Oh Flow, Are Thou Happy?
TCP sendbuffer advertising for make benefit of clouds and tenants

Alexandru Agache
University Politehnica of Bucharest

Costin Raiciu
University Politehnica of Bucharest

1 Introduction

Datacenter networks have evolved from simple trees to multi-rooted tree topologies such as FatTree [1] or VL2 [9] that provide many paths between any pair of servers to ensure high performance under all traffic patterns. The standard way to load balance traffic across these links is Equal Cost Multipathing that randomly places flows on paths. ECMP may wrongly place multiple flows on the same congested link, wasting as much as 60% of total capacity in a worst case scenarios for FatTree networks [2, 16]. These networks need information about the traffic they route to avoid collisions, by steering it towards idle paths [2, 7], or by creating more capacity on the fly between groups of hot racks [8, 21, 10, 24]. Additionally, ECMP creates uncertainty about the path a given flow has taken, making network debugging difficult.

Traffic information in datacenters comes mostly in the form of rate and loss rate measurements at various vantage points including switches, routers and hypervisors (this is less common with the advent to I/O passthrough). Additionally, monitoring protocols like sFlow allow sampling a percentage of packet headers at selected links. This information is simply not sufficient to enable satisfactory solutions to the problems faced by datacenter operators: we show in Section 2 that link utilization and aggregate loss rate measurements are poor indicators of network hotspots, and that the current threshold-based traffic engineering approaches are suboptimal when traffic is not network-limited.

What is the least amount of information from the endpoints that would enable the network to do a better job? We find that simply knowing whether an application is backlogged or not would help a lot. That is why we propose that senders include **sendbuffer occupancy**—the number of bytes waiting to be sent in the in-kernel buffer of a TCP connection—in the packets they originate to inform the network about the status of the connection. We propose an implementation of sendbuffer advertising that has zero bandwidth overhead: we encode the sendbuffer in the receive window field of TCP headers. Finally, we have performed an initial exploration of its applications, finding benefits in a varied range of scenarios.

2 Problem Statement

What's in a (flow) rate? Traffic engineering solutions like Hedera [2] or Devoflow [7] use a rate-based threshold to decide which flows they should schedule; this is heuristically chosen to be 10% of the host NIC speed. Flows above the threshold are assumed to be network bound and assigned completely idle paths in the hope they will fill them up. Flows below the threshold are simply ignored: the hope is that their traffic is insignificant.

We ran a `rsync` of large files between two machines connected via a 1GBps link and measured link utilization, finding it is around 33%: the app can't fill the pipe because the hard disk drives of the servers are bottlenecks, and cannot sustain 1Gbps throughput. In our `rsync` test, both Hedera's assumptions are broken. Our evaluation in Section 4 shows the performance penalty can be as high as 35% compared to the optimal; the exact hit depends on the traffic matrix. Previous work [16](§4.4) shows that the 10% threshold seems to be a sweetspot; changing it yields worse performance.

Loss rates are poor indicators of capacity shortage. Conventional wisdom says that the higher the loss rate at a link, the busier that link is. This is the case when connections arriving at a link are not synchronized, as duly confirmed by an experiment we ran on Amazon EC2, where one VM receives `iperf` traffic from multiple other VMs: loss rates for a single connection running one minute are around 0.1%, and this increases to 0.2% with 5 connections; the average throughput of all connections received is around 150Mbps, indicating that EC2 shapes its tenants' traffic.

Next, we run an experiment where one EC2 VM runs multiple rounds of a synthetic scatter-gather application, sending out short requests to 99 EC2 VMs which reply immediately with *B* bytes; the next round starts only when all the replies from the previous round have been received. We vary the size of the response *B* from one packet to tens of packets, running many rounds per experiment. We plot the average measured loss rates in Figure 3, where we vary the size of the reply on the X axis.

Loss rates in this scenario are much higher than in the previous experiment, yet the traffic rate is at most 10Mbps and thus the link is not a bottleneck: the problem comes from the synchronization of reply traffic which repeatedly overflows the shallow receive buffer of the link towards the destination server. This is a well known problem called *incast* that has been studied extensively (e.g. see [20, 23]), but identifying occurrences of it without packet-level traces is very difficult. A traffic engineering solution using loss rates might wrongly decide to migrate the frontend VM based on loss rates, only to find that the problem persists after the move.

Finally, the combination of link utilization and loss rates is not enough to predict bottlenecks either: when app-limited traffic shares a link with scatter-gather traffic, link utilization can be high (driven by the app-limited traffic) and loss rates will soar because of incast traffic. Migrating the traffic away from this link is pointless.

What is missing? In all examples above, the applications running at the endpoints are not bottlenecked by the network: `rsync` can't send faster, and the scatter-gather servers send very little data back: providing more bandwidth to these apps will not help. The network is unaware of the status of the application, and may take the wrong decisions when shifting traffic around. It should know whether endpoint apps are backlogged to take proper decisions. The most obvious choice is to try to infer this by examining their traffic.

There are two known methods to infer if an app is backlogged by looking at its traffic. Jaiswal et al. [14] propose a method that runs close to the sender and sees all packets to measure the flight size and estimate the congestion window for a number of well-known congestion controllers. If the flight size is smaller than the congestion window, the flow is app-limited. This technique has two drawbacks: it only works for known congestion controllers, and may struggle in clouds where tenants may run their new and fancy congestion control algorithms; secondly, it is very expensive to implement, as the only reasonable choice is to deploy it in the hypervisor (e.g. Dom0 in Xen). However, this requires that tenant traffic is routed through the hypervisor, instead of being sent directly to the NIC with techniques such as PCI passthrough / SR-IOV.

The second method also requires packet-level logs and looks for packets smaller than the maximum segment size. It is similarly expensive to implement, and doesn't have very good accuracy because it depends on whether Nagle's algorithm is enabled, and whether the application is using the push flag (which causes segments to be sent out immediately, rather than waiting for a full segment). We ran experiments that use `rsync` to copy a 4MB file across a 1Gbps link. When using the HDD, `rsync` is disk-bound and 3% of its segments are smaller than the maximum segment size; when reading the files from memory, `rsync` is network-bound yet 2% of the TCP segments have sizes smaller than MSS.

3 Sendbuffer Advertising

To ease the job of the network, endsystems should explicitly inform the network on whether their flows are backlogged at any point in time. The operating system can easily tell whether a TCP connection is backlogged by examining its sendbuffer, as shown in Figure 1. When the application issues a `send` system call, the kernel copies as much data as possible into the sendbuffer, and then sends segments on the wire as long as they fit in the congestion window and receive window advertised by the remote endpoint. The sendbuffer has two distinct parts: a) the segments that are in-flight have been already sent but not acknowledged by the receiver, and must be kept until ACKed, and b) the segments that are waiting to be sent, or the backlog.

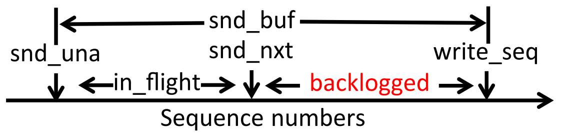


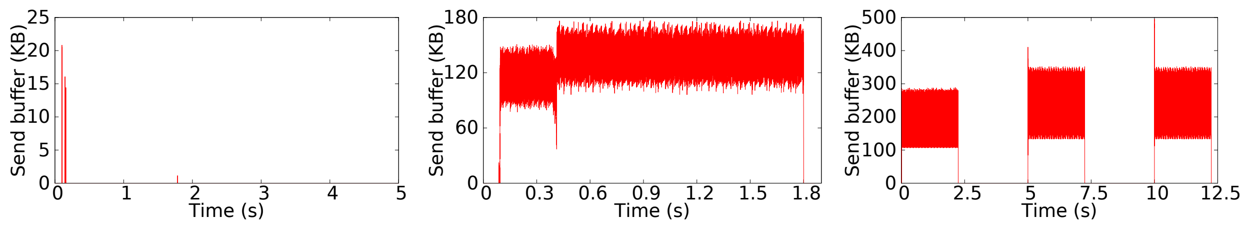
Figure 1: Anatomy of a send buffer

We propose to advertise the number of bytes in this backlog in every TCP segment. The advertisement can be easily placed in a new TCP option, but this would add 8B of overhead [1] to every segment (0.5%); worse yet, it may disable hardware offloading support on modern NICs, such as large receive offload, reducing performance.

Instead, we encode the sendbuffer advertisement in the receive window field of outgoing segments; we use one of the reserved flags to indicate that the receive window field has a different meaning in this segment; this encoding has zero bandwidth overhead. The stack encodes a sendbuffer advertisement only when the receive window and ACK combination is exactly the same as sent previously. Since traffic is unidirectional most of the time, the receive window advertisements from the data source are highly redundant (same ACK, same window) and replacing them with sendbuffer advertisements creates no performance issues. In the reverse direction, sendbuffer advertisements will not be sent since the ACK is progressing and the window is constantly evolving; luckily, the sendbuffer advertisements are usually not needed in this direction.

We have implemented TCP sendbuffer advertising in the Linux kernel; our patch is less than 100 lines of code. We first ran a few experiments to understand how different applications' sendbuffer information looks like. In Figure 2 we show the sendbuffer carried in the segments for three different apps: a HDD-bound `rsync`, a network-bound `rsync` (using a ramdisk to transfer the same file), and an on-off application that sends high-speed bursts followed by silence (such as video streaming). The figures show what we expected: there is vir-

[1] Two bytes for option type and size, four bytes for the sendbuffer advertisement, and two bytes of padding.



(a) HDD bound (b) Network bound (c) Bursty transfer

Figure 2: Bandwidth-hungry network transfers lead to heavy sendbuffer occupancy.

tually no backlog in HDD-bound scenarios, and it's easy to distinguish whenever the network becomes the bottleneck by examining sendbuffer advertisements.

4 Use Cases

Detecting network hotspots. We have seen that neither loss rate nor utilization are accurate indicators of bottlenecks in datacenter networks. A better way to estimate bottlenecks is to rely on sendbuffer information carried by packets. The most straightforward manner to measure this is to average the sendbuffer information of all packets on a specific link; alternatively, packet sampling can be used to only examine a subset of traffic.

We reran the EC2 experiments in Section 2 with our Linux kernel implementation of sendbuffer advertising. When running app-limited flows through the link, the average reported sendbuffer is zero. In the incast traffic pattern, the sendbuffer depends on the size of the client reply and is shown in Figure 3 as the linepoint graph corresponding to the second Y axis; the values are very low for small responses, and only increase for larger ones because the congestion window doesn't open fast enough at the very start of the connection; the max send buffer is around 60KB. Running network limited traffic through a link results in sendbuffer averages that are on the order of hundreds of KB, making it easy for datacenter operators to discover when a link is a bottleneck.

This information can be cheaply gathered by operators using sFlow: the sFlow server has visibility not only in the average sendbuffer at a given link, but also in per-flow sendbuffer information. This makes it possible to single out congested links and take corrective action (e.g. migrate VMs away from servers on that link), and to reroute bottlenecked flows onto paths that have more spare capacity. We discuss such a solution next.

Accurate traffic engineering. Traffic engineering solutions such as Hedera [2] make the assumption that any flow whose throughput is larger than some threshold will expand to fill the maximum available capacity. This assumption is wrong for app-limited traffic, and could lead

Connections	Hedera First Fit	Sendbuffer TE
NW	84%	82.7%
NW NW	87%	92%
NW 110Mbps	77%	84%
NW 90Mbps	68%	81%
NW 2·90Mbps	70%	76%
110Mbps 790Mbps	78%	83.5%
200Mbps 800Mbps	76%	81%

Table 1: Traffic engineering in datacenters using sendbuffer advertising

to inefficient path allocations.

To find out how bad this effect is, we implemented sendbuffer advertisement in the *htsim* simulator and tested how well traffic engineering performs on a 1Gbps FatTree topology with $k = 8$ (128 servers) [2]:

- The First Fit heuristic of Hedera [2] that places any flow larger than 100Mbps onto the first path that has enough capacity to fit it. Hedera estimated the capacity needed for a flow by counting the number of large flows originating or ending at the same server, and dividing the NIC capacity equally between these flows.

- A toy traffic engineering solution that only only schedules a flow if its sendbuffer has been non-empty for more 80% of the packets in the current measurement period and the achieved rate is smaller than 50% of the flow's maximum theoretical rate.

In all experiments, we ran a permutation traffic pattern where every server sends traffic to a single other server, and no destination receives traffic from more than one source. The number of connections between each pair of servers is varied from one to three. The table 1 gives the average flow throughput measured over a 10s experiment, expressed as a percent of the optimal throughput.

[2]Previous works have shown that topologies with two orders of magnitude flows show qualitatively similar results, but the simulation speeds are greatly reduced [16]

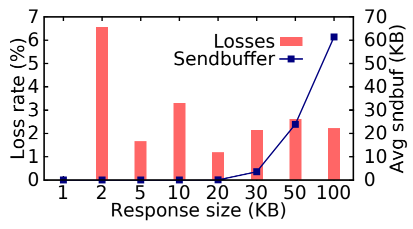


Figure 3: Losses caused by incast

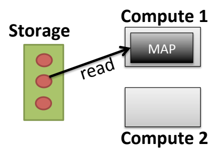


Figure 4: Datacenter compute frameworks can use sendbuffer information to make better scheduling decisions

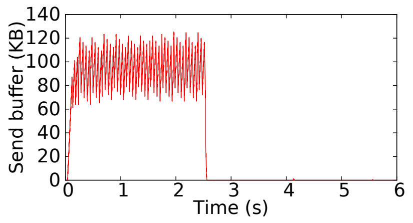


Figure 5: Sendbuffer on an emulated mobile device

The first column describes the connections that were instantiated between each server pair: *NW* means a network limited flow, while throughput numbers mean a flow limited to the specified bandwidth.

The results show that our simple TE scheme is similar in performance to Hedera when flows are network limited (first two rows of the table), but achieves better results when flows are app-limited. The improvement varies from 5% to 15% depending on the setup. Our scheme is nothing more than a proof-of-concept, so these numbers show that assuming all traffic is network limited can lead to inefficient traffic allocation (e.g. as low as 68% of optimal) and that traffic engineering algorithms that take into account sendbuffer information are more robust and have the potential to improve performance across a wide range of scenarios.

An interesting area of research is to explore sendbuffer TE techniques in the context of datacenter topologies where rack-to-rack bandwidth is augmented dynamically, such as Helios [8] or Flyways [10]. There, the cost of wrongly estimating a flow's needs are much higher, since the on-demand cross-rack capacity is often an order of magnitude higher than the basic one.

Troubleshooting flow performance. A side-effect of having send-buffer information in every packet is that it reveals information about the sender's congestion window. Assuming the total size of the sendbuffer is constant, the amount of bytes buffered by the receiver shrinks when its congestion window grows (and more packets are in flight), and increases when the congestion window is decreased upon a loss.

These trends are not immediately visible in the sendbuffer graphs because the sendbuffer also varies due to app writes into the buffer and the kernel sending packets out on the wire when possible. This is what we see, for instance, in Figure 2b, where the app writes/kernel drain create a 60KB band where the sendbuffer varies.

We observe that the app-writes operate on a much shorter timescale than congestion window updates (increases or decreases), so if we smooth out the app-write it should be possible to see the cwnd evolution against time just by monitoring the sendbuffer information.

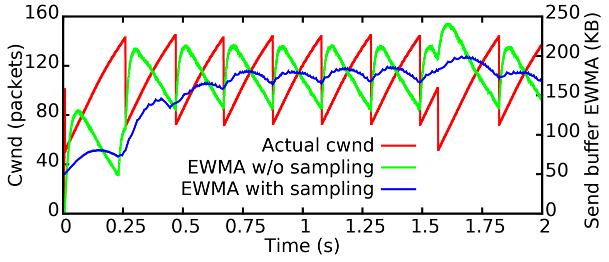


Figure 6: Sendbuffer information can be used to estimate the *cwnd* of the sender. Even when packets are sampled 1/100, the resulting information allows inferring losses suffered by the sender.

To see this effect, we run an iperf over a gigabit link; we add 1ms of delay to make it easier to read the congestion window graphs. We plot in Figure 6 the congestion window of the TCP connection and an exponential moving average (without sampling) of the sendbuffer information with $\alpha = 0.001$. We see that this estimate closely tracks $K - cwnd$ where K is the total size of the sendbuffer. We also sample the sendbuffer information 1/100 packets and plot an exponentially moving average of the sendbuffer in the same picture: losses experienced by the flow are easy to detect even with this information.

These results imply that it is possible to use sFlow sampling at rates of 1/100 (or possibly less) to cheaply get accurate estimates of per-flow loss rates. This functionality is only available today for links (i.e. flow aggregates). Per-flow loss rates enable debugging performance problems experienced by flows: it is possible to measure average loss rates across all flows, single out outliers and find out why they suffer more, etc.

Improving datacenter applications. Datacenter compute frameworks such as MapReduce use a mix of I/O and CPU-bound jobs that includes reading from local or network storage, processing data, write to local storage, data transfer, and so forth. Such applications can leverage sendbuffer information, if available, to improve their scheduling decisions and thus reduce job finish times.

We present one possible optimization in Figure 4. A map task is running on compute server 1, reading data from a storage server. If this task is late compared to the other maps in its generation, it is called a straggler and map-reduce implementations (such as Hadoop) will opportunistically schedule the map onto another machine (say compute server 2), hoping it will finish quicker. This strategy implicitly assumes that the bottleneck is somehow related to the mapper.

This assumption can of course be wrong; the storage node can be I/O bound: starting a new map job on compute server 2 will not help; the same data may need to be read again (depending on memory usage), competing for the scarce bandwidth of the the storage node's disks. If the mapper computes average sendbuffer occupancy for its TCP connection to the storage server, it can report this to the controller next time map statistics are sent. The controller can then see the connection is I/O bound on the storage node, and realize that firing another map job that would read from the same node is pointless.

Helping mobile clients. Mobile clients have multiple wireless interfaces including WiFi and cellular. Today, only one connection is used at a time: WiFi if available and cellular otherwise. The implicit assumption is that the user wants to avoid cellular links at all costs, as long as WiFi "works". Unfortunately, WiFi performance can be arbitrarily low, crippling application performance. Multipath TCP [18] is an extension to TCP that allows a mobile to utilize both WiFi and cellular links simultaneously, spreading traffic over both, and can be readily used to increase network capacity when WiFi is not good enough. Always enabling both WiFi and cellular links wastes energy consumption and precious mobile data, thus enabling cellular links when WiFi is available should be done with care.

We can use sendbuffer information to understand whether WiFi is sufficient, assuming that the user will specify which apps are allowed to increase capacity when needed. These may include VOIP and streaming apps that have (mostly) inelastic bandwidth demands. We have coded a Python application that monitors the sendbuffer and will open a new subflow over another interface when an app is backlogged. We have run this on a Linux box with dual gigabit interfaces, (limited to 20Mbps each). A transfer is initiated from a source which generates data at 30Mbps. The send buffer advertisements that we receive are indicative of typical network-limited flow behaviour. After a new subflow is initiated, the advertised values quickly drop to zero, and only increase slightly from time to time due to the operation of MPTCP. Figure 5 shows the evolution of the send buffer information received on the primary subflow. The opening of the second subflow is delayed in order to provide better contrast between the two states. After both subflows are enabled, the sendbuffer is virtually empty as the available bandwidth is no longer a limiting factor.

5 Related work

The problem we address is the lack of information in the network about the desires of the endpoints. We are not the first to notice this problem or to offer solutions to it.

XCP proposes that routers and endpoints tightly collaborate to guide congestion control, rather than relying on endsystem probing alone as in TCP [15]. In particular, every endpoint places in each packet the expected demand for the next round-trip time, and this allows the routers to allocate capacity appropriately. XCP did not get deployed despite its potential benefits: it is simply impossible to change all routers to support it. Sendbuffer advertising is much simpler, has zero-overhead and does not require network changes.

Mahout wants to enable more accurate traffic engineering by detecting elephant flows at endpoints, not in the network [6]. Endhosts monitor the sendbuffer usage of their connections; if occupancy grows past a threshold, the flow is classified as elephant and it is marked (with a DSCP codepoint) so that the traffic engineering system can then schedule it appropriately. Our solution shares the basic idea of using sendbuffer information to detect backlogged apps, but we expose the actual data to the network instead of checking a sendbuffer occupancy threshold. This ensures our solution is applicable to a wider range of use cases, and actual sendbuffer values (as opposed to binary indications of full/empty) allow to infer second order information such as losses.

B4 [13] and SWAN [12] are inter-datacenter traffic engineering solutions that rely on application changes or ingress rate-limiting to ensure core load is predictable; in these settings, B4 and SWAN use SDN paradigms to attempt near perfect traffic engineering. To achieve their benefits, both solutions change applications to report their demands to the network; while changing a few apps at Google is certainly feasible, changing all datacenter apps run by tenants is certainly not. Sendbuffer advertising aims to allow better traffic engineering without changing endhost apps.

HONE [19] is a scalable and programmable platform for traffic management. Its architecture includes agents running on each host that can collect various data about connections, including socket backlog information. However, this is relayed and used at an entirely different scope, while we focus on adding sendbuffer advertisements to the packets themselves. Varys [5] proposes that application schedulers such as MapReduce make it clear to the network which flows are related, so that traffic engineering can optimize for job completion time, rather than individual flow completion time. A similar solution is proposed by CloudTalk [17]. These are complementary to our work, and can be used in conjunction with it.

6 Discussion

We have proposed sendbuffer advertisement, a minuscule patch to TCP stacks that has zero bandwidth overhead in the average case and provides minimal yet crucial information to the network about the connections it carries. This extension requires no application changes and helps improve networks by enabling accurate traffic engineering, reliable bottleneck detection, per flow loss rate monitoring and application optimizations.

Sendbuffer advertising is particularly beneficial for cloud environments were the cloud provider has currently no means to understand what its tenant apps need. While changing tenant apps to provide such information is infeasible, we believe incentivising tenants to report sendbuffer usage is possible.

At HotCloud we would like to get feedback on the feasibility of introducing sendbuffer advertising in cloud networks, and on discovering other use cases we haven't considered. Our main target is to interact with industry people to better understand real-life constraints and applications. In addition, we seek guidance on a number of important open problems:

1.Preventing cheating. Malicious tenants may lie about their sendbuffer occupancy to fool traffic engineering (or other cloud provider apps) and gain more throughput or attack other customers. There are two possible strategies to prevent cheating: a) monitoring traffic in dom0 to infer whether an app is backlogged by using existing techniques (see Section 2), perhaps intermittently or b) designing shaping schemes that incentivize tenants to declare real sendbuffer information. A starting point for this work could be the Re-Feedback proposal by Briscoe et al. [4]. Which technique should we pursue?

2. Tenant incentives for deployment. Cloud providers provide stock images of popular operating systems, and many tenants use these as basis for their virtual machines. The cloud provider can easily create versions of these images that implement sendbuffer advertising, and it can entice users to them by offering lower prices or higher network speeds. We believe that the provider should be able to offset these costs by the profit it makes from running its network more efficiently. Is this a reasonable deployment strategy?

3. Minimizing flow completion times. Novel datacenter network architectures such as pFabric [3], D3 [22] or PDQ [11] propose changing both the transport protocol and the network to reduce the completion time of flows, Sendbuffer advertisement can be seen as a minimal change to the transport that would allow similar behaviour, as long as switches prioritize traffic with lower sendbuffer utilization (e.g. with Openflow). It would be interesting to discuss the pros and cons of these solutions.

Acknowledgements

This work was partly funded by SSICLOPS, a project funded by the European Comission under its Horizon 2020 programme (contract number 644866).

References

[1] AL-FARES, M., LOUKISSAS, A., AND VAHDAT, A. A scalable, commodity data center network architecture. In *Proc. SIGCOMM 2008*.

[2] AL-FARES, M., RADHAKRISHNAN, S., RAGHAVAN, B., HUANG, N., AND VAHDAT, A. Hedera: Dynamic flow scheduling for data center networks. In *Proc. Usenix NSDI 2010*.

[3] ALIZADEH, M., YANG, S., SHARIF, M., KATTI, S., MCKEOWN, N., PRABHAKAR, B., AND SHENKER, S. pfabric: Minimal near-optimal datacenter transport. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM* (New York, NY, USA, 2013), SIGCOMM '13, ACM, pp. 435–446.

[4] BRISCOE, B., JACQUET, A., CAIRANO-GILFEDDER, C. D., SALVATORI, A., SOPPERA, A., AND KOYABE, M. Policing congestion response in an internetwork using re-feedback. *Proc. ACM SIGCOMM'05, Computer Communication Review 35*, 4 (Aug. 2005), 277–288.

[5] CHOWDHURY, M., ZHONG, Y., AND STOICA, I. Efficient coflow scheduling with varys. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (New York, NY, USA, 2014), SIGCOMM '14, ACM, pp. 443–454.

[6] CURTIS, A., KIM, W., AND YALAGANDULA, P. Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection. In *INFOCOM, 2011 Proceedings IEEE* (April 2011), pp. 1629–1637.

[7] CURTIS, A. R., MOGUL, J. C., TOURRILHES, J., YALAGANDULA, P., SHARMA, P., AND BANERJEE, S. Devoflow: scaling flow management for high-performance networks. In *Proceedings of the ACM SIGCOMM 2011 conference* (New York, NY, USA, 2011), SIGCOMM '11, ACM, pp. 254–265.

[8] FARRINGTON, N., PORTER, G., RADHAKRISHNAN, S., BAZZAZ, H. H., SUBRAMANYA, V., FAINMAN, Y., PAPEN, G., AND VAHDAT, A. Helios: A hybrid electrical/optical switch architecture for modular data centers. In *Proceedings of the ACM SIGCOMM 2010 Conference* (New York, NY, USA, 2010), SIGCOMM '10, ACM, pp. 339–350.

[9] GREENBERG EL AL., A. VL2: a scalable and flexible data center network. In *Proc. ACM Sigcomm 2009*.

[10] HALPERIN, D., KANDULA, S., PADHYE, J., BAHL, P., AND WETHERALL, D. Augmenting data center networks with multi-gigabit wireless links. In *Proceedings of the ACM SIGCOMM 2011 Conference* (New York, NY, USA, 2011), SIGCOMM '11, ACM, pp. 38–49.

[11] HONG, C.-Y., CAESAR, M., AND GODFREY, P. B. Finishing flows quickly with preemptive scheduling. *SIGCOMM Comput. Commun. Rev. 42*, 4 (Aug. 2012), 127–138.

[12] HONG, C.-Y., KANDULA, S., MAHAJAN, R., ZHANG, M., GILL, V., NANDURI, M., AND WATTENHOFER, R. Achieving high utilization with software-driven wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM* (New York, NY, USA, 2013), SIGCOMM '13, ACM, pp. 15–26.

[13] JAIN, S., KUMAR, A., MANDAL, S., ONG, J., POUTIEVSKI, L., SINGH, A., VENKATA, S., WANDERER, J., ZHOU, J., ZHU, M., ZOLLA, J., HÖLZLE, U., STUART, S., AND VAHDAT, A. B4: Experience with a globally-deployed software defined wan.

In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM* (New York, NY, USA, 2013), SIGCOMM '13, ACM, pp. 3–14.

[14] JAISWAL, S., IANNACCONE, G., DIOT, C., KUROSE, J., AND TOWSLEY, D. Inferring tcp connection characteristics through passive measurements. In *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies* (March 2004), vol. 3, pp. 1582–1592 vol.3.

[15] KATABI, D., HANDLEY, M., AND ROHRS, C. Congestion control for high bandwidth-delay product networks. In *Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (New York, NY, USA, 2002), SIGCOMM '02, ACM, pp. 89–102.

[16] RAICIU, C., BARRE, S., PLUNTKE, C., GREENHALGH, A., WISCHIK, D., AND HANDLEY, M. Improving datacenter performance and robustness with Multipath TCP. In *Proc. ACM SIGCOMM 2011*.

[17] RAICIU, C., IONESCU, M., AND NICULESCU, D. Opening up black box networks with cloudtalk. In *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Ccomputing* (Berkeley, CA, USA, 2012), HotCloud'12, USENIX Association, pp. 6–6.

[18] RAICIU, C., PAASCH, C., BARRE, S., FORD, A., HONDA, M., DUCHENE, F., BONAVENTURE, O., AND HANDLEY, M. How hard can it be? designing and implementing a deployable multipath tcp. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2012), NSDI'12, USENIX Association, pp. 29–29.

[19] SUN, P., YU, M., FREEDMAN, M., REXFORD, J., AND WALKER, D. Hone: Joint host-network traffic management in software-defined networks. *Journal of Network and Systems Management 23*, 2 (2015), 374–399.

[20] VASUDEVAN, V., PHANISHAYEE, A., SHAH, H., KREVAT, E., ANDERSEN, D. G., GANGER, G. R., GIBSON, G. A., AND MUELLER, B. Safe and effective fine-grained TCP retransmissions for datacenter communication. *SIGCOMM Comput. Commun. Rev. 39*, 4 (Aug. 2009), 303–314.

[21] WANG, G., ANDERSEN, D. G., KAMINSKY, M., PAPAGIANNAKI, K., NG, T. E., KOZUCH, M., AND RYAN, M. c-through: Part-time optics in data centers. In *Proceedings of the ACM SIGCOMM 2010 Conference* (New York, NY, USA, 2010), SIGCOMM '10, ACM, pp. 327–338.

[22] WILSON, C., BALLANI, H., KARAGIANNIS, T., AND ROWTRON, A. Better never than late: Meeting deadlines in datacenter networks. In *Proceedings of the ACM SIGCOMM 2011 Conference* (New York, NY, USA, 2011), SIGCOMM '11, ACM, pp. 50–61.

[23] WU, H., FENG, Z., GUO, C., AND ZHANG, Y. ICTCP: Incast congestion control for TCP in data center networks. Co-NEXT '10, ACM, pp. 13:1–13:12.

[24] ZHOU, X., ZHANG, Z., ZHU, Y., LI, Y., KUMAR, S., VAHDAT, A., ZHAO, B. Y., AND ZHENG, H. Mirror mirror on the ceiling: Flexible wireless links for data centers. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (New York, NY, USA, 2012), SIGCOMM '12, ACM, pp. 443–454.