DX: Latency-Based Congestion Control for Datacenters

Changhyun Lee, Chunjong Park, Keon Jang, Sue Moon, and Dongsu Han, *Member, IEEE*

***Abstract*— Since the advent of datacenter networking, achieving low latency within the network has been a primary goal. Many congestion control schemes have been proposed in recent years to meet the datacenters' unique performance requirement. The nature of congestion feedback largely governs the behavior of congestion control. In datacenter networks, where round trip times 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 explicit congestion notification (ECN) or even explicit in-network feedback to minimize the queuing delay. In this paper, 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 Gb/s networks. Such high accuracy enables us to design a new congestion control algorithm, DX, 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 5.33 times smaller median queueing delay at 1 Gb/s and 1.57 times at 10 Gb/s.**

***Index Terms*— Datacenter networks, congestion control, TCP, low latency.**

I. 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 [2], [3], 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 [4], [5], as the requirements for congestion control diversified [6].

With the advent of datacenter networking, identifying and leveraging more accurate and fine-grained feedback mechanisms have become even more crucial [7]. 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 [7]. As a result, proposals for datacenter congestion control predominantly leverage ECN (e.g., DCTCP [7] and HULL [8]) or explicit in-network feedback (e.g., RCP-type feedback [3]), 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 hardware modifications.

Although latency-based feedback has been previously explored in WAN [9], [10], 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 (§III). (ii) Measuring queuing delay requires high precision because a single MSS packet introduces only 0.3 (1.2) microseconds of queuing delay

Manuscript received June 5, 2015; revised April 20, 2016; accepted May 31, 2016. This work was supported in part by the Institute for Information and communications Technology Promotion within the Ministry of Science, ICT and Future Planning (MSIP) through the Program titled Development of an NFV-Inspired Networked Switch and an Operating System for Multi-Middlebox Services (B0101-15-1368) and through the Program titled Creation of PEP based on automatic protocol behavior analysis and Resource management for hyper connected for IoT Services (B0126-15-1078) and in part by the National Research Foundation of Korea within MSIP under Grant 2014007580. This paper is an extended version of a previous conference publication [1].

C. Lee is with the National Security Research Institute, Daejeon 34044, South Korea (e-mail: changhnlee@gmail.com).

C. Park and S. Moon are with the School of Computing, Korea Advanced Institute of Science and Technology, Daejeon 34141, South Korea (e-mail: cjpark87@gmail.com; sbmoon@kaist.edu).

K. Jang was with Intel Labs, Santa Clara, CA 95054 USA. He is now with Google, Mountain View, CA 94043 USA (e-mail: gunjang11@gmail.com).

D. Han is with the School of Electrical Engineering, Korea Advanced Institute of Science Technology, Daejeon 34141, South Korea (e-mail: dongsu.han@gmail.com).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TNET.2016.2587286

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 [7], [11].

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 [12], [13] and hardware technology [14] 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 [8] 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) thorough evaluation of ECN-based congestion feedback in comparison to latency feedback; (ii) novel techniques to accurately measure in-network queuing delay based on end-to-end latency measurements; (iii) 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 (iv) a prototype that demonstrates the feasibility and its benefits in our testbed.

II. Comparison of Congestion Feedback

As congestion control in datacenters needs to react within RTTs orders of magnitude smaller than in WAN, most proposals for datacenter congestion control leverage ECN or explicit in-network feedback [7], [8], [15]–[17]. We describe and compare them with latency-based feedback.

Explicit Congestion Notification: DCTCP [7] and many other proposals [8], [15], [16] use ECN to detect congestion before the queue actually overflows. Typically, congestion level is measured by calculating the fraction of ECN-marked ACK packets out of the total ACK packets in each window. To absorb instant fluctuations in queue occupancy, DCTCP takes the moving average of the sample fractions over multiple windows and estimates the probability with which the queue length is higher than the marking threshold. After detecting congestion, it decreases the window size in proportion to the congestion level. This allows DCTCP to maintain the average queuing delay small, near the marking threshold, K.

Explicit in-network feedback provides multi-bit congestion indicator that is much more accurate and fine-grained than ECN and has been used in several proposals for datacenter networking [2], [4], [6], [18], [19]. The key difference is that it also notifies how much the network is under-utilized, in addition to signaling congestion. Such feedback enables multiplicative-increase and multiplicative-decrease (MIMD), which results in high utilization and fast convergence [2], [3], [6]. However, this "idealized" feedback requires in-network support. Currently, we are unaware of any commodity switches that are capable of generating explicit in-network feedback [7].

Latency feedback: TCP Vegas [9] has introduced latency congestion feedback in wide-area network. If latency can be accurately measured to reflect the network congestion, it has more benefits than other types of feedback. First, it is implicit feedback that does not require any in-network support. Second, it can take on a much larger range of values than ECN or QCN [17], [20], offering a finer-grained form of feedback. The difference from in-network feedback of RCP [3] or XCP [2] is that latency feedback cannot signal the remaining network capacity when the network is not being fully utilized, but only notifies when and how congested the network is. Our proposal in this paper is using latency feedback in datacenter networks. In Section III, we show that our measurement methodology effectively captures the network queueing delay in datacenter environment.

The rest of this section provides a quantitative comparison of the feedback. Note that we only try to evaluate the feedback itself, not the congestion control algorithm using the feedback.

A. Accuracy of ECN-Based Feedback

We quantify the accuracy of DCTCP's ECN feedback with respect to an ideal form of explicit in-network feedback that accurately reflects the congestion level, such as that of RCP or XCP. To do this, we take the queue size as the ground truth congestion level and plot the measured feedback using *ns-2* simulation.

We use a simple dumbbell topology of 40 DCTCP senders and receivers with 10 Gbps link capacity. The RTT between nodes is 200 μs, the ECN marking threshold is set to $K = 35$ [7], and the queue capacity on the bottleneck link is set to 100 packets; the queue does not overflow during the simulation. Each sender starts a DCTCP flow and records the congestion level given by the fraction of ECN marked packets for each congestion window. We take the average switch queue occupancy during the window as the ground-truth congestion level.

Figure 1 shows the percentage of ECN marked packets and its moving average as used by DCTCP. The x-axis represents the ground-truth, and the y-axis indicates the measured level of congestion (percentage of ECN marked packets). Along with the ECN congestion feedback, we plot a line for the ideal congestion feedback that informs the exact number of packets in the queue. The ideal congestion feedback models a form of explicit in-network feedback similar to that of RCP [3]. For example, the ideal feedback at 100% congestion, with respect to the maximum queue size, should be 100 queued packets, which is the amount to reduce in the next round to achieve zero queuing delay.

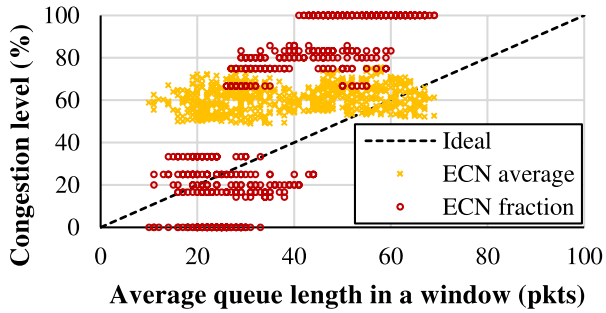

Fig. 1. Congestion level vs. ECN fraction and its moving average.

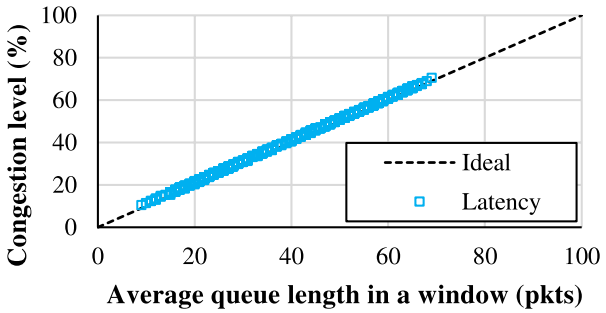

Fig. 2. Congestion level vs. measured latency.

From this simple experiment, we make the following three key observations:

Accuracy is low. The fraction of ECN-marked packets is not a good indicator of the congestion level. About 50% of feedback is either 0 or 100; 16% (33%) of the times, the measured congestion level was 0 (100). Values other than 0 and 100 do not reflect the level of congestion accurately either. A wide range of switch queue occupancy shares the same feedback. For example, both the queue lengths, 28 and 64 can result in the feedback of 80%. As a result, the Pearson correlation coefficient between the actual congestion and measured feedback was only 0.6924 (compared to 0.9998 for latency feedback presented later); 1.0 is the highest correlation and 0.0 means no correlation. The RMSE (root mean square error) with respect to the ideal feedback was 33.79 (compared to 1.05 of latency feedback).

Granularity is too coarse. The congestion feedback in Figure 1 is very coarse grained. The fundamental reason is its dependency on the window size. For example, five is the most frequently appearing window size in our simulation. In this case, the feedback (i.e., ECN-marked fraction) can only take on six values from 0%, 20%, 40%, 60%, 80%, to 100%, while the actual congestion level is between 9 and 69 packets (61 different levels).

Taking the moving average does not help and even degrades the accuracy as the measured congestion level stays the same for a wide range of queue lengths (Figure 1). We observe in Figure 1 that the moving average smoothes out the extreme congestion level values of 0s and 1s. However, very little correlation exists between the ECN-based congestion feedback and the actual queue lengths. The measured congestion level (i.e., the moving average) always resides between 0.475 and 0.755, while the actual queue occupancy (the ground-truth congestion level) varies between 7 and 70 packets. As a result, the correlation coefficient drops to 0.1790, and the RMSE with respect to the ideal feedback is relatively high at 24.63.

B. Accuracy of Latency-Based Feedback

In the face of the above disparity between the actual queueing and ECN-based feedback, we have turned to latency feedback as an alternative. As both senders and receivers are under the same administrative domain in datacenter networks, we assume that we could instrument both ends, and high-precision latency measurements are feasible. Later in Section III, we introduce our detailed techniques for accurate latency measurement. Assuming that latency can be measured accurately for now, we verify that latency measurements accurately reflect the ground-truth congestion level, using *ns-2* simulation.

The congestion level is measured once for every congestion window in the following way. The sender measures the RTT for every data packet and sets its minimum as the base RTT without queueing delay. The difference between the base RTT and a sample RTT represents the queueing delay, which is the congestion level.

Figure 2 shows the actual congestion level versus latency based congestion-level measurement. For ease of comparison, we consider the maximum possible queueing delay as the congestion level 100% and translate the measured latency into congestion level accordingly. Latency feedback (i.e., queuing delay) naturally reflects the average queue lengths. The correlation coefficient is as high as 0.9998, and the RMSE against the ideal feedback is only 1.05, which is 32 times smaller than the raw ECN fraction feedback, and 23 times smaller than the moving average.

Now the next section discusses how to achieve accurate latency measurement to capture the congestion level in the real network.

III. 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 3 shows the round trip time measured by the sender's kernel when saturating a 10 Gbps link; we generate TCP traffic using *iperf* [21] 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.

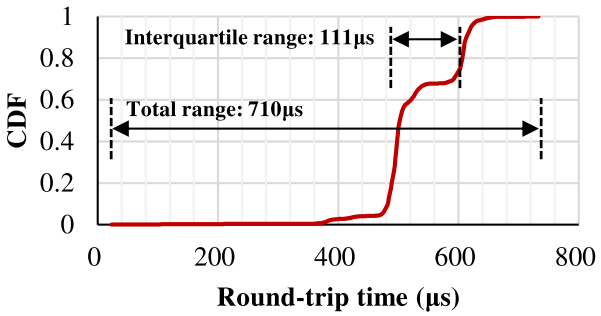

Fig. 3. Round-trip time measured in kernel.

TABLE I
SOURCES OF ERRORS IN LATENCY MEASUREMENT AND OUR TECHNIQUES FOR MITIGATION

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

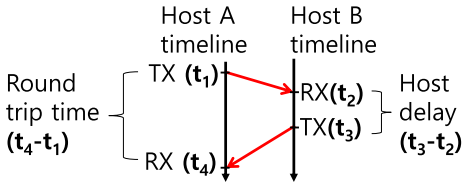

Fig. 4. Timeline of timestamp measurement points.

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

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 80 μs 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 the RX on top of DPDK [13]. Hardware timestamping innately removes such delay.

Now, we need to deal with the end-host stack delay at the receiver. Figure 4 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 μs.

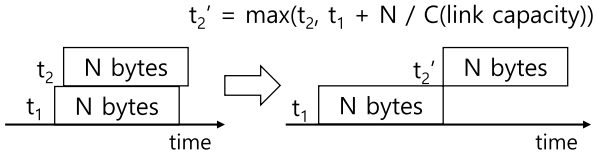

Fig. 5. Example delay calibration for bursty packet reception.

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 [22].

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 [8] 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. To quantify such noise, we run a simple experiment where a sender is connected to a receiver back to back and sends traffic at near line rate of 9.5 Gbps. Ideally, all packets should be spaced with 1.23 μs interval, but the result shows that 68% of the packet gaps for TX and 32% for RX fall below 1.23 μs. The detailed error distribution can be found in our previous paper [1]. 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 5. 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.

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

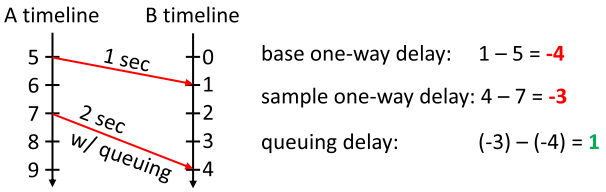

Fig. 6. One-way queuing delay without time synchronization.

synchronization between two hosts. PTP (Precision Time Protocol) enables clock synchronization with sub-microseconds [23]. 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 6. 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 among 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 [24]. 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.

IV. DX: Latency-Based Congestion Control

A. Limitations of Existing Algorithms

Our latency measurement serves as much more accurate congestion feedback than previous kernel-based latency measurement, so existing latency-based congestion control algorithms can also benefit from it. In this subsection, we study whether existing latency-based algorithms can be used in datacenter networks to meet the low queueing delay requirement. The first latency-based algorithm proposed in wide-area networks is TCP Vegas [9], and other later proposed algorithms share the same core idea with TCP Vegas. Therefore we focus on TCP Vegas and analyze its performance in datacenter networks. If TCP Vegas turns out to work well, then we will not need to develop another algorithm and just re-use TCP Vegas. If not, we need to figure out why TCP Vegas does not work and use the lessons learned to design a new algorithm.

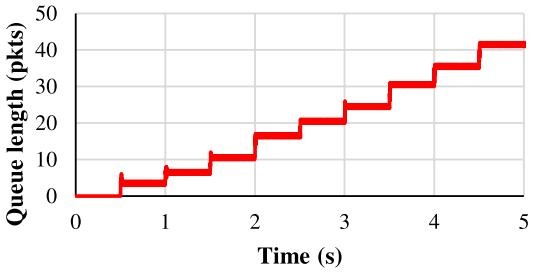

Fig. 7. Queue length with the increasing number of TCP Vegas flows.

We conduct *ns-2* simulation in a dumbbell topology to test if TCP Vegas achieves low queueing delay. We have ten idle senders in the beginning, and we activate each sender with 0.5 second interval so that we have ten active flows in the end. Figure 7 shows the queueing delay evolution as the number of flows is increased. We notice that the queueing delay increases with the number of flows in the bottleneck link; each flow adds up its own share of queueing to existing queueing. The queue length is consistently as high as 42 packets from 4.5 s to 5.0 s where ten flows are sharing the link. As datacenter workloads are very dynamic and the number of flows is not bounded, TCP Vegas cannot always guarantee low queueing delay.

We observe another drawback of TCP Vegas in the fairness among flows. According to the TCP Vegas algorithm, a sender determines the congestion level from (measured RTT - base RTT). This approach can provide fairness only when all the senders maintain the same base RTT value. The simulation result, however, tells us otherwise. The first flow has 205 μs for base RTT and the last flow has 238 μs; later flows in the network get larger base RTT. In this case, the last flow under-estimates the congestion level and tries to send faster than it is supposed to.

From the above simulation, we learn two lessons to be used in designing a new congestion control algorithm for datacenters: i) the algorithm should be able to drop the queue length down to zero quickly as soon as it observes congestion; ii) the algorithm should take into account the number of flows in the network when decreasing window size. We explain how we reflect these lessons in our algorithm in the next subsection.

B. DX Algorithm Details

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 [4], [6]–[8], [11].

DX is a window-based congestion control algorithm, and its congestion avoidance follows the popular AIMD (Additive Increase Multiplicative Decrease) 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 once in a round-trip 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 round-trip, 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 $C \cdot R$:

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

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

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

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.

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

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

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

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

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

Solving for V results in:

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

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 a steady state; DX is an AIMD algorithm, and a previous work [25] has shown that AIMD algorithms converge to fairness. 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:

$$V = \frac{R \cdot W^*}{W^* - 1} \quad (8)$$

In calculating V, the sender only needs to know the base 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.

So far we have explained how DX handles the event of positive queueing delay. Although DX does not experience packet loss by queue overflows, a timeout event can still occur due to physical level failures. In traditional TCP algorithms, these timeout events are considered as congestion alarms and dealt by window size decrement. In DX, however, such timeout events are not caused by congestion confirmed by measured queueing delay, so the window size can remain the

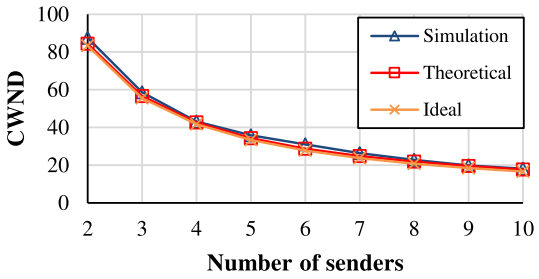

Fig. 8. Steady-state CWND comparison.

same and keep utilizing the network link fully. Being able to identify the source of packet loss is one of the DX's strengths; retransmitting lost packets and updating timeout value can be done in the same manner as TCP.

C. Steady-State Analysis

Here we provide a simple analysis on the steady-state behavior of DX. Our interest is in how close DX is to the ideal congestion control algorithm with completely zero queueing.

In our analysis, we compare the three kinds of window size: ideal, theoretical, and simulational. The ideal window size is easily computed by the bandwidth-delay product divided by the number of flows; 100% link utilization with zero queueing. The theoretical value is computed using the worst case queue length in our algorithm. DX increases the window size only when the queue length is zero, so the worst case happens when all the flows see zero queue length and decide to increase their window by one simultaneously. Then we can have as much as n queued packets where n is the number of flows. The theoretical window size can be now calculated from (bandwidth-delay product $+$ n) divided by n. Finally, the simulational value is the result from *ns-2* simulation. We use 10 Gbps link capacity and 200 μs RTT for this analysis.

We present the result in Figure 8. To test various scenarios, we increase the number of flows from two to ten and plot the results. We observe that the ideal value, which is the lower bound, is very close to theoretical value of DX. The simulation result is also close to the theoretical value as the maximum difference is 2.69 packets at $n = 2$. The disparity between the theoretical and simulation results come from the assumption used in the theoretical computation that all flows are synchronized.

Next we observe the convergence behavior of DX in comparison to DCTCP. For this observation, we use the same convergence analysis methodology from a previous AIMD analysis work [25]. In our analysis scenario, a flow (denoted as flow #1) is occupying the total link bandwidth in the beginning. Then the second flow (denoted as flow #2) comes into the network, and after a certain amount of time, both flows converge to the fair share throughput and reach steady-state.

We plot the change in each flow's window size in Figure 9. The x-axis is the window size of flow #1, and y-axis is the window size of flow #2. The fairness line represents the condition where two flows have the same window size, hence fair throughput. The efficiency line represents the condition where the sum of the window size of two flows is exactly same

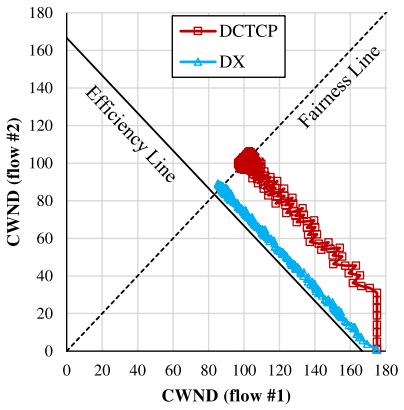

Fig. 9. Convergence of two flows with DX and DCTCP.

as the bandwidth-delay product; the right side of efficiency line means over-utilization (i.e., queueing) and the left side means under-utilization. Starting from the bottom-right corner, both DCTCP and DX converge near the fairness line, but DX takes a more direct path than DCTCP. At steady-state, DX is much closer to the efficiency line than DCTCP so it minimizes the unnecessary queuieng in the network.

V. 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.

A. Timestamping and Delay Calculation

We measure four timestamp values as shown in section III Figure 4: 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 §III. 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 (§V-B).

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 § III, 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 [22] 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.

B. 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, to be robust against latency measurement noise, we use headroom when determining congestion; DX does not decrease window size when $Q < headroom$. The size of the headroom is determined by the level of measurement noise. For example, if each latency measurement has 10 μs error at maximum, the headroom should be set to 10 μs because any measurements smaller than 10 μs can be a false congestion alarm.

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 [7].

VI. 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.

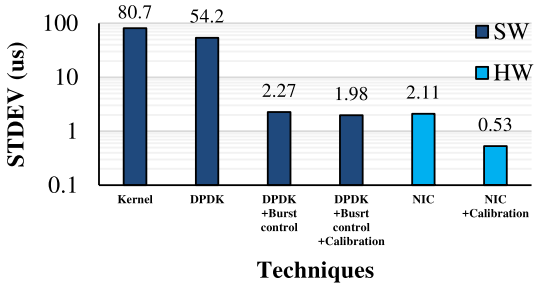

Fig. 10. Improvements with noise reduction techniques.

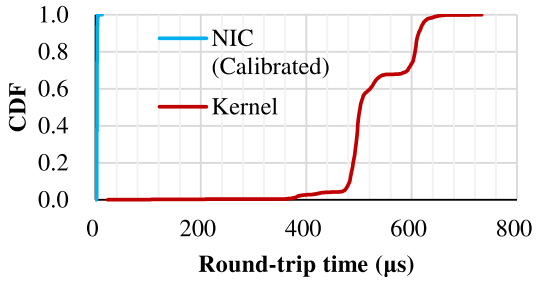

Fig. 11. Improvement on RTT measurement error compared to kernel's.

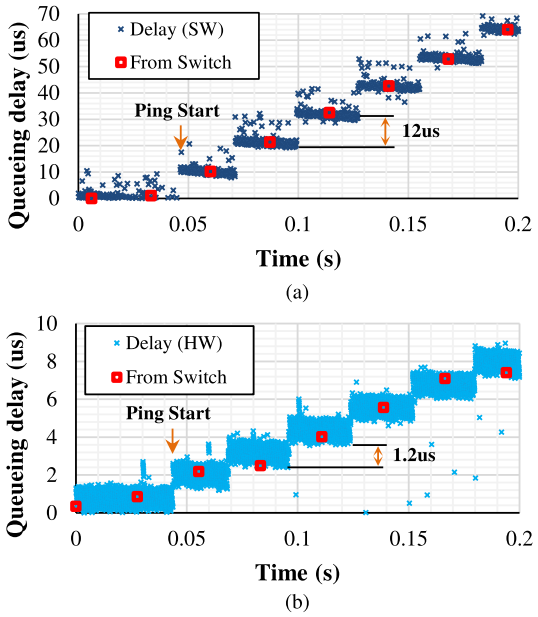

Fig. 12. Accuracy of queuing delay measurement. (a) 1 Gbps with software timestamping. (b) 10 Gbps with hardware timestamping.

Finally, we perform large-scale evaluation to compare DX's overall performance against the state of the art: DCTCP [7] and HULL [8].

A. 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 10. 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. The calibration effectively removes the inter packet gap samples smaller than link transmission delay which originally took up 68% for TX and 32% for RX. The figures for this evaluation can be found in our previous paper [1].

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 11. 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 a 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 12 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.

B. 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* [21] to generate TCP flows for 15 seconds. For comparison, we run DCTCP in the same environment. The ECN marking threshold

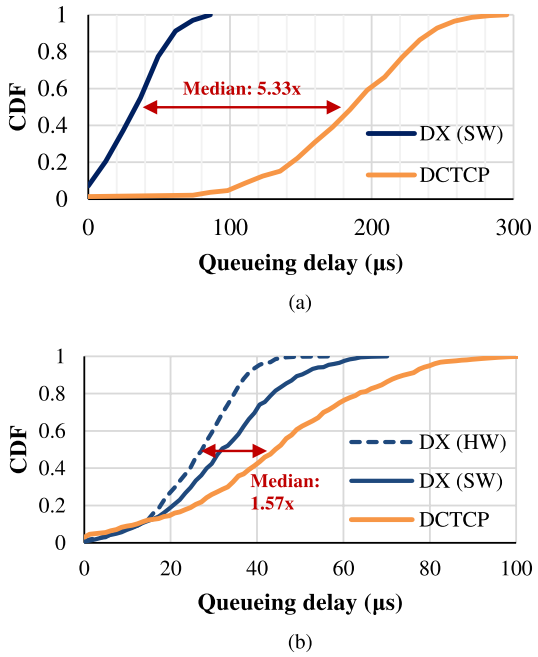

Fig. 13. Queue length comparison of DX against DCTCP in Testbed. (a) 1 Gbps bottleneck. (b) 10 Gbps bottleneck.

for DCTCP is set to the recommended value of 20 at 1 Gbps and 65 at 10 Gbps [7]. During the experiment, the switch queue length is measured every 20 ms by reading the register values from the switch chipset; the queue length is measured in bytes and converted into time.

We first present the result at the 1 Gbps bottleneck link in Figure 13. Here we focus on the queue length of each protocol as the throughput does not exhibit much difference; both protocols successfully saturate the bottleneck link during the experiment, and each flow achieves the fair-share throughput of 500 Mbps in the 1 Gbps link and 5 Gbps in the 10 Gbps link.

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 13 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 more effective in maintaining low queue occupancy than ECN feedback. 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.

C. Large-Scale Simulation

1) Dumbbell Topology With More Senders: In this section, we evaluate DX, DCTCP, and HULL in simulation to observe the performance in larger-scale environment. We run *ns-2* simulator using a dumbbell network topology with 10 Gbps link capacity. The latency measurement in simulation is accurate without any noise.

For scalability test, we 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 [7]. We measure the queuing delay and utilization, and summarize the findings below. The graphs can be found in our previous paper [1].

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 [26]. 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 10 Gbps link one by one with 1 second interval and stop each flow after 5 seconds of transfer. In Figure 14, 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.

Impact of latency noise: We evaluate the impact of latency noise to the headroom size and average queue length in DX. 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 15 shows the required headroom for full link utilization and the resulting queue length in average.

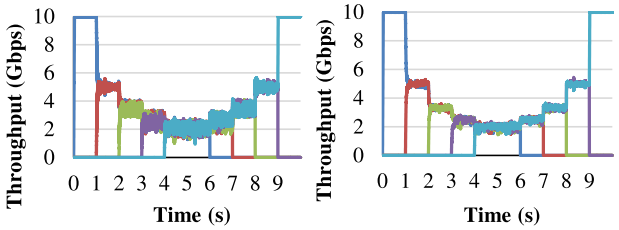

Fig. 14. Fairness of five flows with DCTCP and DX. (a) DCTCP. (b) DX.

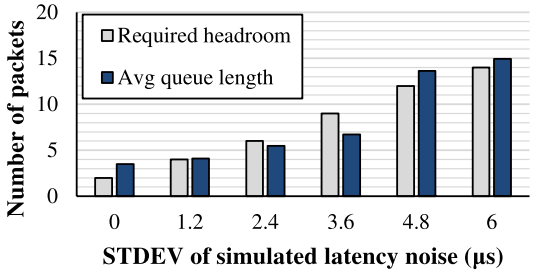

Fig. 15. Impact of latency noise to headroom and queue length.

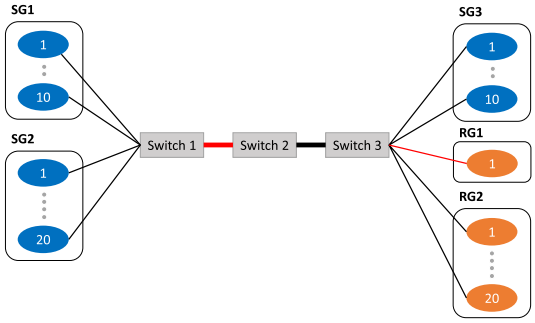

Fig. 16. Multi-bottleneck scenario.

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.

2) Multi-Bottleneck Scenario: We evaluate the performance of DX in a multi-bottleneck scenario. We use the same network topology used in DCTCP evaluation [7] as shown in Figure 16. In this topology, there are three sender groups (i.e., SG1, SG2, and SG3) and two receiver groups (i.e., RG1 and RG2). When the simulation begins, SG1 sends best-effort traffic to RG1, and SG2 sends traffic to RG2. At the same time, RG2 also receives traffic from SG3. When all the senders and receivers are active, the 10 Gbps network link between Switch 1 and Switch 2 becomes a bottleneck link for SG1 and SG2, and the 1 Gbps link between Switch 3 and RG1 becomes a bottleneck link for SG1 and SG3. Therefore the traffic from SG1 to RG1 passes through two bottleneck links. In this scenario, the ideal fair throughput is 50 Mbps for SG1 and SG3, and 475 Mbps for SG2. When we run DX, SG1 gets 51.5 Mbps, SG2 gets 477.7 Mbps, and SG3 gets 48.5 Mbps. For comparison, we also run DCTCP with the same setting, and we get 46 Mbps for SG1, 475 Mbps for SG2, 54 Mbps for SG3. We conclude that DX provides fair enough and comparable throughput to DCTCP even with multi-bottleneck network scenario.

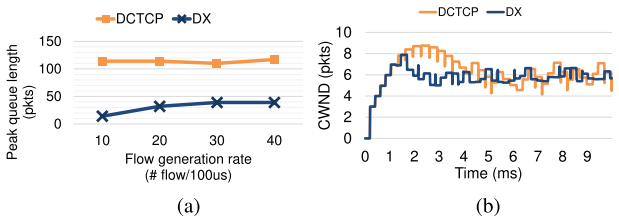

Fig. 17. Reaction to incast workload with DX and DCTCP. (a) Instant queue buildup. (b) CWND evolution.

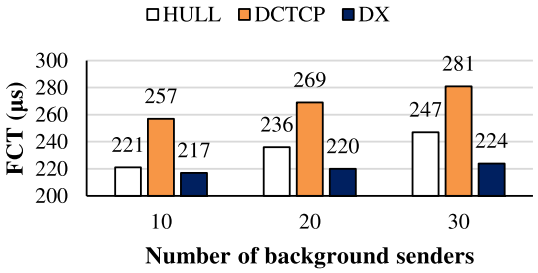

Fig. 18. 99th-percentile FCT of small flows under large background flows.

3) Synthetic Datacenter Workload: In this section, we focus on specific traffic patterns that are common in data center networks and show the performance benefit of DX.

Incast traffic: Although DX does not originally aim at solving the incast problem [27] by design, maintaining small queue and quick responsiveness to the congestion give advantages to certain type of incast traffic.

When a node in datacenters sends queries to multiple different nodes, the responses to the query can arrive back at the sender simultaneously within a short period of time. To simulate such response traffic in data centers, we generate 40 flows to a single receiver at varying flow generation rate and observe the queue length at the bottleneck switch. Each flow is 2 MB size in our simulation.

Figure 17 shows the peak queue length at the time of flow arrival. DCTCP suffers from instant queue buildup of hundred packets throughout all the flow generation rates, even at 10 flows per microsecond. On the other hand, DX has lower queue buildup. To examine the difference in handling incast traffic, we show the behavior of congestion window size. Figure 17 shows the congestion window size when the incast happens at 40 flows per 100 μs. The result shows that DX reacts to the congestion much earlier than DCTCP does reducing excess packets in the network.

Small flows with large background flows: Small flows often co-exist with large bulk transfer in datacenters. In this case, the flow completion time (FCT) of small flows suffers due to the increased queuing delay. To simulate DX's behavior in such scenario, we produce small flows with large background flows in the network. The large flow has infinite size starting at the beginning of the simulation, and we increase the number of

large flow senders from 10 to 30. We then generate a thousand 1 KB flows according to a Possion arrival process. Figure 18 shows 99th percentile FCT for DCTCP, HULL, and DX. The base RTT between a sender and a receiver is 200 μs. Note that the y-axis begins from the base RTT.

DX achieves the minimum 99th percentile FCT among the three. DX performs 18% and 2% better than DCTCP and HULL respectively with 10 background flows. As the number of background flows increases, the 99% FCT of DX does not increase as much compared to DCTCP and HULL; 10 additional senders increase only 3-4 μs in FCT. This behavior originates from DX's zero queue targeting. Even when many large flows are present at the bottleneck switch, the queue length frequently falls down to a small value close to zero.

4) Empirical Datacenter Workload: 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 [8]. 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 a 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 [7] and data mining [28].

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 20 MB, and the average flow size is 654 KB [29]. In Figure 19, we present the flow completion time (FCT) in four flow-size groups: (0 KB,10 KB), [10 KB,100 KB), [100 KB,10 MB), and [10 MB,∞).

For the flows smaller than 10 KB, 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 [10 KB-100 KB) 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 1 GB. The flow size is highly skewed that 80% of flows are smaller than 10 KB [29] so 95% of bytes come from flows larger than 30 MB; the average flow size is 7,452 KB. The flow completion time of data mining workload is presented in Figure 20.

The performance improvement of DX is more outstanding for data minining workload than for search workload. In the three flow groups up to 10 MB, 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.

VII. 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 §III. 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 [30], 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 [7]. 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. Such techniques to segregate latency-sensitive traffic from bulk transfer traffic have been widely discussed in the literature [31]–[33] and some previous low latency transport protocols have been successfully deployed where they co-exist with TCP [34], [35] with only minor switch reconfiguration. DX can adopt the

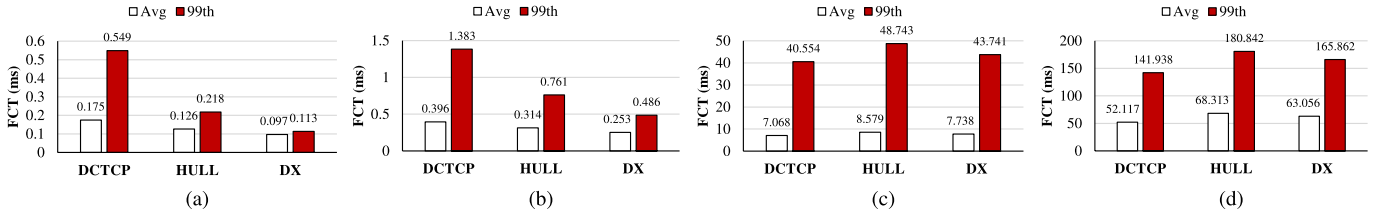

Fig. 19. Flow completion time of search workload. (a) (0 KB, 10 KB). (b) [10 KB, 100 KB). (c) [100 KB, 10 MB). (d) [10 MB, ∞).

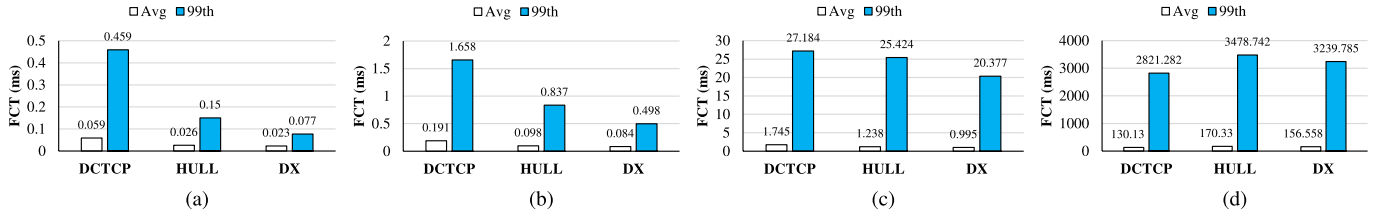

Fig. 20. Flow completion time of data mining workload. (a) (0 KB, 10 KB). (b) [10 KB, 100 KB). (c) [100 KB, 10 MB). (d) [10 MB, ∞).

same isolation technique in the need for co-exiting with other TCP algorithms.

VIII. 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 [9] is one of the earliest work and aims at achieving high throughput by avoiding loss. FAST TCP [10] is designed to quickly reach the fair-share throughput and uses latency for an equation parameter. TCP Nice [36] and TCP-LP [37] operate in low priority minimizing interference with other flows. Using latency feedback for datacenters was first proposed by our previous conference paper [1], and recently TIMELY [38] independently explored latency feedback in datacenters using hardware-based measurements and RTT gradients.

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 [7] 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 [8] 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^{2}TCP [15] 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^{2}DCT [16] considers flows' priority when reducing window size, and the priority is determined by the scheduling policy used in the network. ECN* [39] 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 [4] employs similar mechanism to RCP so that it can control flow rates to implement deadline based scheduling. DeTail [40] has implemented a new cross-layer network stack so that flows can avoid congested paths in the network, and PDQ [41] proposes distributed scheduling of flows that posses different priorities. These solutions are much harder to deploy than end-to-end solutions.

IX. 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 [7] 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 [8] in simulation. Our prototype implementation shows that DX has much potential to be a practical solution in the real-world datacenters.

References

[1] C. Lee, C. Park, K. Jang, D. Han, and S. Moon, "Accurate latency-based congestion feedback for datacenters," in *Proc. USENIX ATC*, 2015, pp. 403–415.

[2] D. Katabi, M. Handley, and C. Rohrs, "Congestion control for high bandwidth-delay product networks," in *Proc. ACM SIGCOMM*, 2002, pp. 89–102.

[3] N. Dukkipati, M. Kobayashi, R. Zhang-Shen, and N. Mckeown, "Processor sharing flows in the Internet," in *Proc. 13th Int. Workshop Quality Service (IWQoS)*, 2005, pp. 271–285.

[4] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron, "Better never than late: Meeting deadlines in datacenter networks," in *Proc. ACM SIGCOMM*, 2011, pp. 50–61.
[5] A. Shieh, S. Kandula, A. Greenberg, C. Kim, and B. Saha, "Sharing the data center network," in *Proc. USENIX NSDI*, 2011, pp. 309–322.
[6] D. Han, R. Grandl, A. Akella, and S. Seshan, "FCP: A flexible transport framework for accommodating diversity," in *Proc. ACM SIGCOMM*, 2013, pp. 135–146.
[7] M. Alizadeh *et al.*, "Data center TCP (DCTCP)," in *Proc. ACM SIGCOMM*, 2010, pp. 63–74.
[8] M. Alizadeh *et al.*, "Less is more: Trading a little bandwidth for ultra-low latency in the data center," in *Proc. USENIX NSDI*, 2012, p. 19.
[9] L. S. Brakmo and L. L. Peterson, "TCP Vegas: End to end congestion avoidance on a global Internet," *IEEE J. Sel. Areas Commun.*, vol. 13, no. 8, pp. 1465–1480, Sep. 1995.
[10] D. X. Wei, C. Jin, S. H. Low, and S. Hegde, "FAST TCP: Motivation, architecture, algorithms, performance," *IEEE/ACM Trans. Netw.*, vol. 14, no. 6, pp. 1246–1259, Dec. 2006.
[11] H. Wu, Z. Feng, C. Guo, and Y. Zhang, "ICTCP: Incast congestion control for TCP in data center networks," in *Proc. ACM CoNEXT*, 2010, Art. no. 13.
[12] M. Flajslik and M. Rosenblum, "Network interface design for low latency request-response protocols," in *Proc. USENIX ATC*, 2013, pp. 333–346.
[13] *Intel DPDK: Data Plane Development Kit*. [Online]. Available: http://dpdk.org/
[14] *Highly Accurate Time Synchronization With ConnectX-3 and TimeKeeper*, Mellanox, Yokne'am Illit, Israel, 2013.
[15] B. Vamanan, J. Hasan, and T. N. Vijaykumar, "Deadline-aware datacenter TCP (D2TCP)," in *Proc. ACM SIGCOMM*, 2012, pp. 115–126.
[16] A. Munir *et al.*, "Minimizing flow completion times in data centers," in *Proc. IEEE INFOCOM*, Apr. 2013, pp. 2157–2165.
[17] R. Pan, B. Prabhakar, and A. Laxmikantha. (2007). *QCN: Quantized Congestion Notification*. [Online]. Available: http://www.ieee802.org/1/files/public/docs2007/au-prabhakar-qcn-description.pdf
[18] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron, "Towards predictable datacenter networks," in *Proc. ACM SIGCOMM*, 2011, pp. 242–253.
[19] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron, "Decentralized task-aware scheduling for data center networks," in *Proc. ACM SIGCOMM*, 2014, pp. 431–442.
[20] A. Kabbani, M. Alizadeh, M. Yasuda, R. Pan, and B. Prabhakar, "AF-QCN: Approximate fairness with quantized congestion notification for multi-tenanted data centers," in *Proc. 18th IEEE Symp. High Perform. Interconnects*, Aug. 2010, pp. 58–65.
[21] *iPerf—The TCP/UDP Bandwidth Measurement Tool*. [Online]. Available: http://iperf.fr/
[22] S. Han *et al.*, "SoftNIC: A software NIC to augment hardware," Dept. EECS, Univ. California, Berkeley, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2015-155, 2015.
[23] (2009). *IEEE 1588 Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*. [Online] Available: http://www.nist.gov/el/isd/iee e/ieee1588.cfm
[24] C. Lenzen, P. Sommer, and R. Wattenhofer, "Optimal clock synchronization in networks," in *Proc. ACM SenSys*, 2009, pp. 225–238.
[25] D.-M. Chiu and R. Jain, "Analysis of the increase and decrease algorithms for congestion avoidance in computer networks," *Comput. Netw. ISDN Syst.*, vol. 17, no. 1, pp. 1–14, Jun. 1989.
[26] M. Chowdhury and I. Stoica, "Coflow: A networking abstraction for cluster applications," in *Proc. ACM HotNets*, 2012, pp. 31–36.
[27] V. Vasudevan *et al.*, "Safe and effective fine-grained TCP retransmissions for datacenter communication," in *Proc. ACM SIGCOMM*, 2009, pp. 303–314.
[28] A. Greenberg *et al.*, "VL2: A scalable and flexible data center network," in *Proc. ACM SIGCOMM*, 2009, pp. 51–62.
[29] M. Alizadeh *et al.*, "pFabric: Minimal near-optimal datacenter transport," in *Proc. ACM SIGCOMM*, 2013, pp. 435–446.
[30] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal, "Fastpass: A centralized 'zero-queue' datacenter network," in *Proc. ACM SIGCOMM*, 2014, pp. 307–318.
[31] S. Bensley, L. Eggert, D. Thaler, P. Balasubramanian, and G. Judd, *Datacenter TCP (DCTCP): TCP Congestion Control for Datacenters*, document draft-ietf-tcpm-dctcp-01, IETF Draft, 2015.
[32] K. De Schepper, B. Briscoe, O. Bondarenko, and I. Tsang, *DualQ Coupled AQM for Low Latency, Low Loss and Scalable Throughput*, document draft-briscoe-aqm-dualq-coupled-00, IETF Draft, 2015.
[33] S. M. Irteza, A. Ahmed, S. Farrukh, B. N. Memon, and I. A. Qazi, "On the coexistence of transport protocols in data centers," in *Proc. IEEE ICC*, Jun. 2014, pp. 3203–3208.
[34] G. Judd, "Attaining the promise and avoiding the pitfalls of TCP in the datacenter," in *Proc. USENIX NSDI*, 2015, pp. 145–157.
[35] Y. Zhu *et al.*, "Congestion control for large-scale RDMA deployments," in *Proc. ACM SIGCOMM*, 2015, pp. 523–536.
[36] A. Venkataramani, R. Kokku, and M. Dahlin, "TCP Nice: A mechanism for background transfers," in *Proc. USENIX OSDI*, 2002, pp. 329–343.
[37] A. Kuzmanovic and E. W. Knightly, "TCP-LP: A distributed algorithm for low priority data transfer," in *Proc. IEEE INFOCOM*, Mar./Apr. 2003, pp. 1691–1701.
[38] R. Mittal *et al.*, "TIMELY: RTT-based congestion control for the datacenter," in *Proc. ACM SIGCOMM*, 2015, pp. 537–550.
[39] H. Wu *et al.*, "Tuning ECN for data center networks," in *Proc. ACM CoNEXT*, 2012, pp. 25–36.
[40] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz, "DeTail: Reducing the flow completion time tail in datacenter networks," in *Proc. ACM SIGCOMM*, 2012, pp. 139–150.
[41] C.-Y. Hong, M. Caesar, and P. B. Godfrey, "Finishing flows quickly with preemptive scheduling," in *Proc. ACM SIGCOMM*, 2012, pp. 127–138.

Changhyun Lee received the B.S. and Ph.D. degrees in computer science from the Korea Advanced Institute of Science and Technology (KAIST) in 2007 and 2015, respectively. He is currently a Researcher with the National Security Research Institute, South Korea. His research interests include datacenter networking, congestion control, and network performance measurement/analysis.

Chunjong Park received the B.S. degree in computer science from the Korea Advanced Institute of Science and Technology (KAIST) in 2015, where he is currently pursuing the master's degree in computer science under the supervision Dr. S.-J. Lee. His research interests are primarily in networking and mobile systems.

Keon Jang received the Ph.D. and B.S. degrees in computer science from Korea Advanced Institute of Science and Technology (KAIST) in 2012 and 2006, respectively. From 2012 to 2014, he was with Microsoft Research, Cambridge, U.K., as a Post-Doctoral Researcher. From 2014 to 2015, he was a Research Scientist with Intel Labs and a Visiting Scholar with UC Berkeley. He is currently a Software Engineer with Google.

Sue Moon received the B.S. and M.S. degrees from Seoul National University, Seoul, South Korea, in 1988 and 1990, respectively, and the Ph.D. degree from the University of Massachusetts at Amherst in 2000, all in computer science. She is currently a Full Professor with the School of Computing, KAIST. From 1999 to 2003, she was with the IPMON project, Sprint ATL, Burlingame, CA, USA. Her main research topics are high-speed networking platforms and online social media. She has been active in technical program committees and received many awards for technical achievements. She has served as a TPC Co-Chair for ACM Multimedia 2004, the APSys 2011, and the WWW 2013. She served as a General Chair for PAM 2009 and APSys 2012, and as a Vice Organization Chair for WWW 2014. She is a recipient of the 16th Young Engineer's Award by National Academy of Engineering of Korea, and KAIST Chair Professorship from 2011 to 2014.

Dongsu Han (M'16) received the B.S. degree in computer science from the Korea Advanced Institute of Science (KAIST) in 2003, and the Ph.D. degree in computer science from Carnegie Mellon University in 2012. He is currently an Assistant Professor with the School of Electrical Engineering, Graduate School of Information Security, KAIST. He is interested in networking, distributed systems, and network/system security.