Minimizing Flow Completion Times in Data Centers

Ali Munir, Ihsan A. Qazi, Zartash A. Uzmi, Aisha Mushtaq, Saad N. Ismail, M. Safdar Iqbal, Basma Khan
Computer Science Department, LUMS
Email: {ali.munir,ihsan.qazi,zartash,14100142,14100055,12100005,basma.khan}@lums.edu.pk

Abstract— **For provisioning large-scale online applications such as web search, social networks and advertisement systems, data centers face extreme challenges in providing low latency for short flows (that result from end-user actions) and high throughput for background flows (that are needed to maintain data consistency and structure across massively distributed systems). We propose L^{2}DCT, a practical data center transport protocol that targets a reduction in flow completion times for short flows by approximating the Least Attained Service (LAS) scheduling discipline, without requiring any changes in application software or router hardware, and without adversely affecting the long flows. L^{2}DCT can co-exist with TCP and works by adapting flow rates to the extent of network congestion inferred via Explicit Congestion Notification (ECN) marking, a feature widely supported by the installed router base. Though L^{2}DCT is deadline unaware, our results indicate that, for typical data center traffic patterns and deadlines and over a wide range of traffic load, its deadline miss rate is consistently smaller compared to existing deadline-driven data center transport protocols. L^{2}DCT reduces the mean flow completion time by up to 50% over DCTCP and by up to 95% over TCP. In addition, it reduces the completion for 99th percentile flows by 37% over DCTCP. We present the design and analysis of L^{2}DCT, evaluate its performance, and discuss an implementation built upon standard Linux protocol stack.**

I. Introduction

Data centers are now being used as a critical infrastructure for high-revenue online services such as web search, social networking, advertisement systems, and recommendation systems. Such data center applications pose demanding latency requirements and even a small fraction of a second can make a quantifiable difference in user experience thus impacting the revenue. For example, Google observed a 20% traffic reduction from an extra 500 ms of latency (introduced inadvertently), and Amazon found that every additional 100 ms of latency costs them a 1% loss in business revenue [1], [2].

Large-scale online applications are typically hosted at data centers and follow the *Partition/Aggregate* workflow pattern, in which user requests are partitioned amongst layers of worker nodes within the data center and the results from the workers are then combined by an aggregator node before a final response is sent back to the user [3], [4].

Provisioning these applications leads to data center traffic that is a mix of short and long flows. Prior work shows that most flows are short arising from end-user actions, however most bytes are contained in a very small number of background long flows that are needed to maintain data consistency and structure across massively distributed systems [3]. The timeliness of response to the end-user is determined by the short foreground flows while the quality of response is determined by both the short and the long flows. Long-lived TCP flows cause the length of the bottleneck queue to grow until packets get dropped. When long flows and latency-sensitive short flows share the same queue, short flows experience increased latency due to queue buildup by long flows [1], [5], [6].

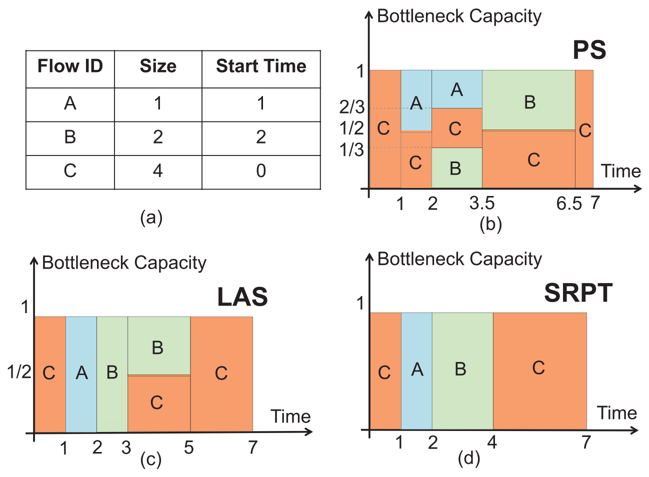
In this paper, we present L^{2}DCT (Low Latency Data Center Transport), a practical data center transport protocol that targets a reduction in the completion times for short flows. L^{2}DCT can be deployed incrementally as it can co-exist with TCP and does not require any changes to router hardware or application software. At the heart of L^{2}DCT is the additive increase and backoff mechanism for setting the transport layer window size. Under this mechanism, end hosts make use of the information inferred from Explicit Congestion Notification (ECN) marking and adjust their flow rates (by setting the window size) based on the amount of data a flow has already sent. Intuitively, conservative backoff and aggressive increase for short flows allows these flows to finish relatively quickly.

A number of transport protocols have previously been proposed for large-scale data center applications with Partition/Aggregate workflow.

One class of data center protocols approximate the processor sharing (PS) discipline by dividing the link bandwidth equally among flows [1], [7], [8]. This solution ignores the disparate requirements for short foreground and long background flows. Furthermore, it has previously been shown that although the PS discipline leads to fairness, it is far from optimal in terms of minimizing the average flow completion time (AFCT) [9].

Another class of data center protocols assign deadlines to flows and try to meet those deadlines as the main objective [4], [6]. Such protocols require changes to applications (for passing deadline and/or flow size information) and may need router hardware modifications [4]. Furthermore, there is no established basis for accurately choosing the deadlines, which are currently set based on user experience surveys [1], [4].

L^{2}DCT focuses on reducing the completion times for short flows. PDQ [10] shares a similar objective and improves the flow completion times over TCP, RCP [8] and D^3 [4]. However, it requires modifications to switch hardware and software and is incompatible with TCP, leading to practical difficulties with deployment. Indeed, a protocol can optimally minimize the AFCT by using the Shortest Remaining Processing Time (SRPT) scheduling discipline. However, SRPT requires knowledge of flow sizes (which may or may not be available), a centralized scheduler, and incurs an overhead for passing flow size information to the scheduler. L^{2}DCT overcomes all these limitations: it does not need a centralized scheduler, is compatible with TCP, does not require any software or hardware support from the routers (except for the ECN marking which is a standard feature in present-day routers [1], [6]), and is easy to implement (requires fewer than 75 lines of code change to TCP in the Linux kernel).



Flow ID	Size	Start Time
A	1	1
B	2	2
C	4	0

Fig. 1. Motivating example.

Practicality of L^{2}DCT also stems from the fact that it approximates the Least Attained Service (LAS) scheduling discipline, which does not require flow size information. LAS also targets a reduction in AFCT and closely approximates SRPT [11] even though it is not optimal, as illustrated in Figure 1. Three flows (A, B, and C), having different transfer sizes, arrive at different times. Assuming a fluid traffic model i.e., with infinitesimally small units of transmission, the progress of flows with fair sharing, SRPT, and LAS is as shown. With fair sharing or *Processor Sharing* (PS), the flows A, B, and C finish at times 3.5, 6.5, and 7, respectively and have an AFCT of $\frac{2.5+4.5+7}{3} = 4.67$. With SRPT, the AFCT becomes $\frac{1+2+7}{3} = 3.33$ and with LAS, the AFCT is $\frac{1+3+7}{3} = 3.67$.

The congestion control mechanism of L^{2}DCT approximates LAS as follows: in the face of congestion, long flows back off aggressively while short flows do so conservatively. In contrast, the additive increase of congestion window is more aggressive in case of short flows. This intuitively favors the short flows to finish quickly without causing any starvation of long flows. Furthermore, window size adjustment based on the extent of congestion, as estimated by ECN marking, allows long flows to achieve high throughput in the absence of short flows and prevents congestion collapse.

L^{2}DCT reduces the AFCT by up to 50% over DCTCP and by up to 95% over TCP. It also reduces the completion time for the 99th percentile flows by 37% over DCTCP. Though L^{2}DCT focuses on AFCT and is deadline unaware, our results also show that it would miss fewer deadlines compared to existing deadline-driven data center transport protocols, for typical data center traffic patterns and deadlines and over a wide range of traffic load. Altogether, this paper makes the following contributions:

- A data center transport protocol L^{2}DCT that targets a reduction in flow completion times, is incrementally deployable, requires no changes to the data center application code base or the router hardware, and is able to co-exist with TCP.
- Design and analysis of the congestion control mechanism used in L^{2}DCT.
- Extensive evaluation of L^{2}DCT (using at-scale simulations) in comparison with TCP and other proposed data center transport protocols under typical data center traffic patterns for large-scale online applications over a wide range of workloads, measuring the impact in: average flow completion times, proportion of deadlines missed, per flow throughput in a multihop setting, and bottleneck queue length.
- A small-scale testbed evaluation using an implementation built upon the standard Linux protocol stack.

The rest of the paper is organized as follows. We describe the details of L^{2}DCT and present its analysis in Section II. We evaluate L^{2}DCT's performance in Section III. Linux implementation and real testbed results are presented in Section IV. We discuss related work in Section V, followed by some discussion and future work in Section VI. We offer concluding remarks in Section VII.

II. L^{2}DCT Protocol

L^{2}DCT modulates the congestion window size based on *estimated* flow sizes as well as the extent of congestion in the network. Flow sizes are estimated based on the amount of data a flow has sent so far. This enables L^{2}DCT to adapt congestion window sizes in a size-aware manner without requiring flow size information and approximate scheduling disciplines such as LAS. With such size-aware congestion management, L^{2}DCT is able to significantly reduce the AFCT.

A. Congestion Avoidance Algorithm

The congestion avoidance algorithm used by L^{2}DCT has two components, one at the sender side and the other at the routers. Like DCTCP [1], a router marks all packets by setting the Congestion Experienced (CE) bits using ECN [12] when the queue length exceeds a certain threshold. L^{2}DCT senders measure the extent of network congestion by maintaining a weighted average of the fraction of marked packets, α, as:

$$\alpha = g \times F + (1 - g) \times \alpha$$

where F is the fraction of packets marked in the most recent window, and g is the weight given to new samples.

In order to realize different scheduling disciplines, such as SRPT and LAS, which prioritize flows based on their sizes, L^{2}DCT modulates the congestion window size of each flow based on α and a weight. In the context of LAS, these weights are assigned based on the amount of flow data sent so far but can differ across scheduling disciplines (which we discuss in Section III-E). These weights implicitly define priorities of a flow. Based on these weights and α, we determine k, the increase in congestion window per round-trip time (RTT) and the backoff penalty b, as follows:

$$k = w_c / w_{max} \tag{1}$$

$$b = \alpha^{w_c} \tag{2}$$

where $w_c \in [w_{min}, w_{max}]$ is the current flow weight, w_{min} is the minimum weight, and w_{max} is the maximum weight any flow can assume. We evaluate the impact of these bounds on w_c in Section III. Each flow starts by setting w_c to w_{max}. As flows send more data, w_c decreases before converging to w_{min}. Observe that since $w_c \leq w_{max}$ and $\alpha \leq 1$, therefore

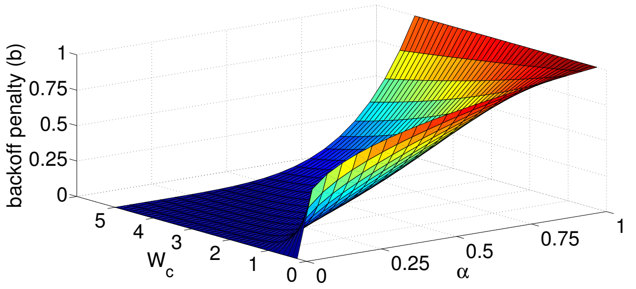


Fig. 2. Changes in backoff penalty b as a function of α and w_c.

$k \leq 1$ and $b \leq 1$, respectively. While the increase in window size per RTT can be more than one but due to burstiness in packet arrivals in data center applications [5], we limit k to at most 1, which is the increase factor used by TCP [1].

When a marked ACK (i.e., with ECN-Echo flag set) is received, L^{2}DCT uses b to reduce the window size as:

$$cwnd = cwnd \times (1 - b/2). \tag{3}$$

Note that TCP, unlike L^{2}DCT, always cuts its window size by half[1]. When no packets are CE-marked, the window size is increased as:

$$cwnd = cwnd + k. \tag{4}$$

Therefore, when congestion is high ($\alpha = 1$ and $b = 1$) the window size is reduced by half, similar to TCP. When $\alpha = 0$, and so is b, window increase depends on w_c. When α varies between 0 and 1, the window size is adapted based on k and b. In particular, flows with high weight (i.e., short flows) incur a smaller backoff penalty and apply a larger k compared to flows with smaller weights (i.e., long flows). Note that when $w_c = w_{max}$, the window size is increased by one packet, similar to TCP.

Flow weights: The weight w_c decreases with the amount of data a flow has already sent. Some possible functions for w_c include $1/s$ and e^{-s}, where s is the data sent so far. We discuss the choice of the weight function in Section III.

B. Understanding Congestion Behavior

The congestion behavior of L^{2}DCT depends on w_c and α as they impact the window increase and decrease policies. We now discuss how the choice of L^{2}DCT's window control policies allows it to achieve the following goals:

1) When short flows and long flows co-exist, the latter should relinquish bandwidth to allow the former a greater short term share of the bandwidth.
2) When only long flows are present, they should be able to achieve high throughput and not be penalized any more than regular TCP or DCTCP.
3) When congestion becomes severe (i.e., α is close to 1), all flows should converge to applying full backoff, similar to TCP, to prevent congestion collapse.

The first goal suggests that short flows should increase their window faster than long flows and backoff less. Consequently, we vary k as a function of w_c. When a new flow starts, it increases its window by 1 pkt/RTT (similar to TCP). However, as the flow transmits more packets, its weight decreases, leading to a proportional decrease in the increase/RTT. This helps in prioritizing short flows over long flows.

[1]TCP and L^{2}DCT both reduce their window size at most once per RTT.

Figure 2 shows the backoff penalty as a function of w_c and α. Observe that when $0 < w_c < 1$ (i.e., long flows), b increases rapidly even with small increases in α, and approaches 1 as α tends to 1. This implies that minor congestion causes rapid reduction in the window sizes of long flows but severe congestion (e.g. $\alpha = 1$) does not penalize such flows any more than regular TCP or DCTCP as suggested by the second goal.

When $w_c > 1$ (i.e., short flows), b increases slowly in response to increases in α, until α approaches 1, at which point b rapidly converges to 1. Therefore, minor congestion does not penalize short flows by much, which allows such flows a greater short-term share of the bandwidth to finish quickly. However, severe congestion causes a full backoff.

When congestion continues to grow in severity after long flows have backed off, then two scenarios are possible: (i) there are many short flows, who are not reducing their share of the bandwidth (ii) there may be TCP flows who are consuming bandwidth. Both these situations are handled because as α tends to 1, even short flows will throttle themselves, thus allowing TCP and other short flows to make progress. Consequently, the shortest flows will have the largest share of the bandwidth.

C. Analysis

To understand the impact of w_c and k on the steady state behavior of L^{2}DCT, we now present the analysis of L^{2}DCT in a simplified setting. We consider N long-lived flows with identical RTTs T and weight w_c, sharing a single bottleneck link of capacity C. It is further assumed that the N flows are synchronized i.e., their window dynamics (or sawtooths) are in-phase. Of course, this assumption is only realistic when N is small, however, this is the case we care about most in data centers, where responses from worker nodes are synchronized [7]. We further assume that w_c is fixed. In reality, w_c changes over time, however, this assumption still allows us to capture the impact of w_c on the protocol performance.

Due to flow synchronization, the window sizes of N flows follow identical sawtooths, and therefore, the queue size process also follows a sawtooth [13]. We are interested in determining the backoff penalty b as a function of w_c, k, and the maximum window size (W^o) as well as quantities which completely specify the queue sawtooth: the amplitude of queue oscillations (A), period of oscillations (T_C), and the maximum queue length (Q_{max}).

With synchronized flows, the queue length exceeds the marking threshold K for exactly one RTT in each period of the sawtooth, before the sources receive ECN marks and reduce their window sizes accordingly. Therefore, we compute the fraction of marked packets, α, by dividing the number of ACKs received during the last RTT by the total number of ACKs received during the full period of the sawtooth, T_C.

We now consider one of the flows and determine its backoff penalty. Let $X(W_1, W_2)$ denote the number of packets sent by a flow, while its window increases from W_1 to $W_2 > W_1$. This takes $(W_2 - W_1)/k$ RTTs[2] during which the average window size is $(W_1 + W_2)/2$.

$$X(W_1, W_2) = (W_2^2 - W_1^2)/2k$$

Let $W^o = (CT + K)/N$. This is the window size at which the queue length reaches K, and switch starts marking the packets with the CE codepoint. During the round-trip time it takes for the sender to react to these marks, another W^o packets have been sent. Hence, fraction of marked packets, α, can be calculated by,

$$\alpha = X(W^o, W^o + k)/X((W^o + k)(1 - b/2), W^o + k)$$
$$= ((W^o + k)^2 - (W^o)^2)/((W^o + k)^2 - (W^o + k)^2(1 - b/2)^2) \quad (5)$$

Simplifying and rearranging the equation gives,

$$\alpha(b - b^2/4) = (2W^o k + k^2)/(W^o + k)^2$$

Assuming b is small, we can rewrite the equation as,

$$\alpha b = (2W^o k + k^2)/(W^o + k)^2$$

Plugging the value of $b = \alpha^{w_c}$, gives us,

$$\alpha = \left[(2W^o k + k^2)/(W^o + k)^2\right]^{1/(w_c+1)} \quad (6)$$

$$and\ b = \left[(2W^o k + k^2)/(W^o + k)^2\right]^{w_c/(w_c+1)} \quad (7)$$

Note that when w_c=1 and $k = 1$, we obtain α for DCTCP [1]. The amplitude of oscillation in the window size of a single flow, D, is given by,

$$D = (W^o + k) - (W^o + k)(1 - b/2) = b(W^o + k)/2 \quad (8)$$

As there are N flows in total, A can be computed as follows,

$$A = ND = Nb(W^o + k)/2 \approx \frac{NW^o}{2} . \left(\frac{2k}{W^o}\right)^{w_c/(w_c+1)} \quad (9)$$

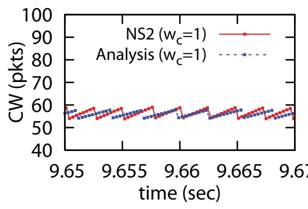
where the final expression assumes that $W^o >> k$. The period of the oscillations and the maximum queue length are,

$$T_c = D = b(W^o + k)/2 \quad (10)$$

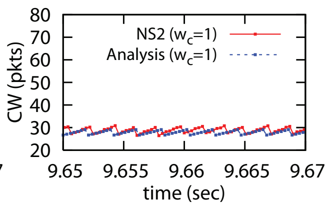
$$Q_{max} = N(W^o + k) - C \times T = K + Nk \quad (11)$$

We compared the accuracy of the above results with NS2 simulations. Figure 3 shows the results for $w_c \in \{0.5, 1\}$ and $N \in \{1, 2\}$ on a 1 Gbps link with a RTT of 300 μs. Observe that the analysis provides a fairly accurate prediction of the window dynamics when $w_c = 1$. For $w_c = 0.5$, the analysis yields larger variations in window size compared to simulations due to the continuous approximation we made. An important property, revealed by Equation 9, is that the amplitude of the queue oscillations of L^{2}DCT is in $O((C \times T)^{1/(w_c+1)})$ when N is small. In particular, the queue oscillations become independent of $C \times T$ (i.e., the BDP) when w_c is large and the ratio $1/(w_c + 1)$ approaches zero

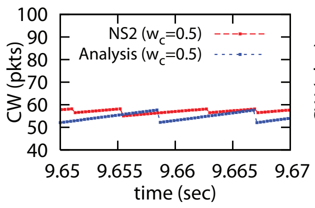
[2]Note that when $k < 1$, a flow sends W packets/RTT until the window becomes $W + 1$. However, we found the above continuous approximation to be fairly accurate compared to the precise expressions, which require solving α numerically.



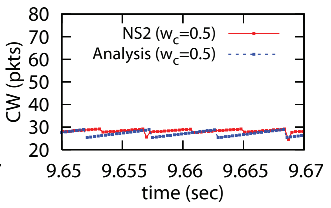
(a) $N = 1$, $w_c = 1$



(b) $N = 2$, $w_c = 1$



(c) $N = 1$, $w_c = 0.5$



(d) $N = 2$, $w_c = 0.5$

Fig. 3. Comparison of the window size dynamics predicted by the analysis with NS2 simulations for different weights w_c.

because senders backoff less. When $w_c = 1$ and N is small, the amplitude is in $O(\sqrt{C \times T})$, which is also the case for DCTCP [1]. Finally, when w_c approaches zero, the oscillations become similar to those of TCP. The lower bound we chose on w_c (i.e., $w_{min} = 0.125$) implies that the amplitude of the queue oscillations for small N in the worst case is in $O((C \times T)^{8/9})$, which is much smaller than that of TCP. This implies that we can have a small marking threshold K without losing throughput in the low statistical multiplexing regime seen in data center environments. We know that,

$$Q_{min} = Q_{max} - A \quad (12)$$

To determine the lower bound on K, we minimize (12) over N to get,

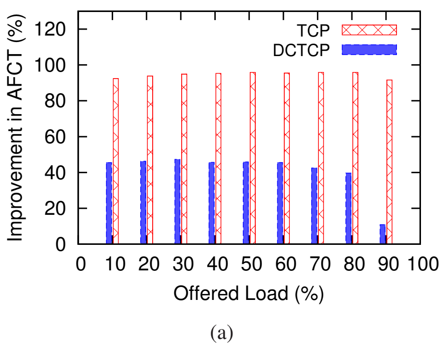
$$K > \frac{w_c^{w_c}(C \times T)}{2(w_c + 1)^{w_c+1} - w_c^{w_c}} \quad (13)$$

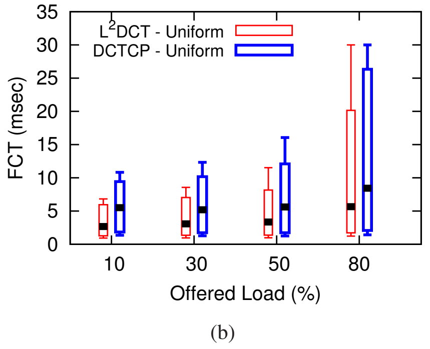
In the worst-case scenario, where $w_c = 0.125$, we get $K > (C \times T)/2$.

III. Evaluation

We evaluate the performance of L^{2}DCT using at-scale simulations and a real Linux implementation. First, we evaluate L^{2}DCT using a benchmark generated from the traffic measurement study conducted in [1]. Second, we evaluate L^{2}DCT's throughput and queuing behavior in single and multi-hop environments. Finally, we evaluate L^{2}DCT's performance when it co-exists with TCP.

We compare the performance of L^{2}DCT with DCTCP and TCP SACK with drop-tail queueing. For deadline-aware scenarios, we compare L^{2}DCT with D^{2}TCP [6], a recently proposed deadline-aware transport protocol. Unless otherwise stated, we use a single-rooted tree; a commonly used data center topology for our evaluation [1], [4], [7], [14]. In our simulations, we use 1 Gbps interfaces, round-trip propagation delay of 300 μs, and a static buffer size of 250 packets unless stated otherwise. We set the RTO$_{min}$ of all protocols to be 10 ms as suggested by previous studies [1], [14]. We set the parameters of DCTCP and L^{2}DCT to match those in [1]. In





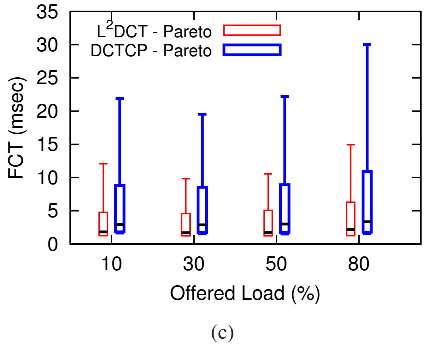


Fig. 4. Benchmark results at varying traffic loads. (a) Improvement in AFCT of L^{2}DCT over DCTCP and TCP. 1st, 5th, 50th, 95th, and 99th percentile completion times of L^{2}DCT and DCTCP with two long-lived flows along with (b) uniformly distributed flow sizes and (c) Pareto distributed flow sizes.

particular, the weighted averaging factor g is set to 1/16 and the marking threshold K to 20. For L^{2}DCT, we cap w_c between 2.5 and 0.125, except in cases where we explore the effect of varying the cap on w_c. In our evaluation, we assume that w_c, which is initially 2.5, stays constant when the amount of data sent so far is $<$ 200 KB, and then decreases linearly to 0.125 for 1 MB, and stays constant afterwards. This matches the traffic profiles observed in real data centers, where delay-sensitive traffic is generally less than 200 KB and long-lived flows are of $>$ 1 MB size [1], [5].

A. Data Center Specific Impairments

1) Benchmark Settings: We generate short query traffic with flow sizes drawn from the interval [2 KB, 98 KB] using a uniform distribution, as done in a prior study [4]. In addition, we generate two long-lived flows, which represents the 75th percentile traffic multiplexing in data center networks [1].

Figure 4 shows the flows completion time results as a function of the offered traffic load. Observe that L^{2}DCT improves the AFCT over DCTCP and TCP by up to 45% and 95%, respectively (see Figure 4(a)). The improvement in AFCT over DCTCP is at least 40% for 10-20% load, which is a realistic load in present-day data centers [4], [5], [6]. L^{2}DCT also improves both the 95th and 99th percentile of completion times by up to 37% compared to DCTCP (see Figure 4(b)).

Figure 5 shows the corresponding throughput of long background flows. Observe that for data center traffic loads of 10%, there is a difference in the throughput between L^{2}DCT and DCTCP of about 6.7%. At higher loads, more short flows arrive per second, which increases this difference. As we discuss in Section III-B.3, throughput of long flows can be increased in these scenarios by a corresponding reduction in flow completion times by adjusting the cap on w_c.

2) Pareto Distributed Traffic: We generated flows with sizes drawn from a Pareto distribution with mean 50 KB and shape 1.2. This yields flow sizes that capture realistic data center workloads [1]. In these settings, L^{2}DCT improves over DCTCP at all loads as shown in Figure 4(c), with 99th percentile of FCT improved by up to 53%.

3) Incast Behavior: With L^{2}DCT, short flows behave more aggressively to increase their share of the network bandwidth. Consequently, L^{2}DCT may exacerbate Incast. To study the

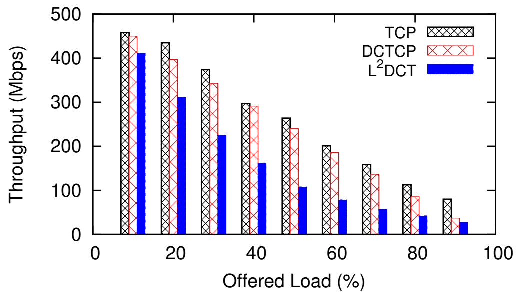
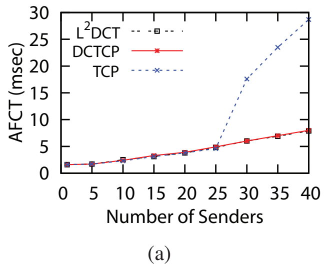


Fig. 5. Throughput of long flows under the benchmark settings.



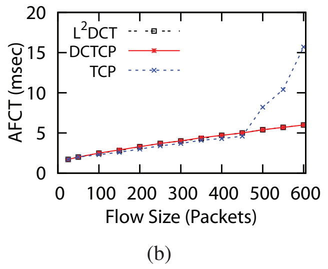


Fig. 6. AFCT under the Incast scenario. (a) Varying number of senders with Uniform flow sizes with 25 KB mean. (b) Varying flow size with client requests from 5 different servers.

Incast behavior, we assume a topology where multiple servers send data (responses to queries) to one aggregator simultaneously through a single bottleneck link.

a) Impact of Number of Sending Servers: We vary the number of simultaneous senders from 1 to 40 as done in [1] and flow sizes are generated uniformly at random with a mean of 25 KB. Even though RTO$_{min}$ is set to 10 ms, TCP performance still degrades when the number of senders increases beyond 25. In contrast, even though L^{2}DCT uses more aggressive parameters for short flows, its stringent marking policy allows Incast to be mitigated as shown in Figure 6(a).

b) Impact of Flow Size: We assume 5 sending servers and vary the mean flow size (i.e., the size of response from each server) from 50 KB to 600 KB. As the size of the flow increases, AFCT for L^{2}DCT and DCTCP degrades gracefully. However, AFCT with TCP starts degrading when the mean flow size is increased beyond 400 KB (see Figure 6(b)).

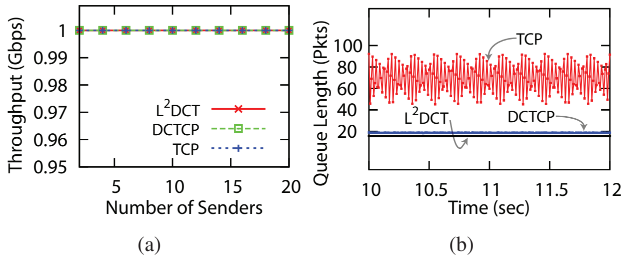


Fig. 7. Performance of long-lived L^{2}DCT flows. (a) Throughput achieved by flows. (b) Queue length dynamics for two long-lived flows.

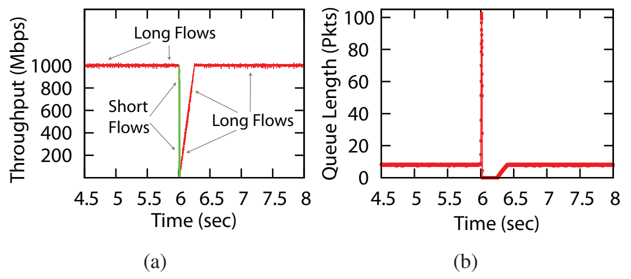


Fig. 8. Performance of two long L^{2}DCT flows when a sudden burst of short flows arrives at time $t = 6$ s. (a) Throughput (b) Queue length dynamics.

B. L^{2}DCT Specific Testing

We now evaluate how L^{2}DCT performs as a congestion control protocol.

1) Single Bottleneck Scenario: In this scenario, we evaluate the throughput performance of long-lived L^{2}DCT flows and perform comparison with TCP and DCTCP. Figure 7(a) shows the throughput of L^{2}DCT, DCTCP, and TCP flows as a function of the number of senders, each generating a single long-lived flow. Observe that all protocols achieve $\sim$100% throughput. However, L^{2}DCT and DCTCP maintain much smaller queue length compared to TCP as shown by Figure 7(b). L^{2}DCT is able to maintain small queues due to gentle backoffs based on the extent of network congestion.

2) Effect of Sudden Short Flow Bursts: We now consider the scenario where a sudden burst of short flows arrive while long-lived flows are active. This is a fairly common scenario in data center environments.

Figure 8(a) shows the throughput of flows when 50 short flows, each of size 50KB, arrive simultaneously in the presence of an ongoing long-lived flow. Observe that L^{2}DCT quickly adapts to the sudden bursts of short flows converging to full link utilization afterwards. In particular, the arrival of short flows increases the queue occupancy, causing the long flows to backoff more than short flows (see Figure 8(b)). When short flows complete, long flow quickly grabs the entire bottleneck capacity. Note that the low queue occupancy with L^{2}DCT implies that there is more room in the queue for absorbing packet bursts. Moreover, long flows (> 1 MB) use $k = 0.05$, which means they take $\sim$200 ms to achieve 1 Gbps.

3) Impact of the Weight Function: The performance achieved by L^{2}DCT depends on the weights assigned to flows. Extremely low or high values may cause undesirable behavior due to which we cap w_c to be within ($w_{max} = 2.5$, $w_{min} = 0.125$). In this section, we evaluate the impact of varying w_{max} and w_{min}.

Figures 9(a) and 9(b) show the impact of varying w_{max} on the AFCT of short flows and the throughput of long-lived flows when the offered load is 20%. Observe that increasing w_{max} reduces the AFCT as well as the throughput of long flows. This happens because increasing w_{max} makes short flows more aggressive, which leads to higher values for α. Since long flows backoff significantly even for small α, this reduces their throughput. Next, we vary w_{min}. Observe that the AFCT of short flows and the throughput of long flow decreases when w_{min} is decreased as shown in Figures 9(c) and 9(d). This happens because decreasing w_{min}, increases the backoff factor and lowers the additive increase for long flows, which reduces their throughput in the presence of short flows. Therefore, we set w_{max} to 2.5 and w_{min} to 0.125 to achieve a compromise between the performance of short and long flows.

4) Deadline Constrained Flows: We now evaluate the performance of L^{2}DCT when flows have deadlines associated with them and compare its performance with D^{2}TCP [6], a recently proposed deadline-aware protocol.

We replicate the traffic settings of Section III-A.1, and determine the number of flows missing their deadlines. To generate deadlines, we use the same approach as employed in [4]. In particular, flow deadlines are generated using the exponential distribution with mean 40 msec (tight deadlines), 60 msec (moderate deadlines) and 80 msec (lax deadlines). Figure 10(a), 10(b), and 10(c) show that L^{2}DCT outperforms all protocols across a range of traffic loads including D^{2}TCP, which specifically accounts for flow deadlines. Since deadlines are typically associated with short flows, these results suggest that a deadline agnostic protocol, which minimizes completion times, can achieve better performance than deadline-aware protocols.

C. Multiple Bottleneck Scenario

To evaluate L^{2}DCT's performance in a multi-hop, multi-bottleneck environment, we use the topology shown in Figure 11(a). A total of 30 senders (where S_1, S_2, and S_3 represent a set of senders) and 11 receivers are used in this case. We generate long-lived flows, $S_1 \rightarrow R_1$, $S_2 \rightarrow R_1$ and $S_3 \rightarrow R_1$. Note that S_1 competes with both S_2 and S_3 but at different links. Therefore, we expect the throughput of S_1 to be lower than S_2 and S_3 because the throughput of TCP flows is inversely proportional to the number of bottlenecks it traverses [15]. Figure 11(b) shows the average per-flow throughput in each sender set. Observe that with L^{2}DCT, S_1 achieves higher throughput compared to TCP and DCTCP and is also able to maintain better fairness.

D. Co-existence with TCP

TCP is a widely used congestion control protocol in cloud data centers [1], [7]. Therefore, we now evaluate L^{2}DCT's performance when it co-exists with TCP. Using a single bottleneck topology, we generate multiple long-lived flows and observe the effect of varying k and w_c on the relative throughput of L^{2}DCT and TCP.

Figures 12(a) and 12(b) show the throughput as a function of k for two cases: (a) one flow is generated by each protocol, and (b) each protocol generates two flows. Observe that as

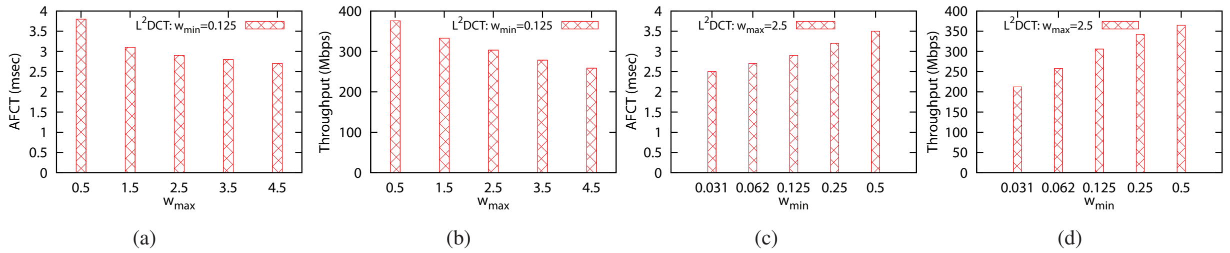
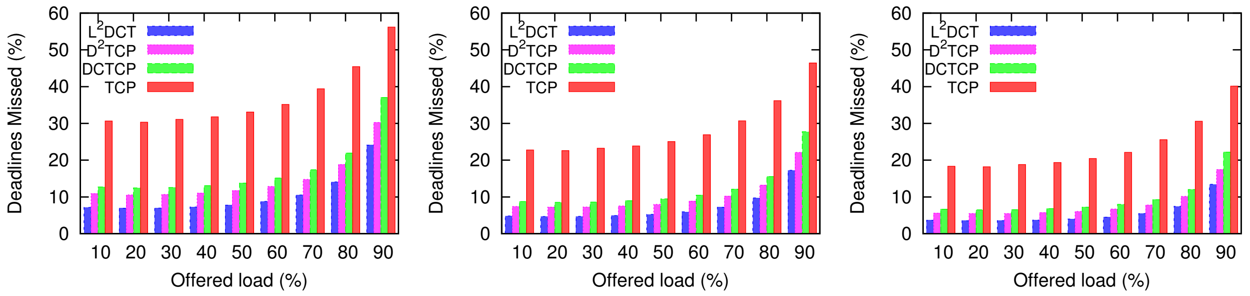


Fig. 9. Impact of varying w_{max} and w_{min} on the AFCT of short flows and the throughput performance of long-lived flows.



(a) Tight Deadlines (b) Moderate Deadlines (c) Lax Deadlines

Fig. 10. Deadlines missed by various protocols. (a) Tight deadlines (40 ms). (b) Moderate deadlines (60 ms). (c) Lax (80 ms). Observe that L^{2}DCT misses the least number of deadlines.

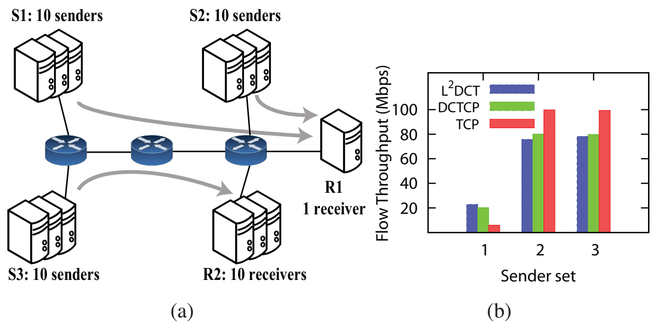


Fig. 11. Throughput performance under multiple bottleneck topology

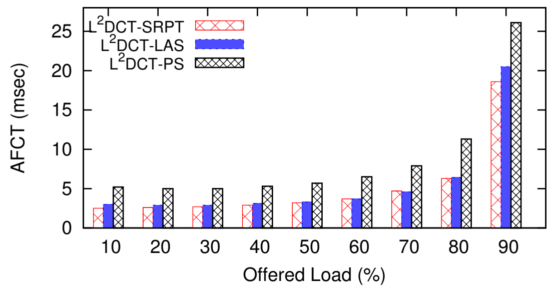


Fig. 13. AFCT under different scheduling disciplines

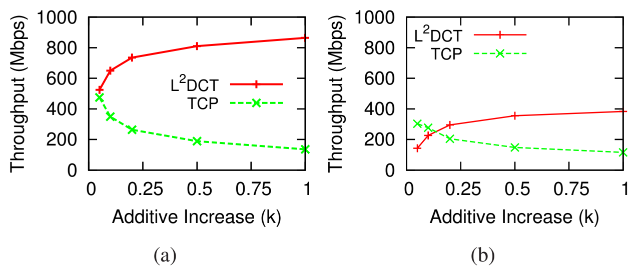


Fig. 12. Throughput of long-lived TCP and L^{2}DCT flows when they co-exist. Each protocol generates (a) one flow and (b) two flows.

k increases, L^2DCT flows become more aggressive, thus achieving much higher throughput than TCP flows. Since there are very few concurrent long flows in data centers [1], using $k < 0.25$ can provide suitable fairness between TCP and L^{2}DCT flows. Note that L^{2}DCT uses $k = 0.05$ (as $w_{min} = 0.125$) for long flows.

E. *Realizing Other Scheduling Policies*

In this section, we show how L^{2}DCT can approximate a variety of scheduling policies such as SRPT and PS by adapting the flow weights. To realize SRPT, the weights are now adapted based on the remaining flow data instead of the data sent so far. For PS, we set $w_c = 1$ for all flows. Figure 13 shows that LAS is a good approximation of SRPT. PS, however, results in a much larger AFCT.

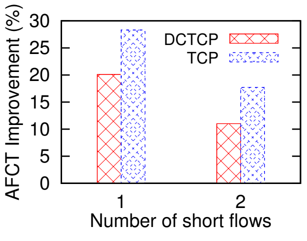
IV. Real Testbed Implementation

We built a small-scale testbed to evaluate the performance of L^{2}DCT in real network settings.

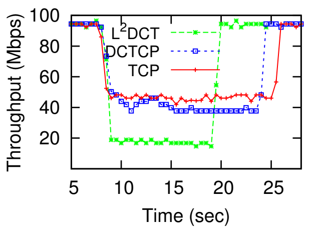
A. *Linux Implementation Details*

L^{2}DCT requires very few changes at the end-hosts and none at the routers. It is implemented as a kernel module in Linux 2.6.38, which supports pluggable congestion control. L^{2}DCT inherits important features of TCP such as retransmission and fast recovery mechanism. We used DCTCP's end host implementation for building L^{2}DCT.

a) Marking at the Switch: For realizing L^{2}DCT's switch, we use the RED queue implementation in Linux. We set the



(a)



(b)

Fig. 14. Linux results for L^{2}DCT (a) AFCT and (b) Long flow throughput.

low and high thresholds of RED queue to K and perform marking based on the instantaneous rather than the average queue length as done in RED.

b) Sender-Side Modifications: L^{2}DCT introduces changes in the additive increase and multiplicative decrease parameters used by TCP. In addition, it computes the fraction of marked packets to infer the degree of congestion at the bottleneck. L^{2}DCT uses an array to hold pre-computed values of the backoff penalty b for each α and weight w_c in order to avoid floating point calculations.

c) Receiver ECN Echo Mechanism: L^{2}DCT introduces no changes at the receiver end and its ECN echo mechanism is the same as DCTCP.

B. Linux Evaluation

Our testbed comprises of three Linux machines, each equipped with a 100 Mbps Realtek ethernet card. One machine acts as a client, one as a server and the other as a switch. We set K at 8 packets. We compare L^{2}DCT's performance with DCTCP and TCP NewReno with ECN support.

We use Iperf for traffic generation. Due to the timer resolution limitations in Iperf, we generate and give maximum weight to flows of length 100 MB and adapt the weight assignment policy accordingly. Note that this is just for the validation of our implementation and does not represent actual data center workloads.

We start a single long-lived flow in the background and generate multiple short flows simultaneously. We repeat each experiment 10 times for each number of flows. Figure 14(a) shows the improvement in AFCT of L^{2}DCT over DCTCP and TCP as a function of the number of *simultaneous* short flows. Note that this represents a very high load scenario as each flow has 100 MB of data to send. Observe that L^{2}DCT improves the AFCT by up to 20% and 29% over DCTCP and TCP, respectively. Figure 14(b) shows the corresponding throughput of the long flow when one short flow arrives. Observe that with L^{2}DCT, the long flow throughput reduces to $\sim$20 Mbps whereas the short flow gets $\sim$80 Mbps. This allows the short flow to finish quickly, thus causing the long flow to obtain maximum throughput sooner than other protocols. Since DCTCP and TCP are fair sharing protocols, short flows obtain $\sim$50 Mbps of throughput with them. This causes the DCTCP and TCP short flows to finish 4 s and 5.5 s later, respectively.

V. Related Work

The relevant literature on congestion control, scheduling, and reducing latency is vast. Therefore, in this section, we only summarize some of the most relevant works.

CUBIC [16], BMCC [17], XCP [18], and delay-based congestion control protocols, such as FAST [19] and CTCP [20], all successfully improve performance in high BDP networks. However, these protocols approximate fair sharing and thus are sub-optimal in terms of completion times. Rate Control Protocol (RCP) improves the AFCT by reducing the startup latency of flows [8]. However, RCP is also a fair sharing protocol and requires router hardware modifications [1].

In [21], authors propose HULL which is based on capping utilization at less than link capacity. By sacrificing some amount (e.g., 10%) of bandwidth, HULL can reduce the average and tail latencies. Our work is complementary to HULL. In particular, L^{2}DCT can be combined with their proposal to further reduce completion times. However, it is useful to note that L^{2}DCT reduces completion times without sacrificing link capacity. In [22], Zats et al. propose DeTail, an in-network congestion management mechanism that reduces the flow completion time *tail* in data center environments. However, DeTail does not target AFCT. Unlike DeTail, L^{2}DCT can save up to $\sim$45% over DCTCP and $\sim$95% over TCP. In addition, it reduces the completion time of almost every flow.

Earliest Deadline First (EDF) is provably optimal when individual packets are associated with deadlines. However, when associated with *flows*, applying EDF to individual packets is not only suboptimal but can increase network congestion [4].

QCN, an optional standard for Ethernet, uses multibit feedback from the switches to reduce recovery time during congestion. However, QCN cannot span beyond L2 domain limiting its scope of application [1].

To approximate LAS, one could use priority queuing at the switches. However, prior studies show that using two-level priorities, TCP/RCP with priority queuing suffer from high loss rate and falls behind D^3 [4]. Further, increasing the priority does not significantly improve performance as flows within each class may have widely different sizes and yet they are not differentiated. Consequently, large number of priority classes are needed, however, switches nowadays provide only a small number of classes, usually no more than ten [4], [10].

Yang et al. [23] proposed TCP SAReno, which adapts AIMD parameters based on the residual flow size assuming droptail queuing. First, it uses a small number of classes and therefore, faces the same issues as TCP with priority queuing. Second, it uses fixed parameters for each class, and thus considerably degrades the performance of long flows even in the absence of short flows. Zieglar et. al. [24] also proposed to dynamically adapt AIMD parameters for improving the startup latency of short flows. However, with their protocol, a short flow achieves no higher throughput than long flows for the protocol to be incentive-compatible with TCP.

Several recent works, such as [25], show the benefit of using multi-path TCP, ranging from improved network utilization to better reliability. However, developing a multi-path version of L^{2}DCT is part of future work.

VI. Discussion and Future Work

Fairness. One may argue that the performance gains of L^2DCT over other protocols are due to the fact that it unfairly penalizes long flows. It turns out that the performance improvement over fair sharing protocols does not usually come at the expense of long jobs. Bansal et al. [9] showed that with SRPT, at least 99% of the jobs have smaller completion times compared with fair sharing[3]. Moreover, this percentage increases even further when the traffic load is less than half. Typical data center workloads are generally less than 50% [6]. Further, if desired, an operator can always set the weight of flows to achieve a wide range of bandwidth sharing criteria, including fairness. For instance, to achieve fair sharing, an operator could set $w_c = 1$ for all flows.

Gaming the System. One could ask whether users will have an incentive to improve the completion time of their flows by splitting them into smaller flows. While the incentive is greater in case of L^2DCT compared to fair sharing protocols, similar issue also arises in TCP and RCP, where users may achieve higher aggregate throughput by splitting a flow into smaller ones, as well as in D^3, where users may request a higher rate than the flow actually needs. With PDQ, however, the incentive may be even greater as PDQ does preemption of flows whereas L^2DCT does not. To address this, users with multiple flows can be penalized by changing their weights. However, designing a scheme to achieve this end remains a future work. We would like to point out that in data center environments, connectivity to the external Internet is typically managed through application proxies that effectively separate internal traffic from external, therefore, issues of fairness with conventional TCP outside are irrelevant [1].

When flow completion time is not the priority. For real-time applications such as VoIP, flow completion time is not the best metric. For such applications, a constant w_c can be assigned, making it equivalent to a DCTCP flow.

Stability. If all flows are short and thus demand a high share of the bandwidth, network overload may occur. However, L^2DCT's backoff mechanism guards against such overload on two fronts: (1) When α approaches one, L^2DCT defaults to the same backoff as TCP, therefore, L^2DCT's worst case stability is similar to that of TCP. (2) We limit the maximum value of w_c to 2.5, which limits the aggressiveness of short flows.

VII. Conclusion

We design and implement L^2DCT, a data center transport protocol, which targets minimizing the flow completion times by approximating the LAS scheduling discipline. L^2DCT can co-exist with TCP and delivers high application throughput by meeting more deadlines than existing protocols. For a wide range of data center workload scenarios L^2DCT provides up to 45% and 95% reduction in AFCT over DCTCP and TCP. The 99^{th} percentile improvement in flow completion times over DCTCP is up to 37% and 53% when the flow size distribution is uniform and Pareto, respectively. Though L^2DCT is deadline unaware, owing to its goal of minimizing flow completion times, it would miss up to about 35% fewer deadline compared to a recent deadline-aware protocol D^2TCP for tight as well as lax deadlines. These improvements come at a cost of small throughput degradation for background flows; although this does not impact the end-user response time, it can still be easily addressed by adjusting w_c. However, data center operators must evaluate the impact of using L^2DCT using their own traffic and network profile. We also implemented L^2DCT on Linux demonstrating that it does not require any flow specific information from applications and can be easily deployed without requiring any additional hardware and software support.

[3]Assuming M/G/1/SRPT queueing model with heavy-tailed distributions.

References

[1] M. Alizadeh, A. Greenberg, D. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center tcp (dctcp)," in *SIGCOMM'10.*

[2] T. Hoff, "Latency is everywhere and it costs you sales how to crush it," July 2009, http://highscalability.com/blog/2009/7/25/latency-is-everywhere-and-it-costs-you-sales-how-to-crush-it.html.

[3] D. Abts and B. Felderman, "A guided tour of data-center networking," *Commun. ACM*, vol. 55, no. 6, pp. 44–51, June 2012.

[4] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowstron, "Better never than late: Meeting deadlines in datacenter networks," in *SIGCOMM'11.*

[5] T. Benson, A. Akella, and D. Maltz, "Network traffic characteristics of data centers in the wild," in *IMC'10.*

[6] B. Vamanan, J. Hasan, and T. N. Vijaykumar, "Deadline-aware datacenter tcp (d2tcp)," in *SIGCOMM'12.*

[7] H. Wu, Z. Feng, C. Guo, and Y. Zhang, "Ictcp: Incast congestion control for tcp in data center networks," in *Co-Next'10.*

[8] N. Dukkipati, M. Kobayashi, R. Zhang-Shen, and N. McKeown, "Processor sharing flows in the internet," in *IWQoS*, 2005.

[9] N. Bansal and M. Harchol-Balter, "Analysis of srpt scheduling: investigating unfairness," in *SIGMETRICS'01.*

[10] C.-Y. Hong, M. Caesar, and P. B. Godfrey, "Finishing flows quickly with preemptive scheduling," in *SIGCOMM'12.*

[11] I. A. Rai, G. Urvoy-Keller, and E. W. Biersack, "Analysis of las scheduling for job size distributions with high variance," in *SIGMETRICS'03.*

[12] K. K. Ramakrishnan and S. Floyd, "The addition of explicit congestion notification (ECN) to IP," in *IETF RFC 3168*, Sep 2001.

[13] G. Appenzeller, I. Keslassy, and N. McKeown, "Sizing router buffers," in *SIGCOMM'04.*

[14] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. Andersen, G. Ganger, G. Gibson, and B. Mueller, "Safe and effective fine-grained tcp retransmissions for datacenter communication," in *SIGCOMM'09.*

[15] Y. Xia, L. Subramanian, I. Stoica, and S. Kalyanaraman, "One more bit is enough," in *SIGCOMM'05.*

[16] I. Rhee and L. Xu, "CUBIC: A new TCP-friendly high-speed TCP variant," in *PFLDNet'05*, 2005.

[17] I. A. Qazi, L. L. H. Andrew, and T. Znati, "Congestion control with multipacket feedback," in *IEEE/ACM Trans. on Networking*, 2012.

[18] D. Katabi, M. Handley, and C. Rohrs, "Internet congestion control for high bandwidth-delay product networks," in *SIGCOMM'01.*

[19] C. Jin, D. Wei, and S. Low, "FAST TCP: Motivation, architecture, algorithms and performance," in *INFOCOM'04.*

[20] K. Tan, J. Song, Q. Zhang, and M. Sridharan, "A compound TCP approach for high-speed and long distance networks," in *INFOCOM'06.*

[21] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda, "Less is more: Trading a little bandwidth for ultra-low latency in the data center," in *NSDI'12.*

[22] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz, "DeTail: Reducing the Flow Completion Time Tail in Datacenter Networks," in *SIGCOMM'12.*

[23] S. Yang and G. De Veciana, "Enhancing both network and user performance for networks supporting best effort traffic," *IEEE/ACM Trans. on Networking*, 2004.

[24] T. Ziegler, H. Tran, and E. Hasenleithner, "Improving perceived web performance by size based congestion control," *NETWORKING'04.*

[25] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley, "Improving datacenter performance and robustness with multipath tcp," in *SIGCOMM'03.*