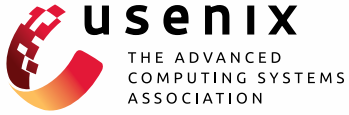


Accurate Latency-based Congestion Feedback for Datacenters

Changhyun Lee and Chunjong Park, ***Korea Advanced Institute of Science and Technology (KAIST);*** **Keon Jang,** ***Intel Labs;*** **Sue Moon and Dongsu Han,** ***Korea Advanced Institute of Science and Technology (KAIST)***

https://www.usenix.org/conference/atc15/technical-session/presentation/lee-changhyun

This paper is included in the Proceedings of the 2015 USENIX Annual Technical Conference (USENIC ATC '15).

July 8–10, 2015 • Santa Clara, CA, USA

ISBN 978-1-931971-225

Open access to the Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC '15) is sponsored by USENIX.

Accurate Latency-based Congestion Feedback for Datacenters

Changhyun Lee Chunjong Park Keon Jang† Sue Moon Dongsu Han
KAIST †Intel Labs

Abstract

The nature of congestion feedback largely governs the behavior of congestion control. In datacenter networks, where RTTs are in hundreds of microseconds, accurate feedback is crucial to achieve both high utilization and low queueing delay. Proposals for datacenter congestion control predominantly leverage ECN or even explicit in-network feedback (e.g., RCP-type feedback) to minimize the queuing delay. In this work we explore latency-based feedback as an alternative and show its advantages over ECN. Against the common belief that such implicit feedback is noisy and inaccurate, we demonstrate that latency-based implicit feedback is accurate enough to signal a single packet's queuing delay in 10 Gbps networks.

DX enables accurate queuing delay measurements whose error falls within 1.98 and 0.53 microseconds using software-based and hardware-based latency measurements, respectively. This enables us to design a new congestion control algorithm that performs fine-grained control to adjust the congestion window just enough to achieve very low queuing delay while attaining full utilization. Our extensive evaluation shows that 1) the latency measurement accurately reflects the one-way queuing delay in single packet level; 2) the latency feedback can be used to perform practical and fine-grained congestion control in high-speed datacenter networks; and 3) DX outperforms DCTCP with 5.33x smaller median queueing delay at 1 Gbps and 1.57x at 10 Gbps.

1 Introduction

The quality of network congestion control fundamentally depends on the accuracy and granularity of congestion feedback. For the most part, the history of congestion control has largely been about identifying the "right" form of congestion feedback. From packet loss and explicit congestion notification (ECN) to explicit in-network feedback [1, 2], the pursuit for accurate and fine-grained feedback has been central tenet in designing new congestion control algorithms. Novel forms of congestion feedback have enabled innovative congestion control behaviors that formed the basis of a number of flexible and efficient congestion control algorithms [3, 4], as the requirements for congestion control diversified [5].

With the advent of datacenter networking, identifying and leveraging more accurate and fine-grained feedback mechanisms have become even more crucial [6]. Round trip times (RTTs), which represent the interval of the control loop, are few hundreds of microseconds, where as TCP is designed to work in the wide area network (WAN) with hundreds of milliseconds of RTTs. Prevalence of latency-sensitive flows in datacenters (e.g., Partition/Aggregate workloads) requires low latency while the end-to-end latency is dominated by in-network queuing delay [6]. As a result, proposals for datacenter congestion control predominantly leverage ECN (e.g., DCTCP [6] and HULL [7]) or explicit in-network feedback (e.g., RCP-type feedback [2]), to minimize the queuing delay and the flow completion times.

This paper takes a relatively unexplored path of identifying a better form of feedback for datacenter networks. In particular, this paper explores the prospect of using network latency as congestion feedback in the datacenter environment. We believe latency can be a good form of congestion feedback in datacenters for a number of reasons: (i) by definition, it includes all queuing delay throughout the network, and hence is a good indicator for congestion; (ii) a datacenter is typically owned by a single entity who can enforce all end hosts to use the same latency-based protocol, effectively removing potential source of errors originating from uncontrolled traffic; and (iii) finally, latency-based feedback does not require any switch support.

Although latency-based feedback has been previously explored in WAN [8, 9], the datacenter environment is very different, posing unique requirements that are difficult to address. Datacenters have much higher bandwidth (10 Gbps to even 40 Gbps) at the end host and very low latency (few hundreds of microseconds) in the network.

This makes it difficult to measure the queuing delay of individual packets for a number of reasons: (i) I/O batching at the end host, which is essential for high throughput, introduces large measurement error (§2). (ii) Measuring queuing delay requires high precision because a single MSS packet introduces only 0.3 (1.2) microseconds of queuing delay in 40*GbE* (10*GbE*) networks. As a result, the common belief is that latency measurement might be too noisy to serve as reliable congestion feedback [6, 10].

On the contrary, we argue that it is possible to accurately measure the queuing delay at the end-host, so that even a single packet queuing delay is detectable. Realizing this requires solving several design and implementation challenges. First, even with very accurate hardware measurement, bursty I/O (e.g., DMA bursts) leads to inaccurate delay measurements. Second, ACK packets on the reverse path may be queued behind data packets and add noise to the latency measurement. To address these issues, we leverage a combination of recent advances in software low latency packet processing [11, 12] and hardware technology [13] that allows us to measure queuing delay accurately.

Such accurate delay measurements enable a more fine-grained control loop for datacenter congestion control. In particular, we envision a fine-grained feedback control loop achieves near zero-queuing with high utilization. Translating latency into feedback control to achieve high-utilization and low queuing is non-trivial. We present DX, a latency based congestion control that addresses these challenges. DX performs window adaptation to achieve low queuing delay (as low as that of HULL [7] and 6.6 times smaller than DCTCP), while achieving 99.9% utilization. Moreover it provides advantages over recent works in that it does not require any switch modifications.

To summarize, our contributions in this paper are the followings: (i) novel techniques to accurately measure in-network queuing delay based on end-to-end latency measurements; (ii) a congestion control logic that exploits latency-based feedback to achieve just a few packets of queuing delay and high utilization without any form of in-network support; and (iii) a prototype that demonstrates the feasibility and its benefits in our testbed.

2 Accurate queuing delay measurement

Latency measurement can be inaccurate for many reasons including variability in end-host stack latency, NIC queuing delay, and I/O batching. In this section, we describe several techniques to eliminate such sources of errors. Our goal is to achieve a level of accuracy that can distinguish even a single MSS packet queuing at 10 Gbps, which is 1.2 μs. This is necessary to target near zero queuing as congestion control should be able to back off even when a single packet is queued.

Before we introduce our solutions to each source of error, we first show how noisy the latency measurement is without any care. Figure 1 shows the round trip time measured by the sender's kernel when saturating a 10 Gbps link; we generate TCP traffic using *iperf* [14] on Linux kernel. the sender and the receiver are connected back to back, so no queueing is expected in the network. Our measurement shows that the round-trip time varies from 23 μs to 733 μs, which potentially gives up to 591 packets of error. The middle 50% of RTT samples still exhibit wide range of errors of 111 μs that corresponds to 93 packets. These errors are an order of magnitude larger than our target latency error, 1.2 μs.

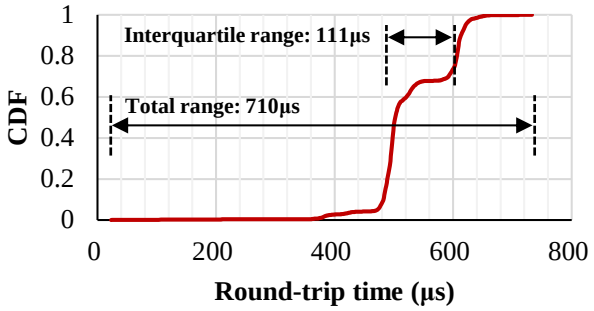


Figure 1: Round-trip time measured in kernel

Source of error	Elimination technique
End-host network stack ($\sim 100\mu s$)	Exclude host stack delay
I/O batching & DMA bursts (tens of μs)	Burst reduction & error calibration
Reverse path queuing ($\sim 100\mu s$)	Use difference in one-way latency
Clock drift (long term effect)	Frequent base delay update

Table 1: Sources of errors in latency measurement and our techniques for mitigation.

Table 1 shows four sources of measurement errors and their magnitude. We eliminate each of them to achieve our target accuracy ($\sim$1μsec).

Removing host stack delay: End-host network stack latency variation is over an order of magnitude larger than our target accuracy. Our measurement shows about 80μs standard deviation, when the RTT is measured in the Linux kernel's TCP stack. Thus, it is crucial to eliminate the host processing delay in both a sender and a receiver.

For software timestamping, our implementation choice eliminates the end host stack delay at the sender as we timestamp packets right before the TX, and right after

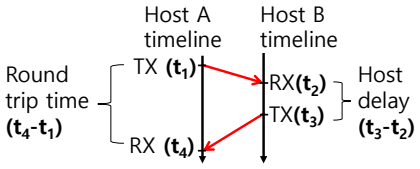


Figure 2: Timeline of timestamp measurement points

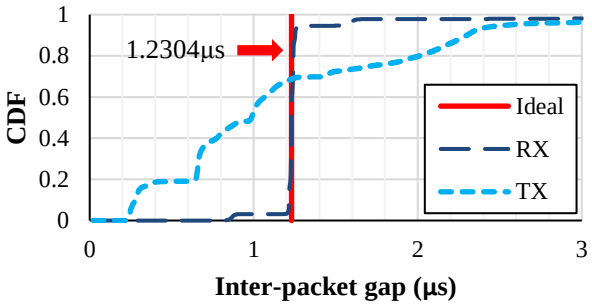


Figure 3: H/W timestamped inter-packet gap at 10 Gbps

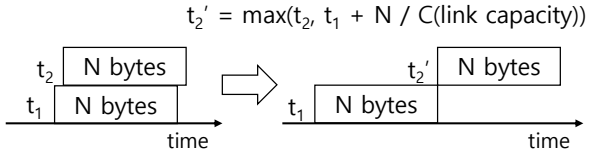


Figure 4: Example delay calibration for bursty packet reception

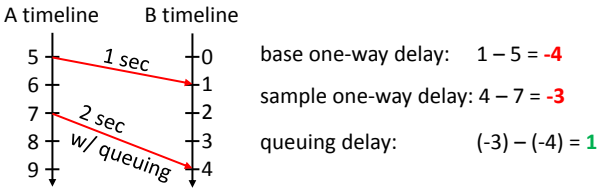


Figure 5: One-way queuing delay without time synchronization

the RX on top of DPDK [12]. Hardware timestamping innately removes such delay.

Now, we need to deal with the end-host stack delay at the receiver. Figure 2 shows how DX timestamps packets when a host sends one data packet and receives back an ACK packet. To remove the end host stack delay from the receiver, we simply subtract the $t_3 - t_2$ from $t_4 - t_1$. The timestamp values are stored and delivered in the option fields of the TCP header.

Burst reduction: TCP stack is known to transmit packets in a burst. The amount of burst is affected by the window size and TCP Segmentation Offloading (TSO), and ranges up to 64 KB. Burst packets affect timestamping because all packets in a TX burst get the almost the same timestamp, and yet they are received by one by one at the receiver. This results in an error as large as $50\mu s$.

To eliminate packet bursts, we use a software token bucket to pace the traffic at the link capacity. The token bucket is a packet queue and drained by polling in SoftNIC [15].

At each poll, the number of packets drained is calculated based on the link rate and the elapsed time from the last poll. The upper bound is 10 packets, which is enough to saturate 99.99% of the link capacity even in 10 Gbps networks. We note that our token bucket is different from TCP pacing or the pacer in HULL [7] where each and every packet is paced at the target rate; our token bucket is simply implemented with very small overhead. In addition, we keep a separate queue for each flow to prevent the latency increase from other flows' queue build-ups.

Error calibration: Even after the burst reduction, packets can be still batched for TX as well as RX. Interestingly, we find that even hardware timestamping is subject to the noise introduced by packet bursts due to its implementation. We run a simple experiment where sending a traffic near line rate 9.5 Gbps from a sender to a receiver connected back to back. We measure the inter packet gap using hardware timestamps, and plot the results in Figure 3. Ideally, all packets should be spaced at $1.23\mu s$. As shown in the figure, a large portion of the packet gaps of TX and RX falls below $1.23\mu s$. The packet gaps of TX are more variable than that of RX, as it is directly affected by I/O batching, while RX DMA is triggered when a packet is received by the NIC. The noise in the H/W is caused by the fact that the NIC timestamps packets when it completes the DMA, rather than timestamping them when the packets are sent or received on the wire. We believe this is not a fundamental problem, and H/W timestamping accuracy can be further improved by minor changes in implementation.

In this paper, we employ simple heuristics to reduce the noise by accounting for burst transmission in software. Suppose two packets are received or transmitted in the same batch as in Figure 4. If the packets are spaced with timestamps whose interval is smaller than what the link capacity allows, we correct the timestamp of the latter packet to be at least transmission delay away from the former packet's timestamp. In our measurement at $10Gbps$, 68% of the TX timestamp gaps need such calibration.

One-way queuing delay: So far, we have described techniques to accurately measure RTT. However, RTT includes the delay on the reverse path, which is another source of noise for determining queuing on the forward path. A simple solution to this is measuring one-way delay which requires clock synchronization between two hosts. PTP (Precision Time Protocol) enables clock synchronization with sub-microseconds [16]. However it requires hardware support and possibly switch support to

remove errors from queuing delay. It also requires periodic synchronization to compensate clock drifts. Since we are targeting a microsecond level of accuracy, even a short term drift could affect the queuing delay measurement. For these reasons, we choose not to rely on clock synchronization.

Our intuition is that unlike one-way delay, queuing delay can be measured simply by subtracting the baseline delay (skewed one-way delay with zero queuing) from the sample one-way delay even if the clocks are not synchronized. For example, suppose a clock difference of 5 seconds, as depicted in Figure 5. When we measure one-way delay from A to B, which takes one second propagation delay (no queuing), the one-way delay measured would be -4 seconds instead of one second. When we measure another sample where it takes 2 seconds due to queuing delay, it would result in -3 seconds. By subtracting -4 from -3, we get one second queuing delay.

Now, there are two remaining issues. First is obtaining accurate baseline delay, and second is dealing with clock drifts. The base line can be obtained by picking the minimum one-way delay amongst many samples. The frequency of zero queuing being measured depends on the congestion control algorithm behavior. Since we target near zero-queuing, we observe this every few RTTs.

Handling clock drift: A standard clock drifts only 40 nsecs per msec [17]. This means that the relative error between two measurements (e.g., base one-way delay and sample one-way delay) taken from two clocks during a millisecond can only contain tens of nanoseconds of error. Thus, we make sure that base one-way delay is updated frequently (every few round trip times). One last caveat with updating base one-way delay is that clock drift differences can cause one-way delay measurements to continuously increase or decrease. If we simply take minimum base one-way delay, it causes one side to update its base one-way delay continuously, while the other side never updates the base delay because its measurement continuously increases. As a workaround, we update the base one-way delay when the RTT measurement hits the new minimum or re-observe the current minimum; RTT measurements are not affected by clock drift, and minimum RTT implies no queueing in the network. This event happens frequently enough in DX, and it ensures that clock drifts do not cause problems.

3 DX: Latency-based Congestion Control

The ability to accurately measure the switch queue length from end-hosts enables new opportunities. In particular, DX leverages its power for finer-grained congestion control.

We present a congestion control algorithm for datacenters that targets near zero queueing delay based on implicit feedback, without any form of in-network support. Because latency feedback signals the amount of excessive packets in the network, it allows senders to calculate the maximum number of packets to drain from the network while achieving full utilization. This section presents the basic mechanisms and design of our new congestion control algorithm, DX. Our target deployment environment is datacenters, and we assume that all traffic congestion is controlled by DX, similar to the previous work [3, 5–7, 10].

DX is a window-based congestion control algorithm. DX's congestion avoidance follows the popular Additive Increase Multiplicative Decrease (AIMD) rule. The key difference from TCP (e.g., TCP Reno) is its congestion avoidance algorithm. DX uses the queueing delay to make a decision on whether to increase or decrease congestion window in the next round at every RTT. Zero queueing delay indicates that there is still more room for packets in the network, so the window size is increased by one at a time. On the other hand, any positive queueing delay means that a sender must decrease the window.

DX updates the window size using the formula below:

$$new\ CWND = \begin{cases} CWND + 1, & \text{if } Q = 0 \\ CWND \times (1 - \frac{Q}{V}), & \text{if } Q > 0, \end{cases} \quad (1)$$

where Q represents the latency feedback, that is, the average queueing delay in the current window, and V is a self-updated coefficient of which role is critical in our congestion control.

When $Q > 0$, DX decreases the window proportional to the current queueing delay. The amount to decrease should be just enough to drain the currently queued packets not to affect utilization. An aggressive decrease in the congestion window will cause the network utilization to drop below 100%. For DX, the exact amount depends on the number of flows sharing the bottleneck because the aggregate sending rate of these flows should decrease to drain the queue. V is the coefficient that accounts for the number of competing flows. We drive the value of V using the analysis below.

We denote the link capacity (packets / sec) as C, the base RTT as R, single-packet transmission delay as D, the number of flows as N, and the window size and the queueing delay of flow k at time t as $W_k{}^{(t)}$ and $Q_k{}^{(t)}$, respectively. Without loss of generality, we assume at time t the bottleneck link fully utilized and the queue size is zero. We also assume that their behaviors are synchronized to derive a closed-form analysis and verify the results using simulations and testbed experiments. At time t, because the link is fully utilized and the queuing delay is zero, the sum of the window size equals to the bandwidth delay product $C \cdot R$:

$$\sum_{k=1}^{N} {W_k}^{(t)} = C \cdot R \tag{2}$$

Since none of the N flows experiences congestion, they all increase their window size by one at time $t+1$:

$$\sum_{k=1}^{N} {W_k}^{(t+1)} = C \cdot R + N \tag{3}$$

Now all the senders observe a positive queueing delay, and they respond by decreasing the window size using the multiplicative factor, $1 - Q/V$, as in (1). As a result, at time $t+2$, we expect fewer packets in the network; we want just enough packets to fully saturate the link and achieve zero queuing delay in the next round. We calculate the total number of packets in the network (in both the link and the queues) at time $t+2$ from the sum of window size of all the flows.

$$\sum_{k=1}^{N} {W_k}^{(t+2)} = \sum_{k=1}^{N} {W_k}^{(t+1)} (1 - \frac{{Q_k}^{(t+1)}}{V}) \tag{4}$$

Assuming every flow experiences maximum queueing delay $N \cdot D$ in the worst case, we get:

$$\begin{aligned} \sum_{k=1}^{N} {W_k}^{(t+2)} &= \sum_{k=1}^{N} {W_k}^{(t+1)} (1 - \frac{N \cdot D}{V}) \\ &= (C \cdot R + N)(1 - \frac{N \cdot D}{V}) \end{aligned} \tag{5}$$

We want total number of in-flight packets at time $t+2$ to equal to the bandwidth delay product:

$$(C \cdot R + N)(1 - \frac{N \cdot D}{V}) = C \cdot R \tag{6}$$

Solving for V results in:

$$V = \frac{N \cdot D}{(1 - \frac{C \cdot R}{C \cdot R + N})} \tag{7}$$

Among the variables required to calculate V, the only unknown is N, which is the number of concurrent flows. The number of flows can be estimated from the sender's own window size because DX achieves fair-share throughput at steady state. For notational convenience, we denote ${W_k}^{(t+1)}$ as W^* and rewrite (3) as:

$$\sum_{k=1}^{N} {W_k}^{(t+1)} = N \times W^* = C \cdot R + N \Leftrightarrow N = \frac{C \cdot R}{W^* - 1}$$

Using (5) and replacing D, single-packet transmission delay, with $(1/C)$, we get:

$$V = \frac{R \cdot W^*}{W^* - 1} \tag{8}$$

In calculating V, the sender only needs to know the based RTT, R, and the previous window size W^*. No additional measurement is required. We do not need to rely on external configuration or parameter settings either, unlike the ECN-based approaches. Even if the link capacity in the network varies across links, it does not affect our calculation of V.

4 Implementation

We have implemented DX in two parts: latency measurement in DPDK-based NIC driver and latency-based congestion control in the Linux's TCP stack. This separation provides a few advantages: (i) it measures latency more accurately than doing so in the Linux Kernel; (ii) legacy applications can take advantage of DX without modification; and (iii) it separates the latency measurement from the TCP stack, and hides the differences between hardware implementations, such as timestamp clock frequencies or timestamping mechanisms. We present the implementation of software- and hardware-based latency measurements and modifications to the kernel TCP stack to support latency feedback.

4.1 Timestamping and delay calculation

We measure four timestamp values as shown in section 2 Figure 2: t_1 and t_2 are the transmission and reception time of a data packet, and t_3 and t_4 are the transmission and reception time of a corresponding ACK packet.

Software timestamping: To eliminate host processing delay, we perform TX timestamping right before pushing packets to the NIC, and RX timestamping right after the packets are received, at the DPDK-based device driver. We use `rdtsc` to get CPU cycles and transform this into nanoseconds timescales. We correct timestamps using techniques described in §2. All four timestamps must be delivered to the sender to calculate the one-way delay and the base RTT. We use TCP's option fields to relay t_1, t_2, and t_3 (§4.2).

To calculate one-way delay, the DX receiver stores a mapping from expected ACK number to t_1 and t_2 when it receives a data packet. It then puts them in the corresponding ACK along with the ACK's transmission time (t_3). The memory overhead is proportional to the arrived data of which the corresponding ACK has not been sent yet. The memory overhead is negligible as it requires store 8 bytes per in-flight packet. In the presence of delayed ACK, not all timestamps are delivered back to the sender, and some of them are discarded.

Hardware timestamping: We have implemented hardware-based timestamping on Mellanox ConnectX-3 using a DPDK-ported driver. Although the hardware supports RX/TX timestamping for all packets, its driver did not support TX timestaming. We have modified the driver to timestamp all RX/TX packets.

The NIC hardware delivers timestamps to the driver by putting the timestamps in the ring descriptor when it completes DMA. This causes an issue with the previous logic to carry t_1 in the original data packet. To resolve this, we store mapping of expected ACK number to the t_1 at the sender, and retrieve this when ACK is received.

LRO handling: Large Receive Offload (LRO) is a widely used technique for reducing CPU overhead on the receiver side. It aggregates received TCP data packets into a large single TCP packet and passes to the kernel. It is crucial to achieve 10 Gbps or beyond in today's Linux TCP stack. This affects DX in two ways. First, it makes the TCP receiver generate fewer number of ACKs, which in turn reduces the number of t_3 and t_4 samples. Second, even though t_1 and t_2 are acquired before LRO bundling at the driver, we cannot deliver all of them back to the kernel TCP stack due to limited space in the TCP option header. To work around the problem, for each ACK that is processed, we scan through the previous t_1 and t_2 samples, and deliver average one-way delay with the sample count. In fact, instead of passing all timestamps to the TCP layer, we only passes one-way delay t_2 - t_1 and RTT $((t_4 - t_1) - (t_3 - t_2))$

Burst mitigation: As shown in § 2, burstiness from I/O batching incurs timestamping errors. To control burstiness, we implement a simple token bucket with burst size of MTU and rate set to link capacity. SoftNIC [15] does polling on the token bucket to draw packets and passes them to the timestamping module or the NIC. If the polling loop takes longer than the transmission time of a packet, the token bucket emits more than one packet, but limits the number of packets to keep up with link capacity.

4.2 Congestion control

We implement DX congestion control algorithm in the Linux 3.13.11 kernel. We add DX as a new TCP option that consumes 14 bytes of additional TCP header. The first 2 bytes are for the option number and the option length required by the TCP option parser. The remaining 12 bytes are divided into three 4 byte spaces and used for storing timestamps and/or an ACK number.

Most of modifications are made in the *tcp_ack()* function in TCP stack. This is triggered when an ACK packet is received. An ACK packet carries one-way delay and RTT in the header that are pre-calculated by the DPDK-based device driver. For each round trip time, the received delay samples are averaged and used for new CWND calculation. The current implementation takes the average one-way delay observed during the last round trip.

Practical considerations: In real-world networks, a transient increase in queueing delay Q does not always mean network congestion. Reacting to wrong congestion signals results in low link utilization. There are two sources of error: measurement noise and instant queueing due to packet bursts. Although we have shown that our latency measurement has a low standard deviation up to about a microsecond, it can still trigger undesirable window reduction as DX reacts to a positive queueing delay whether large or small. On the other hand, instant queueing can happen with even very small number of packets. For example, if two packets arrive at the switch at the exactly same moment, one of them will be served after the first packet's transmission delay, hence positive queueing delay.

To tackle such practical issues, we come up with two simple techniques. First, we use headroom when determining congestion; DX does not decrease window size when $Q < headroom$.

Second, to be robust against transient increase in delay measurements, we use the average queueing delay during an RTT period. In an ideal network without packet bursts, the maximum queueing delay is a good indication of excess packets. In real networks, however, taking the maximum is easily affected by instant queueing. Taking the minimum removes the burstiness most effectively, but it detects congestion only when all the packets in the window experience positive queueing delay. Hence we choose the average to balance them out.

Note that DCTCP, a previous ECN-based solution, also suffers from bursty instant queueing and requires higher ECN threshold in practice than theoretic calculation [6].

5 Evaluation

Throughout the evaluation, We answer three main questions:

- Can DX obtain the accuracy of a single packet's queuing delay in high-speed networks?
- Can DX achieve minimal queuing delay while achieving high utilization?
- How does DX perform in large scale networks with realistic workloads?

By using testbed experiments, we show that our noise reduction techniques are effective and queuing delay can be measured with an accuracy of a single MSS packet at 10 Gbps. We evaluate DX against DCTCP and verify that it reduces queuing in the switch up to five times.

Next, we use *ns-2* packet level simulation to conduct more detailed analysis and evaluate DX in large-scale with realistic workload. First, we verify the DX's effectiveness by looking at queuing delay, utilization and fairness. We then quantify the impact of measurement errors on DX to evaluate its robustness. Finally, we perform large-scale evaluation to compare DX's overall performance against the state of the art: DCTCP [6] and HULL [7].

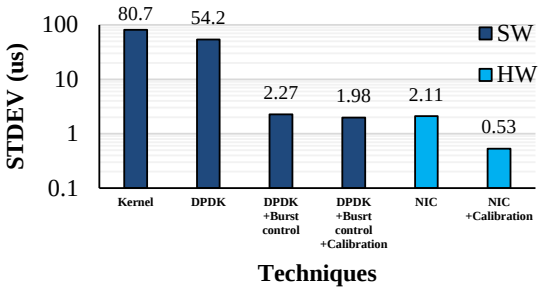
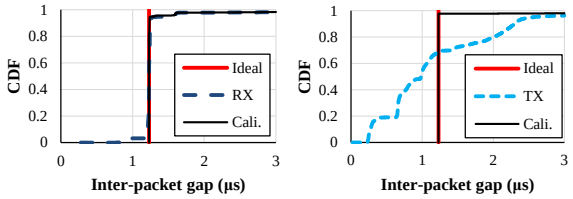


Figure 6: Improvements with noise reduction techniques



(a) RX Calibration (b) TX Calibration

Figure 7: Effect of calibration in H/W timestamped inter-packet gap at 10 Gbps

5.1 Accuracy of queuing delay in testbed

For testbed experiments, we use Intel 1 GbE/10 GbE NICs for software timestamping and Mellanox ConnectX-3 40 GbE NIC for hardware timestamping; the Mellanox NIC is used in 10 Gbps mode due to the lack of 40 GbE switches.

Effectiveness of noise reduction techniques: To quantify the benefit of each technique, we apply the techniques one by one and measure RTT using both software and hardware. Two machines are connected back to back, and we conduct RTT measurement at 10 Gbps link. We plot the standard deviation in Figure 6. Ideally, the RTT should remain unchanged since there is no network queueing delay. In software-based solution, we reduce the measurement error (presented as standard deviation) down to 1.98 μs by timestamping at DPDK and applying burst control and calibration. Among the techniques, burst control is the most effective, cutting down the error by 23.8 times. In hardware solution, simply timestamping at NIC achieves comparable noise with all techniques applied in the software solution. After inter-packet interval calibration, the noise drops further down to 0.53 μs, less than half of a single packet's queueing delay at 10 Gbps, which is within our target accuracy.

Calibration of H/W timestamping: We look further into how calibration affects the accuracy of hardware timestamping. Figure 7 shows the CDF of inter packet gap measurements before and after calibration for both RX and TX. The calibration effectively removes the inter packet gap samples smaller than link transmission delay

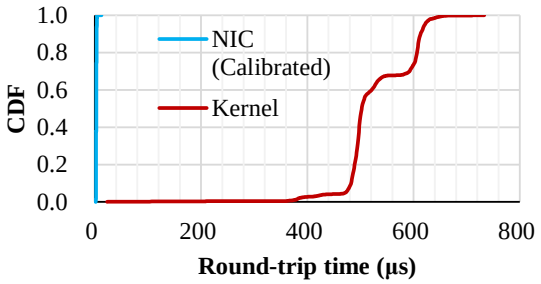
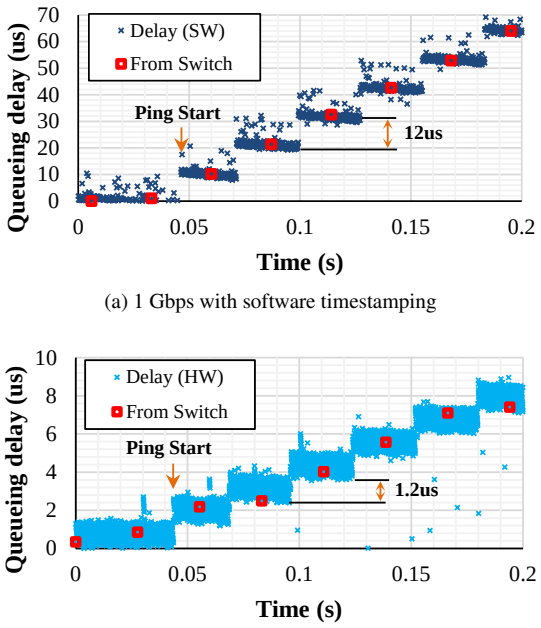


Figure 8: Improvement on RTT measurement error compared to kernel's



(a) 1 Gbps with software timestamping

(b) 10 Gbps with hardware timestamping

Figure 9: Accuracy of queuing delay measurement

which originally took up 68% for TX and 32% for RX.

Overall RTT measurement accuracy improvement: Now, we look at how much overall improvements we made on the accuracy of RTT measurement. We plot the CDF of RTT measurement for our technique using hardware and RTT measured in the Kernel in Figure 8. The total range of RTT has decreased by 62 times, from 710 μs to 11.38 μs. The standard deviation is improved from 80.7 μs to 0.53 μs by two orders of magnitude, and falls below a single packet queuing at 10 Gbps.

Verification of queuing delay: Now that we can measure RTT accurately, the remaining question is whether it leads to accurate queuing delay estimation. We conduct a controlled experiment where we have a full control over the queuing level. To create such scenario, we saturate a port in a switch by generating full throttle traffic from one host, and inject a MTU-sized ICMP packet to the same port at fixed interval from another host. This way, we

increase the queuing by a packet at fixed interval, and we measure the queuing statistics from the switch to verify our queuing delay measurement.

Figure 9 shows the time series of queuing delay measured by DX along with the ground truth queue occupancy measured at the switch (marked as red squares). We use software and hardware timestamping for 1 Gbps and 10 Gbps, respectively. Every time a new ping packet enters the network, the queueing delay increases by one MTU packet transmission delay: 12 μs at 1 Gbps and 1.2 μs at 10 Gbps. The queue length retrieved from the switch also matches our measurement result. The result at 10 Gbps seems noisier than at 1 Gbps due to the smaller transmission delay; note that the scale of Y-axis is different in two graphs.

Overall, we observe that our noise reduction techniques can effectively eliminate the sources of errors and result in accurate queuing delay measurement.

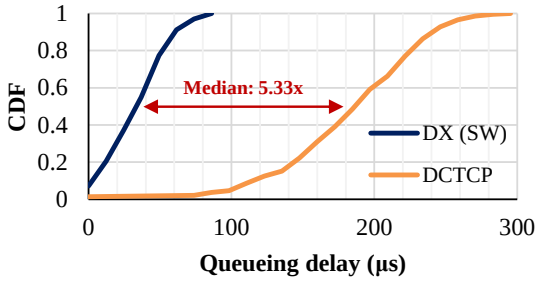
5.2 DX congestion control in testbed

Using the accurate queueing delay measurements, we run our DX prototype with three servers in our testbed; two nodes are senders and the other is a receiver. We use *iperf* [14] to generate TCP flows for 15 seconds. For comparison, we run DCTCP in the same topology. The ECN marking threshold for DCTCP is set to the recommended value of 20 at 1 Gbps and 65 at 10 Gbps [6]. During the experiment, the switch queue length is measured every 20 ms by reading the register values from the switch chipset. We first present the result at 1 Gbps bottleneck link in Figure 10a. In both protocols, two senders saturate the bottleneck link with fair-share throughput. The queue length is measured in bytes and converted into time.

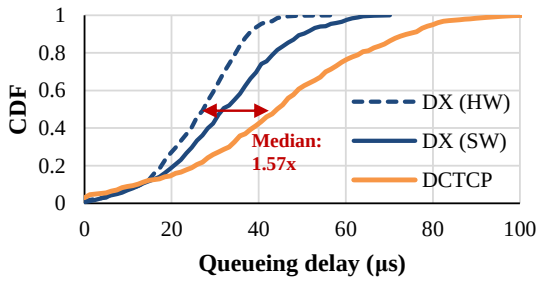
We observe that DX consistently reduces the switch queue length compared to that of DCTCP. The average queueing delay of DX, 37.8 μs, is 4.85 times smaller than that of DCTCP, 183.4 μs. DX shows 5.33x improvement in median queue length over DCTCP (3 packets for DX and 16 packets for DCTCP). DCTCP's maximum queue length goes up to 24 packets, while DX peaks at 8 packets.

We run the same experiment with 10 Gbps bottleneck. For 10 Gbps, we additionally run DX with hardware timestamp using Mellanox ConnectX-3 NIC. Figure 10b shows the result. DX (HW) denotes hardware timestamping, and DX (SW) denotes software timestamping. DX (HW) decreases the average queue length by 1.67 times compared to DCTCP, from 43.4 μs to 26.0 μs. DX (SW) achieves 31.8 μs of average queuing delay. The result also shows that DX effectively reduces the 99th-percentile queue length by a factor of 2 with hardware timestamping; DX (HW) and DX (SW) achieve 52 packets and 38 packets respectively while DCTCP achieves 78 packets.

To summarize, latency feedback is effective in maintaining low queue occupancy than ECN feedback, while



(a) 1Gbps bottleneck



(b) 10Gbps bottleneck

Figure 10: Queue length comparison of DX against DCTCP in Testbed

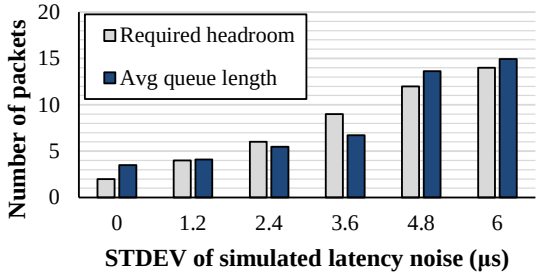


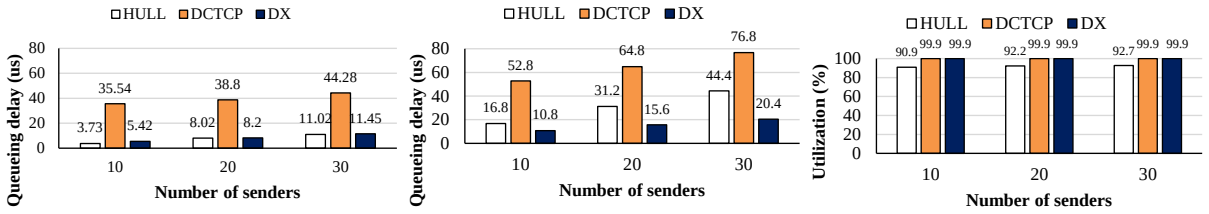
Figure 11: Impact of latency noise to headroom and queue length

saturating the link. DX achieves 4.85 times smaller average queue size at 1 Gbps and 1.67 times at 10 Gbps compared to DCTCP. DX reacts to congestion much earlier than DCTCP and reduces the congestion window to the right amount to minimize the queue length while achieving full utilization. DX achieves the lowest queueing delay among existing end-to-end congestion controls with implicit feedback that do not require any switch modifications,

In the next section, we also show that DX is even comparable to HULL, a solution that requires in-network support and switch modification.

5.3 Large-scale simulation

In this section, we run DX, DCTCP, and HULL in simulation to observe the performance in larger-scale environment.



(a) Queueing delay (average) (b) Queueing delay (99th percentile) (c) Utilization

Figure 12: Queueing delay and utilization of HULL, DCTCP, and DX

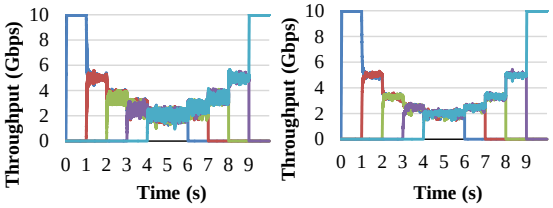
First, we run *ns-2* simulation using a dumbbell topology with 10 Gbps link capacity. Before the main simulation, we evaluate the impact of latency noise to the headroom size and average queue length. We generate latency noise using normal distribution with varying standard deviation. The noise level is multiples of 1.2 μs, single packet's transmission delay. As the simulated noise level increases, we need more headroom for full link utilization. Figure 11 shows the required headroom for full utilization and the resulting queue length in average. We observe that even if the noise becomes as large as 6 μs, DX can sustain noise error by simply increasing headroom size followed by the same amount of increase in queue length. Note that the standard deviation of our hardware timestamping is only 0.53 μs.

For scalability test, we now vary the number of simultaneous flows from 10 to 30 as queuing delay and utilization are correlated with it; the number of senders has a direct impact on queueing delay as shown in DCTCP [6]. We measure the queuing delay and utilization, and plot them in Figure 12.

Queueing delay: Many distributed applications with short flows are sensitive to the tail latency as the slowest flow that belongs to a task determines the completion time of the task [18]. Hence, we look at the 99th percentile queuing delay as well as the average queueing delay. On average, DX achieves 6.6x smaller queueing delay than DCTCP with ten senders, and slightly higher queuing delay than HULL. At 99th percentile, DX even outperforms HULL by 1.6x to 2.2x. The reason that DX achieves such low queuing is because of the immediate reaction to the queuing whereas both DCTCP and HULL uses weighted averaging for reducing congestion window size that takes multiple round trip times.

Utilization: DX achieves 99.9% of utilization which is comparable to DCTCP, but with much smaller queuing. HULL sacrifices utilization to reduce the queuing delay achieving about 90% of the bottleneck link capacity. We note that low queueing delay of DX does not sacrifice the utilization.

Fairness and throughput stability: To evaluate the throughput fairness, we generate 5 identical flows in the



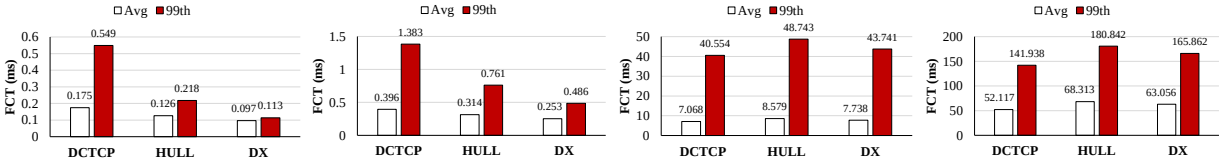
(a) DCTCP (b) DX

Figure 13: Fairness of five flows with DCTCP and DX

10 Gbps link one by one with 1 second interval and stop each flow after 5 seconds of transfer. In Figure 13, we see that both protocols offer fair throughput to exiting flows at each moment. One interesting observation is that DX flows have more stable throughput than DCTCP flows. This implies that DX provides higher fairness than DCTCP in small time scale. We compute the standard deviation of throughput to quantify the stability; 268 Mbps for DCTCP and 122 Mbps for DX.

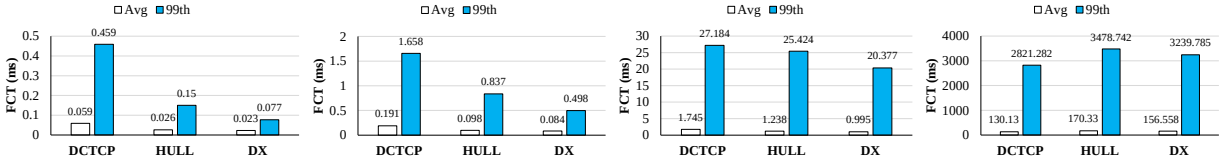
To understand the performance of DX in a large-scale data center environment, we perform simulations with realistic topology and traffic workload. The network consists of 192 servers and 56 switches that are connected as a 3-tier fat tree; there are 8 core switches, 16 aggregation switches, and 32 top-of-rack switches. All network links have 10 Gbps bandwidth, and the path selection is done by ECMP routing. The network topology we use is similar to that of HULL [7]. Once the simulation starts, the flow generator module selects a sender and a receiver randomly and starts a new flow. Each new flow is generated following Poisson process to produce 15% load at the edge. We run simulation until we have 100,000 flows started and finished. To test realistic workload, we choose flow size according to empirical workload reported from real-world data centers. We use two workload data: web search [6] and data mining [19].

Web search workload: The web search workload mostly contains small and medium-sized flows from a few KB to tens of MB; more than 95% of total bytes come from the flow smaller than 20MB, and the average flow size is 654KB [20]. In Figure 14, we present the flow comple-



(a) 0KB-10KB (b) 10KB-100KB (c) 100KB-10MB (d) 10MB-

Figure 14: Flow completion time of search workload



(a) 0KB-10KB (b) 10KB-100KB (c) 100KB-10MB (d) 10MB-

Figure 15: Flow completion time of data mining workload

tion time (FCT) in four flow-size groups: (0KB,10KB), [10KB,100KB), [100KB,10MB), and [10MB,∞).

For the flows smaller than 10KB, DX significantly reduces the 99th percentile FCT; it is 4.9x smaller than DCTCP and 1.9x smaller than HULL. DX also achieves minimal flow completion time in the 10KB-100KB group.

In larger flow size group, the performance of DX falls between DCTCP and HULL. DX achieves 7.7% lower average flow completion time compared to HULL and 20.9% higher than DCTCP for flows of size 10 MB and greater. This is because when ACK packets from other flows share the same bottleneck link, the queuing delay increases slightly. As a result, DX senders respond to the increased queuing delay. This is a side effect of targeting zero queueing. Because ACK packets are small and often piggy-backed on data packets we believe this is not a serious problem, but leave this as future work.

Data mining workload: The data mining workload is comprised of tiny and large-sized flows from hundreds of bytes to 1GB. The flow size is highly skewed that 80% of flows are smaller than 10KB [20] so 95% of bytes come from flows larger than 30MB; the average flow size is 7,452KB. The flow completion time of data mining workload is presented in Figure 15.

The performance improvement of DX is more outstanding for data minining workload than for search workload. In the three flow groups up to 10MB, DX flows finish early in every case. The biggest benefit comes from the smallest flow group as tail FCT is 6.0x smaller than DCTCP and 1.9x than HULL. For the largest flow group, DX suffers the same problem from the search workload but still shows shorter completion time than HULL's.

6 Discussion

NIC support for latency measurements: Current commodity NICs' support for timestamping is primarily for IEEE 1588 PTP, a hardware-based time synchronization protocol, designed to achieve sub-microsecond accuracy. While we leverage this functionality in DX, it is not perfectly suitable for our network latency measurements as explained in §2. In particular, it timestamps TX packets after completing DMA, and it does not support recording the TX time directly on the packets at the time of transmission. Although, our implementation works around these issues in software to reduce measurement errors, we believe changes in hardware will be more effective, especially for 10G/40G networks. If the hardware timestamps packets as it sends them out in the wire, the errors from NIC queueing and DMA bursts would be eliminated. Also, if it allows us to directly write timestamps on the packet header, this can shorten the feedback loop of DX by an RTT.

Deployment and co-existence with TCP: DX strictly targets datacenter networks for deployment. Datacenter environment favors DX deployment in that 1) it belongs to a single administration domain that can readily adopt a new protocol, and 2) network structure is more homogeneous and static than WAN, which helps latency measurement stability. As DX does not require any changes to the existing network switches, we can deploy DX with only end-host modification. Software-based solution can be deployed on existing machines, and hardware-based solution requires timestamping-enabled NICs. IEEE 1588 PTP-enabled NICs are already popular [21], and we envision timestamping-enabled NICs become more popular in the near future.

DX is specifically designed for handling only internal datacenter traffic, not external traffic to WAN. Separation between internal and external traffic is attainable by using load balancers and application proxies in existing datacenters [6]. We do not claim that DX can operate with conventional TCP sharing the same queue at network switches; a single TCP flow can cause a switch queue to overflow, which is directly against DX's goal. Our best resort to co-existing with TCP flows is to exploit priority queues at the switch and separate DX traffic from other TCP traffic. How to design such network efficiently is out of this paper's scope and we leave it as future work.

7 Related Work

Latency-based feedback in wide area network: There have been numerous proposals for network congestion control since the advent of the Internet. Although the majority of proposals use packet loss to detect network congestion, a large body of work has studied latency feedback. Latency-based TCP all agree on latency being more informative source of measuring congestion level, but the purpose and control mechanism is different in each protocol. TCP Vegas [8] is one of the earliest work and aims at achieving high throughput by avoiding loss. FAST TCP [9] is designed to quickly reach the fair-share throughput and uses latency for an equation parameter. TCP Nice [22] and TCP-LP [23] operate in low priority minimizing interference with other flows. So far, the latency-based approach has only been used in wide area network, and no protocol is known to target zero queueing delay.

ECN-based feedback in datacenter networks: Monitoring congestion level at the switch can help controlling the rate of TCP to minimize queuing. ECN marking in the TCP header has received much attention recently. DCTCP [6] uses a predefined threshold, and end-nodes then count the number of ECN marked packets to determine the degree of congestion and decrease the window size accordingly. HULL [7] is a similar to DCTCP, but sacrifices a small portion of the link capacity with phantom queue implemented at switches to detect congestion early and to achieve lower queueing delay than DCTCP. D^2TCP [24] also follows the same line of idea as DCTCP, and it uses gamma correction function to take into account each flow's deadline when adjusting the window size. As another variant of DCTCP, L^2DCT [25] considers flows' priority when reducing window size, and the priority is determined by the scheduling policy used in the network. ECN* [26] proposes dequeue marking for ECN to work effectively in datacenters. The aforementioned ECN marking approaches require modification of the TCP stack in end-node OS as well as minor parameter tunings at switches.

In-network feedback in datacenter networks: A few approaches have proposed to modify network switches in a way that TCP senders or middle switches can learn congestion status more quickly and accurately. D^3 [3] employs similar mechanism to RCP so that it can control flow rates to implement deadline based scheduling. DeTail [27] has implemented a new cross-layer network stack so that flows can avoid congested paths in the network, and PDQ [28] proposes distributed scheduling of flows that posses different priorities. These solutions are much harder to deploy than end-to-end solutions.

Flow scheduling in datacenter networks: Finally, we note that flow scheduling approaches, such as pFabric, PDQ, Varys, and PASE, also offer low flow completion times using prioritization and multiple queues. While some solutions intermix the congestion control and flow scheduling [29], we believe that congestion control and flow scheduling are largely orthogonal. For example, PASE adopts a DCTCP-like rate control scheme for lower priority queues [29] to ensure fairsharing and low queuing delay. Thus, in general, our latency-based feedback is orthogonal to flow scheduling approaches.

8 Conclusion

In this paper, we explore latency feedback for congestion control in data center networks. To acquire reliable latency measurements, we develop both software and hardware level solutions to measure only the network-side latency. Our measurement results show that we can achieve sub-microseconds level of accuracy. Based on the accurate latency feedback, we develop DX that achieves high utilization and low queueing delay in datacenter networks. DX outperforms DCTCP [6] with 5.33x smaller queueing delay at 1 Gbps and 1.57x at 10 Gbps in testbed experiment. The queueing delay reduction is comparable or better than HULL [7] in simulation. Our prototype implementation shows that DX has much potential to be a practical solution in the real-world datacenters.

Acknowledgements

We thank our shepherd Edouard Bugnion and anonymous reviewers for their valuable comments. We also thank Keunhong Lee for providing his own implementation of Mellanox NIC driver and Sangjin Han for the SoftNIC implementation. This research was supported in part by Cisco Research Center (No. 576768), Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Korean government (MSIP) (No. 2014007580), and an Institute for Information communications Technology Promotion (IITP) grant funded by the Korean government (MSIP) (No. B0126-15-1078).

References

[1] Dina Katabi, Mark Handley, and Charlie Rohrs. Congestion Control for High Bandwidth-delay Product Networks. In *Proceedings of the ACM SIGCOMM conference*, 2002.

[2] Nandita Dukkipati, Masayoshi Kobayashi, Rui Zhang-shen, and Nick Mckeown. Processor Sharing Flows in the Internet. In *Proceedings of the International Workshop on Quality of Service (IWQoS)*, 2005.

[3] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. Better Never than Late: Meeting Deadlines in Datacenter Networks. In *Proceedings of the ACM SIGCOMM conference*, 2011.

[4] Alan Shieh, Srikanth Kandula, Albert Greenberg, and Changhoon Kim. Sharing the Data Center Network. In *Proceedings of USENIX NSDI conference*, 2011.

[5] Dongsu Han, Robert Grandl, Aditya Akella, and Srinivasan Seshan. FCP: A Flexible Transport Framework for Accommodating Diversity. In *Proceedings of the ACM SIGCOMM conference*, 2013.

[6] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data Center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM conference*, 2010.

[7] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less Is More: Trading a Little Bandwidth for Ultra-Low Latency in the Data Center. In *Proceedings of USENIX NSDI conference*, 2012.

[8] Lawrence Brakmo and Larry Peterson. TCP Vegas: End to End Congestion Avoidance on a Global Internet. *IEEE Journal on Selected Areas in Communications*, 13:1465–1480, 1995.

[9] David X. Wei, Cheng Jin, Steven H. Low, and Sanjay Hegde. FAST TCP: Motivation, Architecture, Algorithms, Performance. *IEEE/ACM Trans. Netw.*, 14(6):1246–1259, December 2006.

[10] Haitao Wu, Zhenqian Feng, Chuanxiong Guo, and Yongguang Zhang. ICTCP: Incast Congestion Control for TCP in Data Center Networks. In *Proceedings of the ACM CoNEXT*, 2010.

[11] Mario Flajslik and Mendel Rosenblum. Network Interface Design for Low Latency Request-Response Protocols. In *Proceedings of the USENIX ATC*, 2013.

[12] Intel DPDK: Data Plane Development Kit. `http://dpdk.org/`.

[13] Highly Accurate Time Synchronization with ConnectX-3 and TimeKeeper, Mellanox. `http://www.mellanox.com/related-docs/whitepapers/WP_Highly_Accurate_Time_Synchronization.pdf`.

[14] Iperf - The TCP/UDP Bandwidth Measurement Tool. `http://iperf.fr/`.

[15] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. SoftNIC: A Software NIC to Augment Hardware. Technical Report UCB/EECS-2015-155, EECS Department, University of California, Berkeley, May 2015.

[16] IEEE 1588: Precision Time Protocol (PTP).

[17] Christoph Lenzen, Philipp Sommer, and Roger Wattenhofer. Optimal Clock Synchronization in Networks. In *Proceedings of ACM SenSys Conference*, 2009.

[18] Mosharaf Chowdhury and Ion Stoica. Coflow: A Networking Abstraction for Cluster Applications. In *Proceedings of ACM HotNets*, 2012.

[19] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. VL2: A Scalable and Flexible Data Center Network. In *Proceedings of the ACM SIGCOMM conference*, 2009.

[20] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pFabric: Minimal Near-Optimal Datacenter Transport. In *Proceedings of the ACM SIGCOMM conference*, 2013.

[21] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. Fastpass: A Centralized Zero-Queue Datacenter Network. In *Proceedings of the ACM SIGCOMM conference*, 2014.

[22] Arun Venkataramani, Ravi Kokku, and Mike Dahlin. TCP Nice: A Mechanism for Background Transfers. In *Proceedings of USENIX OSDI conference*, 2002.

[23] Aleksandar Kuzmanovic and Edward W Knightly. TCP-LP: A Distributed Algorithm for Low Priority Data Transfer. In *Proceedings of IEEE INFOCOM Conference*, 2003.

[24] Balajee Vamanan, Jahangir Hasan, and T.N. Vijaykumar. Deadline-aware Datacenter TCP (D2TCP). In *Proceedings of the ACM SIGCOMM conference*, 2012.

[25] Ali Munir, Ihsan Qazi, Zartash Uzmi, Aisha Mushtaq, Saad Ismail, M. Iqbal, and Basma Khan. Minimizing Flow Completion Times in Data Centers. In *Proceedings of IEEE INFOCOM Conference*, 2013.

[26] Haitao Wu, Jiabo Ju, Guohan Lu, Chuanxiong Guo, Yongqiang Xiong, and Yongguang Zhang. Tuning ECN for Data Center Networks. In *Proceedings of the ACM CoNEXT*, 2012.

[27] David Zats, Tathagata Das, Prashanth Mohan, Dhruba Borthakur, and Randy Katz. DeTail: Reducing the Flow Completion Time Tail in Datacenter Networks. In *Proceedings of the ACM SIGCOMM conference*, 2012.

[28] Chi-Yao Hong, Matthew Caesar, and P. Brighten Godfrey. Finishing Flows Quickly with Preemptive Scheduling. In *Proceedings of the ACM SIGCOMM conference*, 2012.

[29] Ali Munir, Ghufran Baig, S Irteza, I Qazi, I Liu, and F Dogar. Friends, not Foes Synthesizing Existing Transport Strategies for Data Center Networks. In *Proceedings of the ACM SIGCOMM conference*, 2014.