (e-mail: zeng@fb.com; jasmeetbagga@fb.com).

Digital Object Identifier 10.1109/TNET.2019.2938228

and the network element(s) seemingly responsible—to adjust path selection (e.g., through updating OpenFlow rules, altering equal-cost multipath routing (ECMP) weights [37], or tweaking flow hash inputs [29]) to mitigate performance impact until they can repair the underlying fault(s).

A naive implementation of our approach is unlikely to succeed given the noisiness of flow-based metrics at individual-host scale. Furthermore, distinct applications, or different workloads for the same application, are likely to be impacted differently by any given fault. Here the massive scale of modern datacenters aids us: Specifically, when aggregated across the full set of traffic traversing any given link, we find that statistical techniques are effective at using end-host metrics to identify under-performing links in real time. Our experience suggests this can remain true even when end hosts service different requests, communicate with disjoint sets of remote servers, or run entirely distinct application services.

To be practical, our approach must not place a heavy burden on the computational resources of either end hosts or network switches, nor can we require significant network bandwidth or esoteric features of the switches themselves. Furthermore, our analysis techniques must avoid false positives despite the diversity and scale of production datacenters, yet remain sensitive to the myriad possible impacts of real-word faults. Our contributions include (1) a general-purpose, end-host-based monitoring scheme that can robustly identify flows traversing faulty network components, (2) a methodology to discover the necessary path information scalably in Facebook's datacenters, (3) a system that leverages both types of information in aggregate to perform network-wide fault-localization, (4) a study of the system's effectiveness when applied to various production Facebook services with differing traffic characteristics, and (5) a discussion of the system's limitations, both in terms of scenarios where it does not apply and capabilities it does not possess.

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 allows us to leverage simple statistical methods to rapidly pinpoint partial faults as they happen. We show that our technique can identify links and switches exhibiting low levels (0.25–1.0%) of packet loss within datacenters hosting user-servicing front-end web and caching servers within 20 seconds of fault occurrence with a minimal amount of processing overhead. While Facebook front-end traffic exhibits significant temporal stability [40] that enables particularly rapid partial-fault detection, we further demonstrate the applicability of our methodology for more temporally unruly bulk-data workloads including Hadoop and Facebook WarmStorage [21]. In particular, we find that periods of just one minute are sufficient to reliably pinpoint low-level packet loss to individual links.

II. Motivation & Related Work

While fault detection is a classical problem in distributed systems [7], [12], [16], [17], [22], [38] and networks [13], [30], [35], modern datacenters provide both significant challenges (e.g., traffic volume, path diversity, and stringent latency requirements) and benefits (large degree of control, regular topologies) that impact the task of effectively finding and responding to network faults. Moreover, recent work has indicated that the types and impacts of faults common in modern datacenters [46], [48] differ from those typically encountered in the wide-area [39], [44] and enterprise [43].

Datacenters are affected by a menagerie of errors, including a mix of software errors (ECMP imbalances, protocol bugs, etc.), hardware errors (packet corruption due to cables or switches, unreliable packet counters, bit errors within routing tables, etc.), configuration errors, and a significant number of errors without an apparent cause [48]. Errors can be classified into two main categories: a complete error, in which an entire link or switch is unable to forward packets, or a partial error, where only a subset of traffic is affected. In this work, we focus primarily on the latter. Even a partial error affecting and corrupting only 1% of packets on two links inside a datacenter network was noted to have a $4.5\times$ increase in the 99th-percentile application latency [46], which can impact revenue generated by the hosted services [27].

Commonly deployed monitoring approaches include end-host monitoring (RPC latency, TCP retransmits, etc.) and switch-based monitoring (drop counters, queue occupancies, etc.). However, such methods can fall short for troubleshooting datacenter networks. Host monitoring alone lacks specificity in the presence of large scale multipath; applications suffering from dropped packets or increased latency do not have any intuition on where the fault is located, or whether a given set of performance anomalies are due to the same fault.

Similarly, if a switch drops a packet, the operator is unlikely to know which applications were impacted, or, more importantly, what is to blame. Even if switches sample dropped packets, operators may not have a clear idea of impacted traffic. Due to sampling bias, mouse flows experiencing loss may be missed, despite incurring great performance impacts. Switch-counter-based approaches are further confounded by cut-through forwarding and unreliable hardware [48].

Thus, we propose a system that focuses not on the occurrence of network anomalies but rather on their traffic impact. We leverage the high path diversity and regular topologies found in modern datacenters, as well as the finely tuned load balancing common in such environments. We also expose to each host, for every flow, the path taken through the network. Our approach has several key differences with both past academic proposals and production systems:

1) **Full path information**: Past fault-finding systems have associated performance degradations with components and logical paths [7], [11], [16], [24] 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 [48].
2) **Passive monitoring, not active probing**: Unlike active probing methods [20], [24], our method leverages readily available metrics from production traffic, simultaneously decreasing network overhead, increasing the detection surface, and decreasing detection and localization time.

3) **Reduced switch dependencies**: Our approach does not require expanded switch ASIC features, unlike some others [26]—allowing resilience to bugs that may escape on-switch monitoring. Network operators may deploy our approach on commodity switches.
4) **No per-application modeling:** Our system leverages the stable and load-balanced traffic patterns found in some modern datacenters [40]. Thus, it does not need to model complicated application dependencies [7], [12], [16] or interpose on application middleware [17]. Since we compare relative performance across network links, we do not require explicit performance thresholds.
5) **Rapid online analysis**: The regularity inherent in several academic [8] and production [9], [42] topologies allow us to rapidly (10–20 seconds) detect partial faults of very small magnitude (under 0.5% packet loss) using an online approach. This result contrasts with prior systems that require offline analysis [7] or much larger timescales to find faults [12]. Furthermore, localizing faults is possible without resource-intensive and potentially time-consuming graph analysis [20], [30]–[33], [36].

A recently proposed alternative, 007 [10], correlates losses to network links (like us), before (unlike us) using a voting scheme to pinpoint faults. Like us, 007 uses passive monitoring, has no switch or application dependencies, and can perform rapid online analysis. Unlike us, 007 recovers paths via traceroute due to complicating factors inside Microsoft datacenters. Furthermore, 007 does not monitor latency; while it may be extended to threshold on high latency, doing so would require additional validation [10]. However, both approaches are complementary and could run side-by-side if desired.

III. System Overview

Here, 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 key components of our system, deferring implementation details to Section IV.

A. Production Datacenter

Facebook's datacenters consist of thousands of hosts and hundreds of switches grouped into a multi-rooted, multi-level tree topology [9]. We consider front-end datacenters serving end-user web requests, comprised primarily of web servers and cache servers [40]. While wide-area connections to other installations exist, we do not focus on those in this work.

Topology: Web and cache hosts are grouped by type into racks, each housing a few tens of hosts. 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 connections between web servers and caches; in particular, each web server has thousands of bidirectional flows spread evenly amongst caches [40]. 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 [47].

A few tens of racks comprises a pod. Each pod also contains four aggregation switches (Aggs); each top-of-rack switch (ToR) has one uplink to each Agg. There are a few tens of pods within the datacenter, with cross-pod communication enabled by four disjoint core-switch planes (each a few tens of cores). Each Agg is connected to cores in exactly one plane in a mesh.

Each host transmits many flows with differing paths. Due to ECMP routing, mesh-like traffic patterns, and extensive load balancing, links at the same hierarchical topology level have close to identical flow (performance metric) distributions. Moreover, if we know per-flow paths (i.e., sets of links), it is straightforward to bin flows based on links traversed at any particular topology level. Hence, we can simultaneously perform fault identification and localization by considering performance metrics across different subsets of flows.

Operational Constraints: The sheer scale of the datacenter environment 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 the smaller errors. In addition, we must detect both packet loss and delay.

Datacenter application and workload variety 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 support measurements from large numbers of end hosts describing the health of large numbers of links, without imposing large computational or data overheads either on the hosts or the network. This is especially true of switches, where relatively under-provisioned control planes are already engaged in critical tasks including Border Gateway Protocol (BGP) routing. Thus, we limit ourselves to capabilities present in commodity switch data planes.

B. System Architecture

Our fault detection and localization approach involves functional components at all end hosts, a subset of switches, and a centralized controller, as depicted in Figure 1. Switches mark packets to indicate network path (1). Hosts then independently compare the performance of their own flows to generate a host-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 expand on the role of each element below.

End Hosts: Hosts run production application traffic and track various per-flow metrics detailed in Section IV-B. In addition, hosts are aware of each flow's network path. Periodically, each host will use collected performance data to

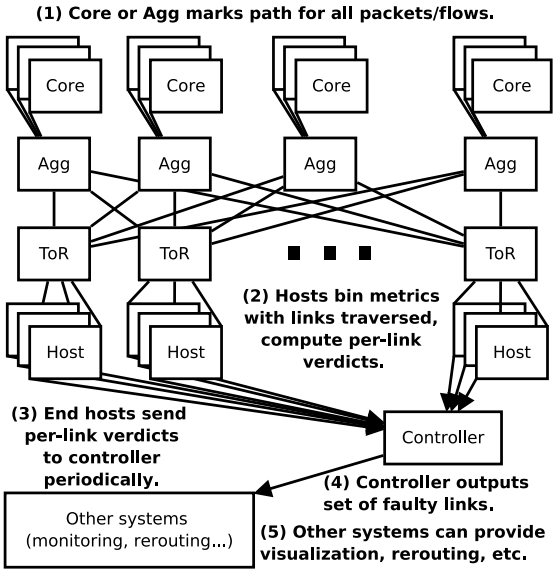


Fig. 1. High-level system overview (single pod depicted).

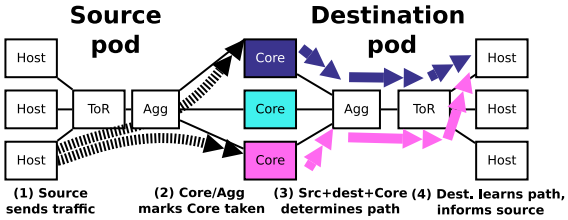


Fig. 2. Determining flow network path.

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 ES. Alternative schemes can give us further resolution into faults; e.g. comparing traffic by queue or subnet [41]. In certain cases, however, ESes may be unavailable; we discuss this situation further in Section VII.

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. Not all end hosts need participate in this process, however; there is a trade-off between participation rate and detection speed that we examine in Section V.

Switches: A subset of switches are responsible for signaling per-flow network paths to end hosts; we describe details in Section IV-A. 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 host-generated verdicts for link health (e.g. hosts 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 end-host 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 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.

IV. Implementation

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

A. Datacenter Flow Path Discovery

Central to our approach is the ability to scalably discover flow path information within datacenters. While switch CPU/dataplane limits complicate this task, the regularity of Facebook's topology aids us. General-case topologies can support flow-level pathfinding (for longer-lived flows only) using more restrictive but widely-supported marking techniques [41].

Topological Constraints: Figure 2 depicts our pathfinding scheme at Facebook. To aid discussion, the diagram shows a restricted subset of an unfolded version of the topology, with source hosts on the left and destinations on the right; the topology is symmetric so our approach works in either direction. 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 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.

Source and destination ToRs are fixed for any particular host pair, and can be determined by examining flow IP addresses. For cross-pod traffic, the core switch uniquely determines flow path, as Agg switches are then constrained on both sides. For intra-pod traffic, the Agg switch determines flow path. Complete link failures may result in the use of alternative, non-shortest backup routes; while not addressed in our prototype, they may be handled by examining routing state [41].

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

TABLE I

METRIC STATISTICS FOR (C)ACHE/(W)EB HOSTS GROUPED BY METRIC. EACH COLOR-BANDED PAIR OF ROWS DENOTES THE BASE AND IMPACTED METRICS FOR (AGGREGATED) WORKING AND (UNIQUE) FAULTY PATHS. `ssthresh` IS OMMITTED FOR BREVITY

#	Host	Metric	Error	p25	p50	p75	p90	p95	p99
1	(C)	cwnd	0.5%	7 (-30%)	8 (-20%)	10 (par)	10 (par)	10 (-41%)	20 (-33%)
2	(C)	cwnd	-	10	10	10	10	17	30
3	(C)	retx	0.5%	0 (par)	1	2	3	4	6
4	(C)	retx	-	0	0	0	0	0	1
5	(W)	cwnd	0.5%	8 (-63.6%)	12 (-60%)	17 (-74.6%)	38 (-60%)	121 (+23.5%)	139 (-33.2%)
6	(W)	cwnd	-	22	30	67	95	98	208
7	(W)	retx	0.5%	0	0	1	3	4	6
8	(W)	retx	-	0	0	0	0	0	0

and are equally capable of marking the packet. We use Linux eBPF (Extended Berkeley Packet Filter) [3] along with `bcc` (BPF Compiler Collection) [2] instrumentation at end hosts to read packet markings and derive flow paths. Our naive implementation imposes less than 1% CPU overhead (top row of Table II), but room for optimization remains.

Several header fields may be used for marking, with the best choice likely depending on the details of any given deployment. One possibility is the 20-bit IPv6 flow label which can scale to networks with over a million core switches. However, our Agg switches do not currently support modifying this field. For our prototype, we instead mark the IPv6 differentiated services code point (DSCP) field using one rule per uplink. While DSCP suffices for a prototype, its length limits the number of discernible paths. Furthermore, datacenter operators may use DSCP to influence queuing. An alternative is to mark TTL; southbound packets marked at Aggs traverse two more hops before reaching their destination, and so hosts can recover IDs in the TTL field if the value is between [3–255]. Note that `traceroute` or eBGP session protection [18] may further constrain TTL usage in some deployments.

B. Aggregating Host Metrics & Path Data

Given per-flow paths, transport-layer metrics can find under-performing links for both latency-sensitive, client/server applications [40] (discussed here) and bulk-flow, large-scale computation applications [4], [19], [28] (Section VI).

Latency-Sensitive Services: Facebook Web servers and caches are latency sensitive [47], and faults or congestion can increase latency. To infer congestion, TCP tracks loss (retransmission counts, congestion window (`cwnd`) and slow-start threshold (`sstresh`)) and latency (smoothed round trip time (`srtt`)) metrics. In isolation, these metrics are of limited usefulness for fault diagnosis. While retransmits signify loss, they provide limited predictive (why was a packet dropped?) and associative (are a set of flows lossy because of the same underlying fault?) value. Furthermore, while these metrics are sensitive to loss and latency, normal values depend on application pattern. Thus, comparing a given flow against an average can be difficult, since it is unclear whether 'below average' performance is due to a fault or traffic characteristics.

But, when augmented with path data, these metrics can identify faults. To illustrate, we induce faults impacting one web server and cache. Each host has a ToR connected to four Aggs. Using `iptables` rules, each host drops 0.5% of incoming packets transiting the Agg 1-ToR link. We measure outgoing flow TCP metrics, grouped by the Agg (and, thus, ToR downlink) transited by the corresponding inbound flow.

TABLE II

END-HOST 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%

Table I depicts metric distributions grouped by inbound downlink for production cache (C) and web servers (W). While we aggregate non-faulty links into one series (even-numbered rows), each individual link closely follows the aggregate. The faulty-link distributions, however, are significantly skewed—towards smaller numbers for `cwnd`, and larger for retransmits.

While these metrics are from servers with identical TCP stacks, multi-stack scenarios (e.g. diverse customer VMs in cloud networks) are also supported with enough traffic volume. If a host operates multiple TCP stacks, then our methodology can be applied independently to the statistics from each stack. If an end host communicates with N_S remote hosts for a given TCP stack S, an ES consists of L links, and if $N_S \gg L$ for all S, then per-link performance will be equivalent even though different stacks may respond differently to loss and delay since these effects are spread evenly across the ES.

Metric Collection Computational Overhead: While the aforementioned statistics provide a useful signal, care must be taken when determining how to collect statistics. We directly read `netlink` sockets to collect TCP statistics in a manner similar to the `ss` command. Table II depicts the overall CPU usage of our TCP statistics collection and verdict generation agent; the cumulative CPU overhead is below 1%.

C. Verdict Generation

We hypothesize that observed metrics are samples from a common underlying distribution characterizing fault-free links. Moreover, a substantially different distribution applies for faulty links for the same period. Thus, deciding link faultiness reduces to deciding if its sample distribution is of the same distribution as that of the fault-free links.

Problem Formulation: While we cannot definitively state the fault-free distribution parameters due to the complexities of network interactions, we conjecture that the instantaneous faulty link count is much smaller than the working link count.

Consider per-flow retransmits per link, for all links in a time period. We compare each link's distribution to the aggregate distribution for all other links. If there exists only one faulty link in the network, there are two possible cases for any link: the corresponding distribution 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 test distribution is skewed to significantly higher values compared 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. Multiple concurrent link errors can shift the aggregate distribution closer to the single faulty link distribution. If every link is faulty, we cannot detect faulty links since we do not find any outliers. However, our method is robust even if 75% of the links have faults; we examine the sensitivity of our approach in Section V-C.

Hypothesis Testing: To compare ES distributions, we leverage well-known statistical tests to perform hypothesis testing. Suppose we are determining if per-flow retransmits for *TESTLINK* exceeds all other links in the ES, given distributions corresponding to each link, measured at a single server within some period. Each empirical distribution corresponds to one link and comprises per-flow retransmit counts for that link and time period. We call the distribution under test L. We create an aggregate distribution L' that contains values from every link $O \neq$ *TESTLINK*. Comparing L to L', we set 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 outputs a p-value $\in [0, 1]$. If $p \leq \alpha$, 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 ES, 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 from this server's viewpoint.

The specific test depends on the metric. While the underlying distributions may not be known, we sample many data points in every distribution, for 10s of 1000s of active flows. Since we consider all flows in each period, we can compute the standard deviation of the full population. Additionally, due to ES formulation, we assume the no-error per-link distributions are nearly identical, and each (per-flow) sample is independent of the others (i.e. cross-flow performance is uncorrelated). By the central limit theorem, we assume the average across all flows in a distribution is approximately normally distributed.

Given these conditions, for TCP retransmit and TCP-over-IPv6 flow-label relabeling (discussed in Section VI) 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 than the other links in aggregate. In every time period, each server checks if the t-statistic is greater than 0 and $p \leq 0.05$. If so, we reject the null hypothesis, considering the link to be faulty.

Due to large traffic volume, even small intervals contain many samples. Thus, we must carefully implement our outlier test to avoid costly computational overhead. We compute the per-link t-statistic over consecutive fixed-length (10 seconds is the default) periods. We 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 [45]. 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.

The bottom row of Table II depicts the CPU utilization at a single production Web server using 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 kernel structures that track flow data.

For `cwnd`, `ssthresh` and `srtt`, the t-test is invalid; in particular, `cwnd` and `ssthresh` samples for a given flow are not independent of each other as they evolve over time. Furthermore, analyzing TCP state-machine metrics is difficult due to the specifics of different TCP versions, application demands, and traffic pattern characteristics. Since we do not make any assumptions on empirically-measured TCP state-machine distributions, we use non-parametric tests instead. Fundamentally, we seek to find the 'distance' between two distributions and determine if it is 'large', which is the precise 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 test link, and a similarly defined distribution for all other links in aggregate. We downsample because the large number of points in the original distributions makes the test too specific—downsampling lets us compare distributions that have roughly the original shape (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.

D. Centralized Fault Localization

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

Controller Processing: For a reliable but noisy end-host-level signal, we hypothesize that false positives are evenly distributed among links in the absence of faults. While some networks may contain hot spots that skew metrics and violate this assumption, Facebook traffic is evenly distributed on the timescales considered, 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 hosts, 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-square test with the null hypothesis that, in the absence of faults, all links will have relatively similar numbers of hosts that flag it not-guilty. The chi-square test outputs a p-value; if $p \leq 0.05$, 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 tens of links in less than one second on a Xeon E5-2660, and scales linearly in the number of links. Each host generates two verdicts per link (inbound/outbound) every 10 seconds, with O(1000s) hosts per pod. Each verdict consists of two 64-bit doubles (t-stat and p-value) and a link ID. In total, this yields a streaming overhead of under 10 Mbps per pod, well within the capabilities of the pub-sub framework.

V. Evaluation

We evaluate our approach within a production Facebook datacenter. First, we 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.

For our experiments, we instrumented 86 web servers across 3 ToRs with our methodology. Path markings are provided by a single Agg, which sets packet DSCP bits based on the transited core switch. (Hence, all experiments consider only the traffic transiting the instrumented Agg, and ignore the remainder.) To inject faults, we use `iptables` rules installed at end hosts to selectively drop inbound packets that traversed specific links (according to DSCP markings). For example, we can configure an end host to drop 0.5% of all inbound packets that transited a particular core-to-Agg link. This has the effect of injecting faults at an arbitrary network location, yet impacting only the systems that we monitor. While all monitored hosts will see the same link loss *rate*, the actual packets dropped may vary because `iptables` functions independently at each host.

A. Motivating Example

In a 5-minute interval, we induce faults on links from 3 core switches to the instrumented Agg, denoted A, B and C.

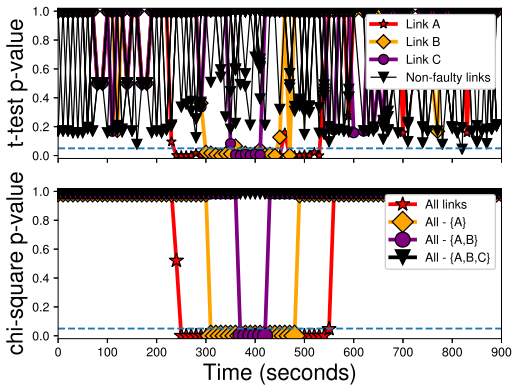


Fig. 3. Single-host t-test output (top) and controller chi-square output (bottom) for three separate link faults.

In particular, we induce faults in the order A, B, C, with a 1-minute gap; we then remove the faults in reverse order with the same gap. Each fault is a random 0.5% packet drop, an overall host-observed loss rate of under 0.02%.

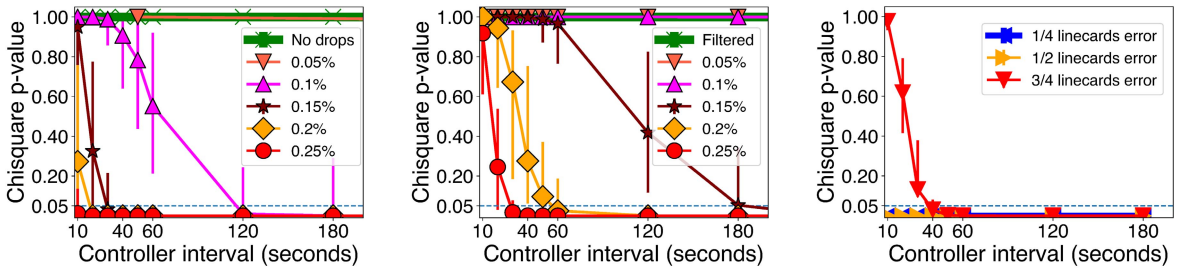
The top portion of Figure 3 depicts the t-test output for a single host. Flows are grouped according to incoming core downlink, and TCP retransmission statistics are aggregated into ten-second intervals. For a single host, for every non-faulty link (every series in black, and the faulty links before/after 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 hosts flag the faulty links as guilty, with few false positives.

Guilty verdicts are sent to the controller, which runs the chi-square test every 10 seconds using each core downlink as a category. It counts the number of *non-guilty* verdicts from end hosts as a metric and flags an error condition if $p \leq 0.05$. 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 3 depicts the controller output using this mechanism. We depict the output of the test for all paths (in red), and 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 flag the presence of a fault as long as there is at least one faulty link in the set of paths under consideration, thus supporting an iterative approach.

B. Speed and Sensitivity

For our mechanism to be useful, it must be able to rapidly detect faults, even if the impact is slight. Moreover, we need to detect latency-inducing faults, not just loss-inducing faults.

Loss Rate Sensitivity: Analysis reveals that while loss rates 0.5% and above are caught by at least 90% of hosts, we see a linear decrease in sensitivity as loss decreases—at 0.1% drop rate, only about 25% of hosts 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 4a depicts the distribution of p-values output by the controller for a given loss rate and calculation interval. Each point with error bars depicts



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

Fig. 4. Controller chi-square p-value convergence for various faults vs. controller interval length.

TABLE III

`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

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 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 (60+ seconds) 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: Within a private $k = 6$ fat-tree testbed [41], we induce delays for traffic traversing a particular core switch. To do this, we use Linux `tc-netem` on '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 run client/server traffic with a single pod of 9 HHVM [5] servers serving static pages, and two pods (18 hosts) configured as Web clients running Apache Benchmark [1]. Each server handles 180 simultaneous clients and serves either small (100-B) or medium (8-KB) requests.

Table III depicts the distributions for TCP `srtt_us` TCP at a particular end host as a function of induced latency and request size. As in previous experiments, the distributions of non-impacted links are very close to each other, while the distribution of the latency heavy link is clearly differentiable. No drops are detected in the 100-B case; due to our use of 1-Gbps links, a handful of drops comparable to the no-fault case are 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.

Detection Speed and End Host Participation: Not all end hosts need participate in our scheme (e.g. we may not control some hosts in a cloud setting). In this case, there is a trade-off between participation and detection time—but not sensitivity. Figure 6 depicts the controller interval required to detect a 0.25% packet loss on a link depending on what percentage of our 86 servers are participating in our detection methodology. Even at 25% participation, we can find a 0.25% loss within a minute. Smaller losses or participation rates will require aggregating for longer periods of time.

C. Precision and Accuracy

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

Concurrent Unequal Faults: Large networks will likely suffer concurrent faults of unequal magnitude, where the largest may mask 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 is flagged by almost all the hosts 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 4b depicts chi-square p-value outputs for the paths *excluding* the path hosting 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 4a), while a 0.15% rate requires at least 3 minutes.

Large Correlated Faults: So far, we have considered faults impacting small numbers of uncorrelated links. However, a single hardware fault can affect multiple links. For example, each Agg switch contains several core-facing linecards providing cross-pod capacity. A linecard-level fault would thus affect several uplinks. Similarly, a ToR-to-Agg switch link

might be impacted, affecting a full quarter of the uplinks for that rack's hosts. Our approach relies on the student's t-test picking outliers from a given average, with the assumption that the average represents a non-broken link. However, certain faults might impact a vast swath of links, driving the average performance closer to that of the impacted 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 is 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 when $p \leq 0.1$. Figure 4c shows the controller performance for various linecard-level faults as a function of interval length. A 10-second interval captures the cases where 1/4 and 1/2 of the uplinks experience correlated issues, but 40 seconds is required in the 3/4 case.

False Positives: Longer controller intervals provide greater loss sensitivity but increase false-positive risk since the statistical tests detect more minute per-distribution differences as input data volume increases. We measured false-positive incidence in production for multiple weeks in the absence of any (known) faults. While a (small) number of per-server false positives were generated in every interval, the controller successfully filtered out these indications since the noise was spread evenly across the monitored links. However, we noticed that the number of false positives had periodic local maxima around 1200 and 1700 GMT. These maxima were correlated with raw (i.e., independent of flow/path) TCP retransmit counts tracked by end hosts. Given that they occurred at similar times each day and were evenly spread across all the monitored hosts, we surmise that these retransmits were not due to network faults, but organically occurring congestion. This experience provides confidence that we can effectively distinguish between transient congestion and partial faults.

D. Small-Scale Deployment Experience

While our experiments focus on injected failures, we must determine whether we can successfully pinpoint network anomalies "in the wild". Thus, we ran our system over a relatively long time period (with no 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?" Thus, we deployed our system on 30 hosts for a two-week period in early 2017. As an experimental system, 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, partial failures occur at a rate of around 10/day in a network containing O(1M) hosts [34]. It is not surprising, then, that our two-week trial on only 30 hosts did not uncover any partial faults. We were, however, able to derive useful operational experience that we relate below.

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.

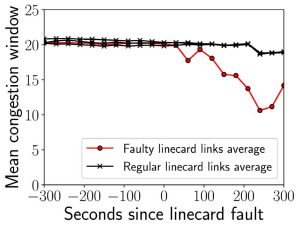


Fig. 5. Mean `cwnd` during linecard fault.

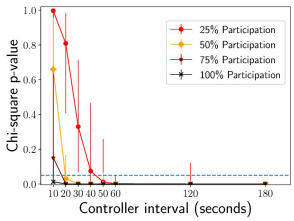


Fig. 6. Controller interval needed to detect 0.25% loss as a function of end host participation.

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 depicts the per-link mean congestion window measured by every host, aggregated per-linecard and averaged across every corresponding host, with the afflicted linecard colored red.

The deviations are immediate and significant, with mean `cwnd` dropping over 10% in the first interval after most traffic routes away, and continually diverging from the working links thereafter. Furthermore, flow volume measured at each host traversing the afflicted linecard rapidly drops from O(1000s) to O(10s) per link. By contrast, one of Facebook's monitoring systems, NetNORAD [6], took several minutes to detect the unresponsive linecard control plane 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.

VI. High-Variability Traffic

Temporally stable and highly load-balanced traffic [40] enables pinpointing loss rates around 0.1% within tens

of seconds. However, not all services possess such favorable characteristics. Traffic can be spatially imbalanced, or possess temporal hot spots for tens of seconds, possibly 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 [21] (henceforth, bulk-traffic) servers. Hadoop [4] 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 [21] is a storage disaggregation system targeted at fixing various shortcomings present in HDFS (the Hadoop Filesystem) when used at Facebook's scale.

A. Fundamental Effect: Temporal Behavior

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. Consider a workload with high temporal variance, like Hadoop at Facebook [40]. In a given 5-millisecond period, Hadoop servers average 25 concurrent active flows (defined by 5-tuple), though within the median-case millisecond, just 2 flows account for over half the sent traffic volume in bytes. Additionally, Hadoop server flow lifespans range from around 10 (near median case) to 100 (95^{th} percentile) seconds. Furthermore, only a subset of these flows leaving the server rack.

Accordingly, if we consider inter-rack Hadoop traffic in a 10-second interval, traffic is unlikely to be evenly spread among the ToR uplinks. A similar argument applies to incoming traffic at a Hadoop server. This effect is exacerbated at higher topology aggregation levels. Because retransmits are more likely on links carrying more traffic, the set of ToR uplinks (or Agg uplinks) is not an ES at this timescale.

Despite these per-second variations, Hadoop traffic eventually evens out— Hadoop clusters exhibit no inter-rack hot spots over a 24-hour period. Thus, outlier analysis may work if we consider longer intervals than for front-end servers. To minimize the length of these intervals, we may aggregate metrics and compute the t-test on a per-rack basis rather than per-server.

In Section VI-C we see that performing bulk-traffic per-rack t-tests over minute-long intervals (c.f. front-end per-server t-tests every 10 seconds) allows outlier analysis to find partial faults. However, simply increasing data collection intervals is insufficient. Due to limitations in our path identification method, we have to revise our choice of test metric.

B. Implementation Effect: Path and Metrics

Our initial private-testbed experiments [41] 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 outbound path links. This effect allowed us to use TCP metrics (retransmits, `srtt`, etc.) in our testbed for both synthetic front-end style traffic and Hadoop. A subtlety applies within our Facebook testing, where we measure outbound metrics but correlate it with *inbound* path information to pinpoint *inbound* per-link packet loss. Here, we explain why this correlation is suitable for front-end but *not* bulk-traffic servers. We then discuss an alternative that supports bulk-traffic servers.

At Facebook, our path-recovery methodology exposes only inbound flow path to hosts. Server TCP metrics track outbound flow performance. Despite this, we can correlate outbound TCP flow metrics with the path taken by their corresponding inbound flows. These correlations then reveal inbound link-by-link packet loss in Section V.

At first glance, this is a surprising result. Since outbound and inbound flows likely traverse different paths, it is odd that outbound flow performance corresponds with inbound flow path. We speculate that this is due to RPC traffic dynamics. Suppose a web server sends a request to a cache. Prior studies suggest typical 200-B single-packet requests [40] and 1–1000-B single-packet responses [15]. Before sending a response, a cache may send a bare ACK packet to the web server. Suppose, further, that a link in the returned ACK network path randomly loses packets, and that a returned ACK is lost. The web server will then consider *the original request packet* lost (since no ACK was received), and retransmit it. This *outbound* retransmit is thus caused by lossy behavior for a link on the *inbound* path for the reverse-direction inbound flow. Consequently, we can correlate outbound TCP metrics with inbound-direction links to find faults on the inbound path.

But, what happens for bulk-flow traffic? As before, inbound ACKs may be lost. Since TCP typically ACKs every other packet, however, a superceding ACK is often received before the retransmit timer expires, and no outbound retransmit occurs. Empirically, bulk-traffic outbound packet retransmits and `cwnd` no longer correlate to inbound packet loss, unlike front-end servers. Thus, we must find alternative metrics.

Inferring Packet Loss Through Flow-Label Churn: Since we cannot correlate bulk-traffic outbound TCP metrics with inbound flow performance, we have two options: either we recover outbound flow path (as was feasible in our testbed) or find a suitable inbound flow performance metric. To find outbound path information, we may set up signaling that allows destination servers to convey inbound flow path from their viewpoint to the sender. While feasible, this would require significant implementation effort and roll-out to provide enough data at Facebook's network scale.

Thus, we need an alternative metric that correlates with inbound packet loss. Since Facebook traffic is mostly TCP/IPv6 on Linux, we can leverage IPv6 header churn. Flowbender's [29] authors propose that servers may change packet header bits to influence routing within ECMP-enabled multipath networks—for example, to avoid potentially faulty paths when faced with degraded flow performance. Recent Linux kernels incorporate such a feature—when enough TCP/IPv6 retransmit timeouts occur, Linux modifies the 20-bit IPv6 flow-label to try for an alternative path.

Suppose a bulk-traffic server receives bulk flows from senders with such a kernel. If a flow suffers from enough loss, the sender will change the flow-label. The receiver can correlate flow-label changes with the inbound links transited at the time. If a particular link in an ES has a higher than

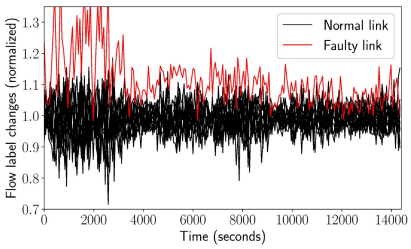


Fig. 7. Normalized flow-label churn per link for WarmStorage.

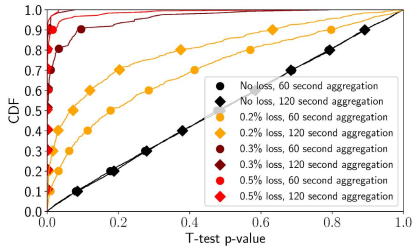


Fig. 8. WarmStorage rack t-test p-value distribution as a function of aggregation interval and loss.

normal amount of associated flow-label churn, it could be due to a partial fault centered at that link.

Flow-label churn is a sparser signal than TCP retransmits. While every loss triggers a retransmit, flow-label churn requires several retransmit timeouts, which are rarer than regular loss. While at least one retransmit occurs for any lost packet, timeouts only occur if the last packet in a send window is lost. Mid-window loss causes rapid duplicate ACKs, which prevent sender retransmit timeouts and thus changes in the flow-label.

To demonstrate this effect, we instrument several Hadoop and WarmStorage servers with a collection agent that correlates per-flow flow-label churn with links in an ES. We induce packet loss on one downlink from either a Core switch to an Agg (WarmStorage) or from an Agg to the ToR (Hadoop). To verify that flow-label churn responds to loss, we measure four hours of data and track the number of flow-label changes per link using 300-second intervals.

Figure 7 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, the lossy link has a higher number of flow-label changes throughout the four-hour period.

Since per-link flow-label churn indicates packet loss, we use it as t-test input for our system. Suppose we want to run the t-test every minute. We generate a per-link distribution for each minute-long period as follows:

1) We arbitrarily divide the 60-second period into 100-msec chunks. For each chunk/link, we sum flow-label churn for all flows, discarding chunks where no links have churn.
2) For the remaining chunks, at least one link had churn. For links where no churn occurred within the 100-msec period, the value is 0. These counts form per-link distributions for the 60-second period.
3) We apply the t-test for each link, comparing it to the aggregate of the other links. t-test execution be per-server or per-rack. Results are processed, as before, by a centralized controller running the chi-square test.

C. Bulk-Traffic Evaluation

Having described our bulk-traffic server partial-fault detection mechanism, we evaluate it in a production WarmStorage pod. We focus primarily on the continued applicability of the per-server or per-rack t-test. We focus on t-test precision (low false positives) and recall (low false negatives) when operating at different frequencies, and for different packet loss rates.

Experimental Setup: We instrument three WarmStorage server racks in a pod (60 servers in all). We install our packet-marking rules onto a single pod switch, revealing inbound traffic path. We install server `iptables` rules that simulate loss affecting one Core-to-Agg. downlink. Servers track IPv6 flow-label churn 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 a period of several hours. To characterize t-test precision and recall, we vary packet loss from 0–0.5%, and t-test frequency from 60–600 seconds. We compute per-server, per-rack and full-population t-tests to determine the benefits of aggregation.

T-Test Precision and Recall: Due to signal sparseness, we find that *per-server* per-minute t-test hit-rate (recall) is low. E.g. for 0.5% loss, the t-test yields $p \leq 0.05$ around 15% of the time (c. f. nearly 90% of the time for front-end servers with 10 second intervals). For 0.2% loss, the p-value distribution is indistinguishable from the no-error case.

However, we are still able to leverage our methodology—rather than treating each server individually, we can aggregate collected data and treat each server rack as a 'large' server. Figure 8 depicts, for various loss rates and test frequencies, the per-rack t-test p-value distributions over four hours. For example, if we consider per-minute t-tests, we receive three data points (one for each rack) per minute, and thus $3 \cdot 240 = 720$ data points over four hours. Typical pods consist of tens of racks and could collectively generate the same volume of data within minutes.

Each series in Figure 8 uses coloration 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 0.3% loss and per-minute t-tests, $p \leq 0.05$ around 80% of the time. Recall that for front-end servers with 0.25% loss, the t-test hit-rate was roughly 75%.[1] Across 90 servers, this hit rate provides enough confidence for a centralized controller running the chi-square test to correctly deduce the presence of a partial fault. Thus, given a 0.3% or higher loss rate, per-rack aggregation and a 60-second

[1] Albeit running only for 10 seconds instead of 60.

t-test interval, outlier-analysis-based partial fault localization appears to be applicable to WarmStorage workloads.

The 0.2% loss case has a significantly lower hit rate of about 20% with a 60-second test interval. Increasing the interval to 120, 300 and 600 seconds does boost hit rate to about 40%, 70% and 90% respectively, however. In the front-end case, these hit-rates allow the chi-square test to deduce the presence of faulty links, though a 40% hit-rate requires the controller to aggregate multiple rounds of t-test results for detection. Thus, even for a 0.2% loss rate, outlier analysis is applicable for partial-fault detection. Conversely, the signal misses 0.1% loss—at that point, it cannot rise above the noise caused by ambient flow-label churn present in non-faulty links.

That said, the behavior 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. Thus, instead of running the chi-square test on the per-link fault counts as we do for front-end servers (where a t-test 'hits' when $p \leq 0.05$ 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.

VII. Limitations

Two broad classes of limitations apply to our methodology—statistical inferences that we *cannot* draw, and scenarios where equivalence sets cannot be formed.

What Our Statistics do Not Allow: We may be tempted to use statistical techniques to more fully characterize or diagnose partial faults after localization, in the vein of prior work [11]. For example, we may wish to determine confidence intervals for partial-fault-imposed packet loss. However, our methodology does not support this goal for a few reasons. First, we make no claims about underlying distributions for the studied metrics. Second, we are not privy to all confounding effects and interactions for examined metrics. Third, we are passively monitoring metrics. While an active-probe injection approach may provide further insight into relevant partial-fault behaviors for a specific fault, we are limited to observable variations within aggregate traffic behavior.

Consequently, we cannot compute confidence intervals for the degree of packet loss on a link, or otherwise 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 V that we can rapidly pinpoint a link with 0.1% or 0.2% loss, 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 [23], [48] to perform this task.

When we Cannot Form Equivalence Sets: We may find scenarios where ESes cannot be formed. For example, total link failures may cause asymmetry, complicating flow path recovery and outlier analysis. Asymmetry may occur with unequal bandwidth link-aggregation groups (LAGs) or unequal queuing policies across links. If a candidate ES is asymmetric, we may consider disjoint, ES criteria-meeting subsets instead, as long as subsets contain at least three items so that outlier analysis may be performed. Even without asymmetry, though, some traffic patterns may have small numbers of flows with pronounced elephant flows, unbalancing the load across different links in what would be a well-formed ES for busier traffic patterns. In these cases, ES requirements are not satisfied—i.e. not all components in the ES are likely to handle the same load, so the performance under non-faulty conditions is unlikely to be similar—and we can offer no guarantees. Qualitatively, we expect that since our method is geared towards flagging 'worse than normal' performance, it is liable to flag the performance on the worst performing (i.e. most heavily loaded) component in the ES.

On the other hand, we may sometimes (inadvertently and incorrectly) execute our methodology on a set of components that do not correctly form an ES; e.g., a set of LAGs where one LAG aggregates less bandwidth than the others (perhaps due to a single link undergoing unplanned or undocumented maintenance). Here, we examine how unaccounted-for topology asymmetry can impact the precision and recall of our fault-localization methodology.

Consider an ES of N links (or LAGs) where $N-1$ links (*NORMAL*) provide 1 Gbps of bandwidth and the last link (*LOWCAP*) provides 500 Mbps. For a given offered load, *LOWCAP* will experience greater congestion and exhibit greater loss. Thus, metrics such as TCP retransmits or `cwnd` will diverge for *LOWCAP* and trigger our fault-detection system. We hypothesize that this behavior both causes false positives (where we flag lower-performance components as faulty) and false negatives (where we fail to notice the loss or delay induced by a partial fault, compared to that measured at lower performance but non-faulty links). Further, we predict that the degree of these effects will vary in a hard-to-characterize manner depending on the disposition of the traffic pattern.

To demonstrate these effects, we consider synthetic Mininet-hosted [25] workloads. We use an ECMP-routed $k = 4$ fat-tree network with 4 Agg-to-core uplinks per pod, each normally 1 Gbps. We emulate a Facebook-style client-server traffic pattern, where three Web-server pods exchange messages with a fourth cache pod. Message sizes are drawn from known workload parameters [40]. We vary link bandwidth on one Agg uplink (*LOWCAP*), reducing bandwidth to induce more asymmetry within the (now compromised) ES. On another link (*FAULTLINK*), we induce progressively higher rates of

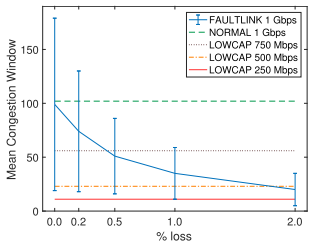


Fig. 9. cwnd sensitivity to link imbalance and fault-driven loss.

randomized packet-loss; here, we depict loss-behavior for the normal (unperturbed) Facebook workload. The remaining links (*NORMAL*) remain fault-free at 1 Gbps in all cases.

Figure 9 depicts cwnd sensitivity to link-bandwidth imbalance. Each horizontal line depicts median cwnd across all flows traversing *LOWCAP*, with the Facebook workload, at varying degrees of diminished bandwidth (representing LAGs of varying capacity shortfall). As bandwidth drops from 1 Gbps (normal) to 500 Mbps, median cwnd drops from 100 to just over 20 as congestion increases—a significant drop. Thus, asymmetric bandwidth can reduce precision and induce false-positives. Consequently, partial-fault-induced cwnd degradation on *FAULTLINK* may be undetected for loss rates under 1% in this case (though it is readily visible if *LOWCAP* offers 750 Mbps bandwidth instead). Thus, asymmetric bandwidth can reduce recall and induce false-negatives as well.

While it may be tempting to account for ES asymmetry (e.g. reducing false-positives by only considering deviations larger than those normally caused by the lower-capacity link), this is unfeasible in general. Asymmetry-driven deviation depends on the traffic pattern; e.g. in the 75% imbalance case (750 Mbps bandwidth when the normal value is 1 Gbps), for the regular Facebook workload, the mean cwnd across the narrower link is about 60% of the nominal value. If we halve the message sizes, though, the narrow-link mean cwnd only drops by around 5–10% (not shown); further reducing message sizes to a single packet leads to no deviation at all as utilization and congestion reduces. Moreover, traffic patterns can change in seconds, either due to load variations [14] or failures elsewhere in the network [41], neither of which we can predict.

In general, attempting to account for imbalance will be both application and load specific; hence, imbalances will be difficult to quantify or predict. Thus, applying our methodology to improperly formed ESes is fraught with danger; while we may yet catch egregious partial faults, we enjoy little of the sensitivity and precision that our methodology exhibits when applied to valid ESes. For example, in the 75%-imbalance case, minuscule drops of 0.2% are no longer noticeable when viewed against congestion losses incurred on the narrower link. With 50% imbalance (i.e. a lag with two links where most have four), only a 2% loss rate induces enough deviation to notice.

VIII. Summary

We have demonstrated a method to use the regularity inherent in real-world datacenters to help rapidly detect and localize the effects of partial failures, for a variety of production services with vastly different traffic characteristics. In particular, we have shown that we can do so with reasonable computational overhead and ease of deployment. We believe improvements in network core visibility, together with coordinating end-host performance monitoring with centralized network control, can aid the task of managing network reliability.

Acknowledgments

The authors are indebted to A. Schulman, A. Mirian, D. Huang, D. Maltz, and S. Cauligi for feedback on earlier drafts. A. Eckert, A. Starovoitov, D. Neiter, H. Morsy, J. Williams, N. Davies, P. Lapukhov and Y. Pisetsky provided invaluable insight into the inner workings of Facebook services. Finally, the authors thank O. Baldonado for his support of our collaboration.

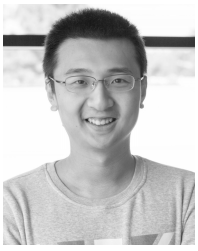
References

[1] *AB—Apache HTTP Server Benchmarking Tool*. Accessed: Jan. 30, 2017. [Online]. Available: https://httpd.apache.org/docs/2.4/programs/ab.html
[2] *BPF Compiler Collection (BCC)*. Accessed: Jan. 30, 2017. [Online]. Available: https://github.com/iovisor/bcc/
[3] *Extending Extended BPF*. Accessed: Jan. 30, 2017. [Online]. Available: https://lwn.net/Articles/603983/
[4] *Hadoop*. Accessed: Jan. 30, 2017. [Online]. Available: http://hadoop.apache.org/
[5] *HHVM*. Accessed: Jan. 30, 2017. [Online]. Available: http://hhvm.com
[6] A. Adams, P. Lapukhov, and H. Zeng. (2016). *NetNORAD: Troubleshooting Networks via End-to-End Probing*. [Online]. Available: https://code.facebook.com/posts/1534350660228025/netnorad-troubleshooting-networks-via-end-to-end-probing/
[7] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen, "Performance debugging for distributed systems of black boxes," in *Proc. ACM SOSP*, Bolton Landing, NY, USA, 2003, pp. 74–89.
[8] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Proc. ACM SIGCOMM*, Seattle, WA, USA, 2008, pp. 63–74.
[9] A. Andreyev. (2014). *Introducing Data Center Fabric, the Next-Generation Facebook Data Center Network*. [Online]. Available: https://code.facebook.com/posts/360346274145943
[10] B. Arzani *et al.*, "007: Democratically finding the cause of packet drops," in *Proc. NSDI*, Renton, WA, USA, 2018, pp. 419–435.
[11] B. Arzani, S. Ciraci, B. T. Loo, A. Schuster, and G. Outhred, "Taking the blame game out of data centers operations with NetPoirot," in *Proc. ACM SIGCOMM*, Florianopolis, Brazil, 2016, pp. 440–453.
[12] P. Bahl *et al.*, "Towards highly reliable enterprise network services via inference of multi-level dependencies," in *Proc. ACM SIGCOMM*, Kyoto, Japan, 2007, pp. 13–24.
[13] D. Banerjee, V. Madduri, and M. Srivatsa, "A framework for distributed monitoring and root cause analysis for large IP networks," in *Proc. IEEE Int. Symp. Reliable Distrib. Syst.*, Niagara Falls, NY, USA, Sep. 2009, pp. 246–255.
[14] T. Benson, A. Anand, A. Akella, and M. Zhang, "MicroTE: Fine grained traffic engineering for data centers," in *Proc. ACM CoNEXT*, Tokyo, Japan, 2011, Art. no. 8.
[15] N. Bronson *et al.*, "TAO: Facebook's distributed data store for the social graph," in *Proc. USENIX ATC*, San Jose, CA, USA, 2013, pp. 49–60.
[16] M. Y. Chen *et al.*, "Path-based failure and evolution management," in *Proc. NSDI*, San Francisco, CA, USA, 2004, pp. 309–322.
[17] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: Problem determination in large, dynamic Internet services," in *Proc. Dependable Syst. Netw.*, Bethesda, MD, USA, 2002, pp. 595–604.
[18] Cisco. *BGP Support for TTL Security Check*. Accessed: Jan. 30, 2017. [Online]. Available: http://www.cisco.com/c/en/us/td/docs/ios/12_2s/feature/guide/fs_btsh.html
[19] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proc. OSDI*, San Francisco, CA, USA, 2004, pp. 137–150.
[20] N. Duffield, F. L. Presti, V. Paxson, and D. Towsley, "Network loss tomography using striped unicast probes," *IEEE/ACM Trans. Netw.*, vol. 14, no. 4, pp. 697–710, Aug. 2006.

[21] Facebook. *Facebook Warm Storage—Next Generation Storage for Data Warehouse in Hadoop Ecosystem*. Accessed: Jun. 21, 2018. [Online]. Available: https://events.static.linuxfound.org/sites/events/files/slides/WarmStorage%20for%20Conference.pdf
[22] G. Forman, M. Jain, M. Mansouri-Samani, J. Martinka, and A. C. Snoeren, "Automated whole-system diagnosis of distributed services using model-based reasoning," in *Proc. IFIP/IEEE Workshop Distrib. Syst. Oper. Manage.*, Newark, NJ, USA, Oct. 1998.
[23] A. Greenberg. (2016). *PingMesh + NetBouncer: Fine-Grained Path and Link Monitoring for Data Centers*. [Online]. Available: https://atscaleconference.com/videos/pingmesh-netbouncer-fine-grained-path-and-link-monitoring-for-data-centers/
[24] C. Guo *et al.*, "Pingmesh: A large-scale system for data center network latency measurement and analysis," in *Proc. ACM SIGCOMM*, London, U.K., 2015, pp. 139–152.
[25] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown, "Reproducible network experiments using container-based emulation," in *Proc. ACM CoNEXT*, Nice, France, 2012, pp. 253–264.
[26] 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," in *Proc. NSDI*, Seattle, WA, USA, 2014, pp. 71–85.
[27] T. Hoff. *Latency is Everywhere and it Costs You Sales—How to Crush it*. [Online]. Available: http://highscalability.com/latency-everywhere-and-it-costsyou-sales-how-crush-it
[28] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *Proc. ACM EuroSys*, Lisbon, Portugal, 2007, pp. 59–72.
[29] A. Kabbani, B. Vamanan, J. Hasan, and F. Duchene, "FlowBender: Flow-level adaptive routing for improved latency and throughput in datacenter networks," in *Proc. ACM CoNEXT*, Sydney, NSW, Australia, 2014, pp. 149–160.
[30] S. Kandula, D. Katabi, and J.-P. Vasseur, "Shrink: A tool for failure diagnosis in IP networks," in *Proc. ACM MineNet*, Philadelphia, PA, USA, 2005, pp. 173–178.
[31] R. R. Kompella, J. Yates, A. Greenberg, and A. C. Snoeren, "IP fault localization via risk modeling," in *Proc. NSDI*, Boston, MA, USA, 2005, pp. 57–70.
[32] R. R. Kompella, J. Yates, A. Greenberg, and A. C. Snoeren, "Detection and localization of network black holes," in *Proc. INFOCOM*, Anchorage, AK, USA, 2007, pp. 2180–2188.
[33] R. R. Kompella, J. Yates, A. Greenberg, and A. C. Snoeren, "Fault localization via risk modeling," *IEEE Trans. Dependable Secur. Comput.*, vol. 7, no. 4, pp. 396–409, Oct./Dec. 2010.
[34] D. Maltz, private communication, Feb. 2017.
[35] V. Mann, A. Vishnoi, and S. Bidkar, "Living on the edge: Monitoring network flows at the edge in cloud data centers," in *Proc. IEEE COMSNETS*, Bengaluru, India, 2013, pp. 1–9.
[36] R. N. Mysore, R. Mahajan, A. Vahdat, and G. Varghese, "Gestalt: Fast, unified fault localization for networked systems," in *Proc. USENIX ATC*, Philadelphia, PA, USA, 2014.
[37] S. Radhakrishnan, M. Tewari, R. Kapoor, G. Porter, and A. Vahdat, "Dahu: Commodity switches for direct connect data center networks," in *Proc. ANCS*, San Jose, CA, USA, 2013, pp. 59–70.
[38] P. Reynolds, J. L. Wiener, J. C. Mogul, M. K. Aguilera, and A. Vahdat, "WAP5: Black-box performance debugging for wide-area systems," in *Proc. ACM WWW*, Edinburgh, U.K., 2006, pp. 347–356.
[39] M. Roughan, T. Griffin, Z. M. Mao, A. Greenberg, and B. Freeman, "IP forwarding anomalies and improving their detection using multiple data sources," in *Proc. ACM SIGCOMM Workshop Netw. Troubleshooting*, Portland, OR, USA, 2004, pp. 307–312.
[40] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network," in *Proc. ACM SIGCOMM*, London, U.K., 2015, pp. 123–137.
[41] A. Roy, H. Zeng, J. Bagga, and A. C. Snoeren, "Passive realtime datacenter fault detection and localization," in *Proc. NSDI*, Boston, MA, USA, 2017, pp. 595–612.
[42] A. Singh *et al.*, "Jupiter rising: A decade of Clos topologies and centralized control in Google's datacenter network," in *Proc. ACM SIGCOMM*, London, U.K., 2015, pp. 183–197.
[43] D. Turner, K. Levchenko, J. C. Mogul, S. Savage, and A. C. Snoeren, "On failure in managed enterprise networks," Hewlett-Packard Labs, Palo Alto, CA, USA, Tech. Rep. HPL-2012-101, May 2012.
[44] D. Turner, K. Levchenko, A. C. Snoeren, and S. Savage, "California fault lines: Understanding the causes and impact of network failures," in *Proc. ACM SIGCOMM*, New Delhi, India, 2010, pp. 315–326.
[45] B. P. Welford, "Note on a method for calculating corrected sums of squares and products," *Technometrics*, vol. 4, no. 3, pp. 419–420, 1962.
[46] X. Wu *et al.*, "NetPilot: Automating datacenter network failure mitigation," in *Proc. ACM SIGCOMM*, Helsinki, Finland, 2012, pp. 419–430.
[47] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz, "DeTail: Reducing the flow completion time tail in datacenter networks," in *Proc. ACM SIGCOMM*, Helsinki, Finland, 2012, pp. 139–150.
[48] Y. Zhu *et al.*, "Packet-level telemetry in large datacenter networks," in *Proc. ACM SIGCOMM*, London, U.K., 2015, pp. 479–491.

Arjun Roy received the B.Sc. degree in computer science from Columbia University in 2009, the M.S. degree in computer science from Stanford University in 2011, and the Ph.D. degree in computer science from the University of California at San Diego in 2018. His research interests include datacenter networking, particularly when it comes to fault detection, localization, and mitigation.

Rajdeep Das received the B.Tech. degree in information technology from the West Bengal University of Technology in 2013, and the M.S. degree in computer science from the Indian Institute of Technology Kanpur in 2015. He is currently pursuing the Ph.D. degree with the Computer Science and Engineering Department, University of California, San Diego. He was a Research Fellow with Microsoft Research India from 2015 to 2017. His research interest includes the broad area of computer networking.



Hongyi Zeng received the Ph.D. degree from Stanford University, under the supervision of Prof. N. McKeown and Prof. G. Varghese. He is currently an Engineering Manager and a Research Scientist with the Facebook Net Systems team. He works on intra- and inter-DC network monitoring and analytics, troubleshooting tools, and security tools. His current research interests include software-defined network, network verification, and programmable hardware.

Jasmeet Bagga received the M.S. degree from the University of Southern California in 2005. He worked at an early-stage start-up building network analytics software. He is currently a Software Engineer with Facebook, working on Facebook's in-house router/switch software called FBOSS.

Alex C. Snoeren (S'00–M'03–SM'18) received the B.Sc. degree in computer science, the B.Sc. degree in applied mathematics, and the M.S. degree in computer science from the Georgia Institute of Technology in 1996, 1997, and 1997, respectively, and the Ph.D. degree in computer science from the Massachusetts Institute of Technology in 2003. He is currently a Professor with the Computer Science and Engineering Department, University of California at San Diego, where he is also a member of the Systems and Networking Research Group. His research interests include operating systems, distributed computing, and mobile and wide-area networking.