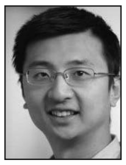
SYSADMIN

Passive Realtime Datacenter Fault Detection and Localization

ARJUN ROY, HONGYI ZENG, JASMEET BAGGA, AND ALEX C. SNOEREN

Arjun Roy is a graduate student in the Computer Science and Engineering Department at the University of California, San Diego, in the Systems and Networking Research Group. His research interests include datacenter networking, particularly when it comes to fault detection, localization, and mitigation. arroy@cs.ucsd.edu


Hongyi (James) Zeng is an Engineering Manager and Research Scientist on the Facebook Net Systems team. He works on intra- and inter-DC network monitoring and analytics, troubleshooting tools, and security tools. He received his PhD from Stanford University, co-advised by Professor Nick McKeown and Professor George Varghese. His current research interests include software-defined networks, network verification, and programmable hardware. zeng@fb.com

Jasmeet Bagga received his master's from the University of Southern California in 2005 and then worked at an early-stage startup building network analytics software. Jasmeet is currently a Software Engineer at Facebook, working on Facebook's in-house router/switch software called FBOSS. jasmeetbagga@fb.com

Alex C. Snoeren is a Professor in the Computer Science and Engineering Department at the University of California, San Diego, where he is a member of the Systems and Networking Research Group. His research interests include operating systems, distributed computing, and mobile and wide-area networking. snoeren@eng.ucsd.edu

Datacenters are characterized by their large scale, comprising a large number of network links and switches. However, these hardware components can develop intermittent faults, resulting in randomly occurring packet drops or delays that harm application performance—several such faults occur daily in large production datacenters. Since the effects are intermittent, traditional detection techniques involving end-host and router statistics or active probe traffic can fall short in their ability to identify and locate these errors. In this article, we present our passive hybrid approach that combines network path information with end-host-based statistics to rapidly detect and pinpoint the location of datacenter network faults inside a production Facebook datacenter.

Modern datacenters continue to increase in scale, speed, and complexity. Unfortunately, experience indicates that modern datacenters are rife with hardware and software failures—indeed, they are designed to be robust to large numbers of such faults. The large scale of deployment both ensures a non-trivial fault incidence rate and complicates the localization of these faults. Recently, authors from Microsoft described [9] a rogue's gallery of datacenter faults: dusty fiber-optic connectors leading to corrupted packets, switch software bugs, hardware faults, incorrect ECMP load balancing, untrustworthy counters, and more. Confounding the issue is the fact that failures can be intermittent and partial: rather than failing completely, a link or switch might only affect a subset of traffic, complicating detection and diagnosis. To illustrate this difficulty, the authors of NetPilot [8] 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 network goose chase to identify the faulty device.

We present our approach [5] to detect and localize such faults by providing greater visibility into the fate of application traffic once it is injected into the network—specifically, by exposing network path information for all datacenter traffic to the end-hosts. This allows us to correlate poor network performance observed at each end-host to the specific component in the network that is responsible passively, without any probe traffic overhead. Furthermore, we find that the vast amount of data available—we use TCP state machine data for every flow on every host—allows us to do so fairly rapidly.

Current Methods

Commonly deployed network monitoring approaches include end-host monitoring (e.g., RPC latency and TCP retransmits) and switch-based monitoring (e.g., drop counters and queue occupancies). However, such methods can fall short for troubleshooting datacenter-scale networks. Host monitoring alone lacks specificity in the presence of large numbers of alternative paths, which is characteristic of datacenter topologies [2, 7]. An application suffering from dropped packets or increased latency does not give any insight on where the fault is located, or whether a given set of performance anomalies are due to the same faults. Similarly, if a switch drops a packet, the operator is unlikely to know which application's traffic was impacted or, more importantly, what is to blame. Even if a switch samples dropped packets, the operator might not have a clear idea of what traffic was impacted. Due to sampling bias favoring high volume flows, mouse flows experiencing loss might be missed. Switch-counter-based approaches are further confounded by cut-through forwarding and unreliable hardware [8, 9]. Recent work [9] uses detailed path tracing of a subset of network traffic, combined with a modicum of active probe traffic, to debug where packets are dropped in the network and why. We argue that, for the highly regular topologies used by current datacenter networks, it should be possible to determine path information for all traffic. Luckily, common datacenter topologies are particularly amenable to providing this functionality.

Getting Path Information Scalably

Facebook's datacenters consist of thousands of hosts and hundreds of switches grouped into a multi-rooted, multi-level tree topology [2]. Figure 1 describes a simplified view of this topology, focusing on one of the several identical "pods" in the network. Each pod consists of several tens of Top-of-Rack (ToR) switches, each responsible for a few tens of servers. Each pod also contains four aggregation switches (Aggs) that enable inter-rack communication; every ToR connects to every Agg in the pod. Pods are in turn interconnected by a layer of core switches; each Agg is connected to a disjoint subset of core switches.

The network uses equal-cost multipath (ECMP) routing. When a host communicates with a host in another rack, a hash function at the ToR switch determines which ToR-to-Agg uplink the packets traverse based on fields such as source and destination IP addresses and network ports. Similarly, when a host communicates with a host in another pod, a hash function at the Agg switch determines which Agg-to-Core uplink is used.

For cross-pod traffic, once a packet reaches the core layer, there is only one path leading to the destination server. Thus, if we know the start and end point of a packet (from the source and destination IP address) and the core switch it transits, the receiving host can learn the entire path traversed. Thus, we assign an ID to each core switch and install a rule on the core switch instructing it to stamp every packet it forwards with this ID (see Figure 2).

Pinpointing a Fault to a Link

Once we have full path information, we could theoretically associate packet loss with a particular network path. However, this doesn't tell us which link along the path is responsible. An observation about the traffic engineering employed at Facebook aids us, however.

A significant amount of engineering effort has been targeted at calming hotspots at the application level to ensure that no particular server is overloaded by requests [4]. Specifically, for a front-end datacenter containing Web and cache servers (which

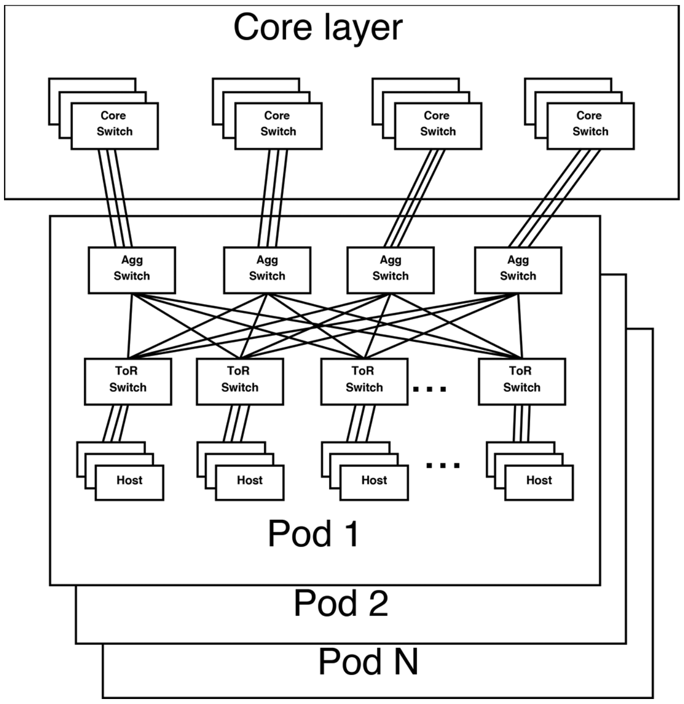


Figure 1: Facebook datacenter topology

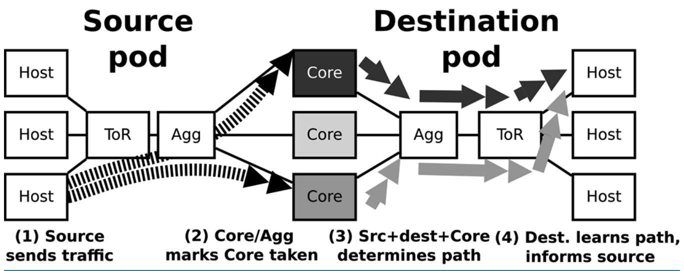

Source pod
Destination pod
Host
ToR
Agg
Core
(1) Source sends traffic
(2) Core/Agg marks Core taken
(3) Src+dest+Core determines path
(4) Dest. learns path, informs source

Figure 2: Determining flow network path

cache user data stored by back-end databases), every Web server spreads its requests for user data across all the cache servers in the datacenter. Furthermore, these requests are individually quite small and evenly spread, but in aggregate constitute the bulk of traffic within the network. This application-level load balancing is in addition to normal network load balancing techniques like ECMP routing.

Consequently, if we look at a level of the multirooted tree topology (for example, every Agg to Core link in a pod, or every ToR to Agg link in a pod), every link in the group we examine has a very even load—both in terms of number of flows and number of bytes handled—on short timescales of just a few seconds. The aggregate performance of the flows for any given link is similar to that of the flows of any other link within the set—we call it an "equivalence set" of links.

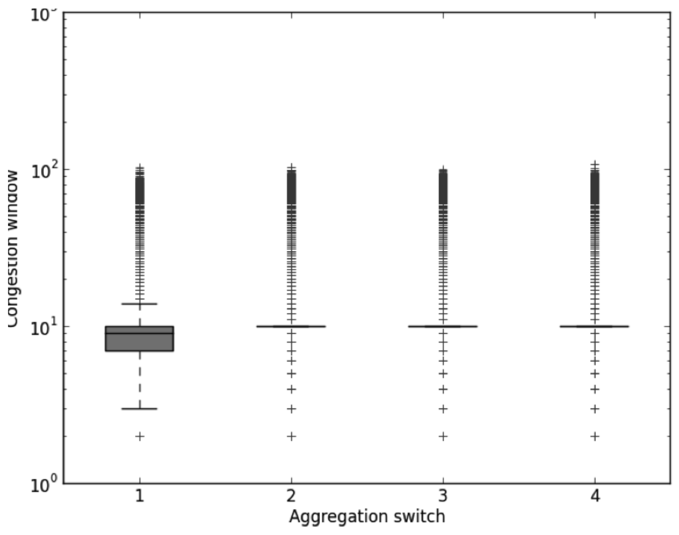


Figure 3: TCP congestion window distribution per ToR uplink for cache server. The ToR to aggregation switch 1 link has 0.5% randomized packet loss, which has shifted the distribution towards smaller values.

On the flip side, if one of the links is faulty, it sticks out like a sore thumb—the aggregate flow performance diverges compared to the other links in the set. Thus, if we pick a metric that is correlated with packet loss—for example, TCP retransmits or congestion window—we can compare the distributions of the metric across links and perform outlier analysis to pinpoint the faulty link. Figure 3 depicts the distributions for TCP congestion window for each ToR to Agg uplink traversed by traffic destined for a single cache server, where link 1 (out of four total) has a 0.5% induced packet loss rate; note the significant skew to lower values present for the distribution corresponding to that link.

Outlier Analysis

While it is visually apparent that the distributions of performance metrics across links are impacted by the presence of a fault, we need a way to automate the process of using this information to generate a verdict for every combination of (host, link) to determine whether the link is faulty or not. Fundamentally, the question we are asking is: does a particular link have more retransmits (or say, a smaller flow congestion windows) than the others? If so, maybe there is a fault at that link!

To answer this question, we use the Student's t-test. The t-test determines whether a given distribution has a mean that is higher than another distribution. It is amenable to efficient streaming computation (a prototype implementation of our system uses approximately 0.5% of CPU on a production Web server) and runs on every host, with each host examining its own traffic. Note, however, that this raises the chance of false positives, where a link might temporarily have worse performance distributions, possibly due to effects like transient congestion. Given a large number of hosts, it is certain that some subset of them will incorrectly flag a link as faulty. We have to account for these false positives.

The observation we leverage is that false positives, in the absence of an actual fault, ought to be evenly distributed among links due to the high degree of load balancing. Thus, we aim to filter out the false positives by asking the question: are links being claimed as faulty roughly evenly, or is there a particular link (or group of links) that is (are) being accused more than the others? For that we use the chi-squared test, which is used to determine whether the frequency distribution of a set of events matches some theoretical distribution. The chi-squared test sees whether the claims that a link is faulty is evenly spread among all the links considered, or if a particular subset of links have a significantly higher percentage of hosts claiming fault. It outputs a p-value ranging from 0 to 1. If the outputted p-value is "close" to 0 (a common cutoff is 0.05), then the link with the most "faulty verdicts" is considered to actually be faulty. In the case of multiple errors, that link can be removed from the set considered, and the chi-squared test can be run again.

Putting It All Together

We combine the functional components described thus far into an always-active fault detection system. Our system involves functional components at all servers, a subset of switches, and a centralized aggregator, depicted step-by-step in Figure 4. Switches mark packets (1) to indicate network path as described before. Hosts then independently compare the performance of their own flows to generate a host-local decision (a "verdict") about the health of all network components (2), performing outlier analysis using the Student's t-test on metrics such as TCP retransmits. Specifically, every host will output a verdict for the ToR to Agg and Agg to Core links in its own pod once every 10 seconds. These verdicts are sent (3) to a central aggregator, which filters false positives to arrive at a final set of faulty components using the chi-squared test (4), aggregating data and outputting a result once every 10 seconds (configurable to increase sensitivity as a tradeoff to reaction time) as well. We do not consider host to ToR uplinks in this system.

Detecting Faults

To validate our approach, we deployed a prototype of our fault detection system inside a production Facebook front-end datacenter serving user Web traffic. For the sake of reproducible experiments, we primarily focus on injected synthetic failures, which we describe momentarily. We also discuss experience gained in tracking down naturally occurring partial faults.

Induced Faults

Within one of Facebook's datacenters, we instrumented 86 Web servers spread across three racks with the monitoring infrastructure described previously. Path markings are provided by a

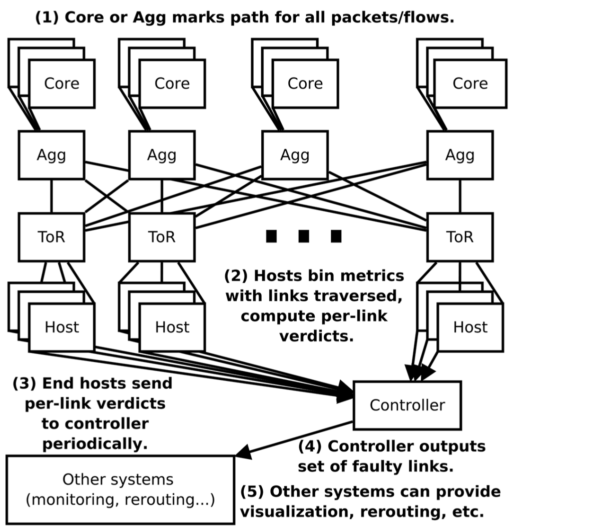

(1) Core or Agg marks path for all packets/flows.
Core
Core
Core
Core
Agg
Agg
Agg
Agg
ToR
ToR
ToR
(2) Hosts bin metrics with links traversed, compute per-link verdicts.
Host
Host
Host
(3) End hosts send per-link verdicts to controller periodically.
Controller
(4) Controller outputs set of faulty links.
Other systems (monitoring, rerouting...)
(5) Other systems can provide visualization, rerouting, etc.

Figure 4: High-level system overview (single pod depicted)

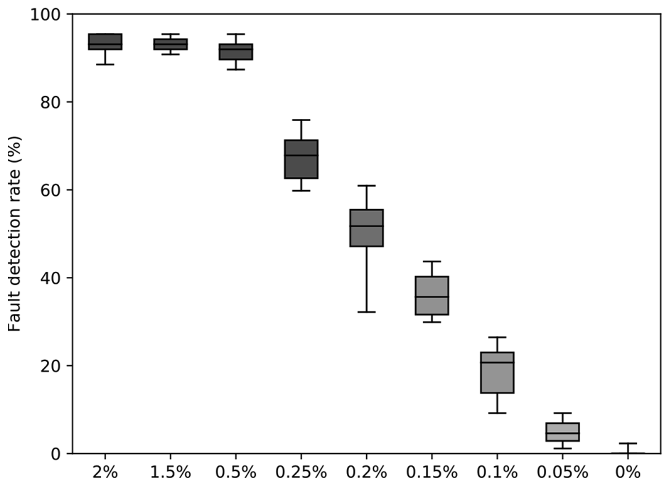


Figure 5: Single fault loss rate sensitivity

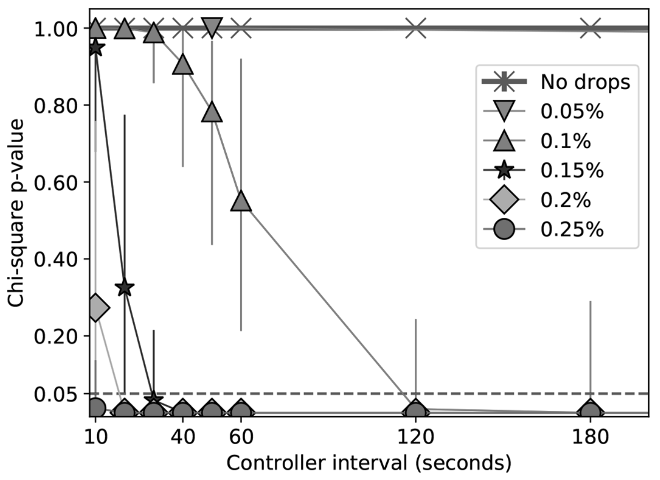


Figure 6: Controller interval required to find single fault vs. packet loss rate

single Agg switch, which sets DSCP bits based on the core switch from which the packet arrived. To inject faults, we use iptable 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.

First, for a single faulty core-to-Agg link, we depict the percentage of hosts that flag the faulty link as having an error as a function of the packet loss rate over consecutive 10-second intervals in Figure 5. For drop rates at 0.5% and higher, close to all of the hosts flag the link as faulty. For drop rates below 0.5%, we observe a linear drop off in the percentage of hosts that catch the fault.

Recall the aggregator, which gives us the overall verdict on per link health, looks for a non-trivial difference in the number of hosts that claim that a link is faulty before marking it faulty. Thus, over a 10-second interval, it might not decide that the fraction of hosts marking a link as faulty is significant for the smaller magnitude errors. Note, however, that since a fault is likely to persist for longer periods of times, we can simply run the aggregator for longer to catch an error. Instead of processing N verdicts over 10 seconds, we could aggregate and operate on $3N$ verdicts over 30 seconds. We find that this allows us to reliably catch the more slight errors as well, without inducing false positives in the no-error case.

Figure 6 depicts the amount of time needed by the aggregator to catch errors ranging from 0.25% packet loss down to 0.1%. Recall that a chi-squared test outputs a p-value, where if the p-value is “close” to 0 (we arbitrarily use 0.05 as our cutoff for “close”) it means that the link with the most faulty verdicts from the end-hosts is likely to, in fact, be faulty. Thus, we depict the p5, p50, and p95 for the p-values outputted by the aggregator for each packet loss rate. We see that a 20-second interval will reliably catch a 0.25% error—in other words, the aggregator almost always outputs a p-value of less than 0.05. However, a 0.15% packet loss rate requires 40 seconds, and we receive an intermittent signal for a 0.1% error—at least some portion of the time the aggregator will find no fault.

Naturally Occurring Faults

To determine whether our system can successfully detect and localize network anomalies in the wild, we deployed our system on 30 Web servers for a two-week period in early 2017, without inducing any synthetic errors. On January 25, 2017, the software agent managing a single switch linecard that our system was monitoring failed. The failure had no immediate impact on traffic, since the existing switch rule set installed by the agent remained in effect. 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.

Our system observed that as traffic was routed away from the failed linecard, the distributions of TCP's congestion window and slow start threshold metrics for the traffic remaining on the faulty linecard's links rapidly diverged from those associated with non-faulty linecards (Figure 7). The deviations are immediate and significant, with the mean congestion window for the faulty linecard dropping over 10% in the first interval after the majority of traffic is routed away, and continually diverging from the working links thereafter. Furthermore, the volume of measured flows 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 [1], 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. Thus, this anecdote shows that our system can complement existing fault-detection systems and provide rapid notification of significant changes in network conditions on a per-link or per-device basis.

Caveats

A couple of caveats apply to this methodology. First, it is conceivable that for more complicated topologies, a single stamp on a packet might not be enough to uniquely resolve the path. Prior work [6] has explored marking multiple packets with partial path information, such that the overall network path can be recovered by examining enough packets. We leverage a similar technique to generalize our packet-stamping mechanism. Suppose there is a maximum of H hops in the network between any pair of communicating servers. We provide every switch an ID instead of a select few. Suppose a flow has H or more packets. The first packet can be marked by the sender with some bits that instruct the first switch in the path to stamp the packet only—for example, we might use the IP TTL field to arrange this. The second can be marked so the second switch in the path marks it, and so on until we send H packets to recover the full path.

Second, when it comes to applying per-switch stamps, we need to choose where in the packet we apply it. Our prototype stamps the packet DSCP field, but this is limited since it is only 6-bits wide and frequently has other uses—typically for choosing switch-queueing policies. A solution that could scale to much larger networks would be to write to the IPv6 flow label field in the packet header, which is 20-bits wide. We are necessarily limited to what switch ASICs support, though the capabilities of switch ASICs has been progressively improving.

Future Directions

While our existing prototype has leveraged some favorable characteristics of the Facebook datacenter environment—most notably, the heavily load balanced traffic distribution—we are optimistic that our approach can generalize to datacenters with different and more variable application traffic patterns, such as Hadoop cluster workloads. Additionally, we hope that our findings incentivize router manufacturers to provide more options to allow packet header manipulation, perhaps through mechanisms such as P4 [3].

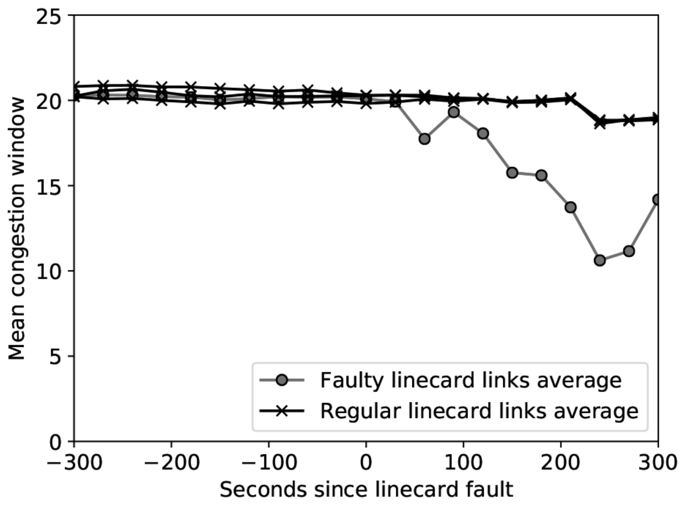


Figure 7: Mean cwnd per (host, link) during linecard fault

References

[1] A. Adams, P. Lapukhov, and H. Zeng, "NetNORAD: Troubleshooting Networks via End-to-End Probing," Facebook Code blog, Feb. 18, 2016: https://code.facebook.com/posts/1534350660228025/netnorad-troubleshooting-networks-via-end-to-end-probing/.

[2] A. Andreyev, "Introducing Data Center Fabric, the Next-Generation Facebook Data Center Network," Facebook Code blog, Nov. 14, 2014: https://code.facebook.com/posts/360346274145943, 2014.

[3] 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," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3 (July 2014), pp. 87–95: http://www.sigcomm.org/sites/default/files/ccr/papers/2014/July/0000000-0000004.pdf.

[4] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the Social Network's (Datacenter) Network," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '15)*, pp. 123–137: http://cseweb.ucsd.edu/~snoeren/papers/fb-sigcomm15.pdf.

[5] A. Roy, H. Zeng, J. Bagga, and A. C. Snoeren, "Passive Realtime Datacenter Fault Detection and Localization," in *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17)*, pp. 595– 612: https://www.usenix.org/system/files/conference/nsdi17/nsdi17-roy.pdf.

[6] S. Savage, D. Wetherall, A. Karlin, and T. Anderson, "Practical Network Support for IP Traceback," in *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '00)*, pp. 295–306: http://cseweb.ucsd.edu/~savage/papers/Sigcomm00.pdf.

[7] 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," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '15)*, pp. 183–197: http://conferences.sigcomm.org/sigcomm/2015/pdf/papers/p183.pdf.

[8] X. Wu, D. Turner, C.-C. Chen, D. A. Maltz, X. Yang, L. Yuan, and M. Zhang, "NetPilot: Automating Datacenter Network Failure Mitigation," in *Proceedings of the ACM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '12)*, pp. 419–430: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/netpilot.pdf.

[9] 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," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '15)*, pp. 479–491: https://www.cs.ucsb.edu/~ravenben/publications/pdf/everflow-sigcomm15.pdf.