Data Center TCP (DCTCP)

Mohammad Alizadeh[‡†], Albert Greenberg[†], David A. Maltz[†], Jitendra Padhye[†],
Parveen Patel[†], Balaji Prabhakar[‡], Sudipta Sengupta[†], Murari Sridharan[†]

[†]Microsoft Research [‡]Stanford University
{albert, dmaltz, padhye, parveenp, sudipta, muraris}@microsoft.com
{alizade, balaji}@stanford.edu

ABSTRACT

Cloud data centers host diverse applications, mixing workloads that require small predictable latency with others requiring large sustained throughput. In this environment, today's state-of-the-art TCP protocol falls short. We present measurements of a 6000 server production cluster and reveal impairments that lead to high application latencies, rooted in TCP's demands on the limited buffer space available in data center switches. For example, bandwidth hungry "background" flows build up queues at the switches, and thus impact the performance of latency sensitive "foreground" traffic.

To address these problems, we propose DCTCP, a TCP-like protocol for data center networks. DCTCP leverages Explicit Congestion Notification (ECN) in the network to provide multi-bit feedback to the end hosts. We evaluate DCTCP at 1 and 10Gbps speeds using commodity, shallow buffered switches. We find DCTCP delivers the same or better throughput than TCP, while using 90% less buffer space. Unlike TCP, DCTCP also provides high burst tolerance and low latency for short flows. In handling workloads derived from operational measurements, we found DCTCP enables the applications to handle 10X the current background traffic, without impacting foreground traffic. Further, a 10X increase in foreground traffic does not cause any timeouts, thus largely eliminating incast problems.

Categories and Subject Descriptors: C.2.2 [Computer-Communication Networks]: Network Protocols
General Terms: Measurement, Performance
Keywords: Data center network, ECN, TCP

1. INTRODUCTION

In recent years, data centers have transformed computing, with large scale consolidation of enterprise IT into data center hubs, and with the emergence of cloud computing service providers like Amazon, Microsoft and Google. A consistent theme in data center design has been to build highly available, highly performant computing and storage infrastructure using low cost, commodity components [16]. A corresponding trend has also emerged in data center networks. In particular, low-cost switches are common at the top of the rack, providing up to 48 ports at 1Gbps, at a price point under $2000 — roughly the price of one data center server. Several recent research proposals envision creating economical, easy-to-manage data centers using novel architectures built atop these commodity switches [2, 12, 15].

Is this vision realistic? The answer depends in large part on how well the commodity switches handle the traffic of real data center applications. In this paper, we focus on soft real-time applications, supporting web search, retail, advertising, and recommendation systems that have driven much data center construction. These applications generate a diverse mix of short and long flows, and require three things from the data center network: low latency for short flows, high burst tolerance, and high utilization for long flows.

The first two requirements stem from the *Partition/Aggregate* (described in §2.1) workflow pattern that many of these applications use. The near real-time deadlines for end results translate into latency targets for the individual tasks in the workflow. These targets vary from ∼10ms to ∼100ms, and tasks not completed before their deadline are cancelled, affecting the final result. Thus, *application requirements for low latency directly impact the quality of the result returned and thus revenue.* Reducing network latency allows application developers to invest more cycles in the algorithms that improve relevance and end user experience.

The third requirement, high utilization for large flows, stems from the need to continuously update internal data structures of these applications, as the freshness of the data also affects the quality of the results. Thus, high throughput for these long flows is as essential as low latency and burst tolerance.

In this paper, we make two major contributions. First, we measure and analyze production traffic (>150TB of compressed data), collected over the course of a month from ∼6000 servers (§2), extracting application patterns and needs (in particular, low latency needs), from data centers whose network is comprised of commodity switches. Impairments that hurt performance are identified, and linked to properties of the traffic and the switches.

Second, we propose Data Center TCP (DCTCP), which addresses these impairments to meet the needs of applications (§3). DCTCP uses Explicit Congestion Notification (ECN), a feature already available in modern commodity switches. We evaluate DCTCP at 1 and 10Gbps speeds on ECN-capable commodity switches (§4). We find DCTCP successfully supports 10X increases in application foreground and background traffic in our benchmark studies.

The measurements reveal that 99.91% of traffic in our data center is TCP traffic. The traffic consists of query traffic (2KB to 20KB in size), delay sensitive short messages (100KB to 1MB), and throughput sensitive long flows (1MB to 100MB). The query traffic experiences the incast impairment, discussed in [32, 13] in the context of storage networks. However, the data also reveal new impairments unrelated to incast. Query and delay-sensitive short messages experience long latencies due to long flows consuming

SIGCOMM'10, August 30–September 3, 2010, New Delhi, India.

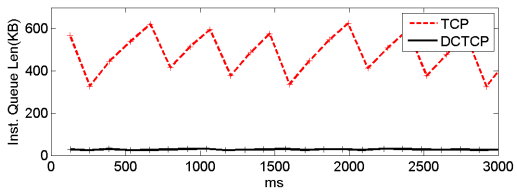


Figure 1: Queue length measured on a Broadcom Triumph switch. Two long flows are launched from distinct 1Gbps ports to a common 1Gbps port. Switch has dynamic memory management enabled, allowing flows to a common receiver to dynamically grab up to 700KB of buffer.

some or all of the available buffer in the switches. Our key learning from these measurements is that to meet the requirements of such a diverse mix of short and long flows, *switch buffer occupancies need to be persistently low, while maintaining high throughput for the long flows.* DCTCP is designed to do exactly this.

DCTCP combines Explicit Congestion Notification (ECN) with a novel control scheme at the sources. It extracts multibit feedback on congestion in the network from the single bit stream of ECN marks. Sources estimate the fraction of marked packets, and use that estimate as a signal for the extent of congestion. This allows DCTCP to operate with very low buffer occupancies while still achieving high throughput. Figure 1 illustrates the effectiveness of DCTCP in achieving full throughput while taking up a very small footprint in the switch packet buffer, as compared to TCP.

While designing DCTCP, a key requirement was that it be implementable with mechanisms in existing hardware — meaning our evaluation can be conducted on physical hardware, and the solution can be deployed to our data centers. Thus, we did not consider solutions such as RCP [6], which are not implemented in any commercially-available switches.

We stress that DCTCP is designed for the data center environment. In this paper, we make no claims about suitability of DCTCP for wide area networks. The data center environment [19] is significantly different from wide area networks. For example, round trip times (RTTs) can be less than 250μs, in absence of queuing. Applications simultaneously need extremely high bandwidths and very low latencies. Often, there is little statistical multiplexing: a single flow can dominate a particular path. At the same time, the data center environment offers certain luxuries. The network is largely homogeneous and under a single administrative control. Thus, backward compatibility, incremental deployment and fairness to legacy protocols are not major concerns. Connectivity to the external Internet is typically managed through load balancers and application proxies that effectively separate internal traffic from external, so issues of fairness with conventional TCP are irrelevant.

We do not address the question of how to apportion data center bandwidth between internal and external (at least one end point outside the data center) flows. The simplest class of solutions involve using Ethernet priorities (Class of Service) to keep internal and external flows separate at the switches, with ECN marking in the data center carried out strictly for internal flows.

The TCP literature is vast, and there are two large families of congestion control protocols that attempt to control queue lengths: (i) Delay-based protocols use increases in RTT measurements as a sign of growing queueing delay, and hence of congestion. These protocols rely heavily on accurate RTT measurement, which is susceptible to noise in the very low latency environment of data centers. Small noisy fluctuations of latency become indistinguishable from congestion and the algorithm can over-react. (ii) Active Queue Management (AQM) approaches use explicit feedback from congested switches. The algorithm we propose is in this family.

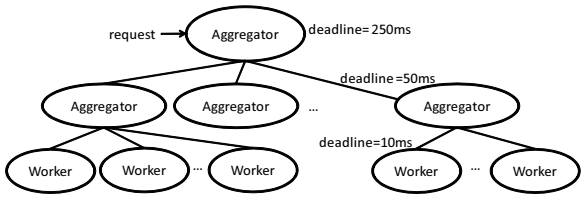


Figure 2: The partition/aggregate design pattern

Having measured and analyzed the traffic in the cluster and associated impairments in depth, we find that DCTCP provides all the benefits we seek. DCTCP requires only 30 lines of code change to TCP, and the setting of a single parameter on the switches.

2. COMMUNICATIONS IN DATA CENTERS

To understand the challenges facing data center transport protocols, we first describe a common application structure, Partition/Aggregate, that motivates why latency is a critical metric in data centers. We measure the synchronized and bursty traffic patterns that result from these application structure, and identify three performance impairments these patterns cause.

2.1 Partition/Aggregate

The Partition/Aggregate design pattern shown in Figure 2 is the foundation of many large scale web applications. Requests from higher layers of the application are broken into pieces and farmed out to workers in lower layers. The responses of these workers are aggregated to produce a result. Web search, social network content composition, and advertisement selection are all based around this application design pattern. For interactive, soft-real-time applications like these, latency is the key metric, with total permissible latency being determined by factors including customer impact studies [21]. After subtracting typical Internet and rendering delays, the "backend" part of the application is typically allocated between 230-300ms. This limit is called an all-up SLA.

Many applications have a multi-layer partition/aggregate pattern workflow, with lags at one layer delaying the initiation of others. Further, answering a request may require iteratively invoking the pattern, with an aggregator making serial requests to the workers below it to prepare a response (1 to 4 iterations are typical, though as many as 20 may occur). For example, in web search, a query might be sent to many aggregators and workers, each responsible for a different part of the index. Based on the replies, an aggregator might refine the query and send it out again to improve the relevance of the result. Lagging instances of partition/aggregate can thus add up to threaten the all-up SLAs for queries. Indeed, we found that latencies run close to SLA targets, as developers exploit all of the available time budget to compute the best result possible.

To prevent the all-up SLA from being violated, worker nodes are assigned tight deadlines, usually on the order of 10-100ms. *When a node misses its deadline, the computation continues without that response, lowering the quality of the result.* Further, high percentiles for worker latencies matter. For example, high latencies at the 99.9^{th} percentile mean lower quality results or long lags (or both) for at least 1 in 1000 responses, potentially impacting large numbers of users who then may not come back. Therefore, latencies are typically tracked to 99.9^{th} percentiles, and deadlines are associated with high percentiles. Figure 8 shows a screen shot from a production monitoring tool, tracking high percentiles.

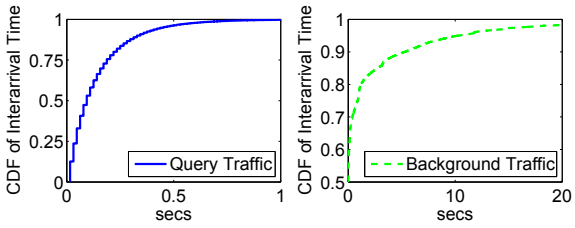


Figure 3: Time between arrival of new work for the Aggregator (queries) and between background flows between servers (update and short message).

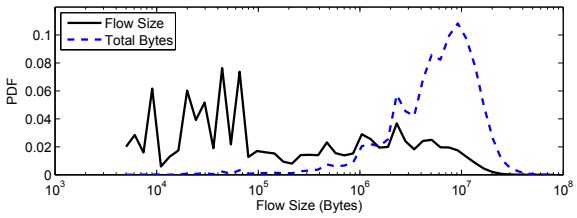


Figure 4: PDF of flow size distribution for background traffic. PDF of Total Bytes shows probability a randomly selected byte would come from a flow of given size.

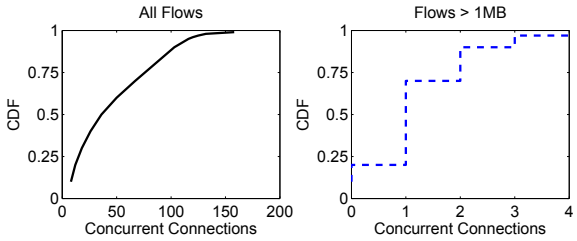


Figure 5: Distribution of number of concurrent connections.

With such tight deadlines, network delays within the data center play a significant role in application design. Many applications find it difficult to meet these deadlines using state-of-the-art TCP, so developers often resort to complex, ad-hoc solutions. For example, our application carefully controls the amount of data each worker sends and adds jitter. Facebook, reportedly, has gone to the extent of developing their own UDP-based congestion control [29].

2.2 Workload Characterization

We next measure the attributes of workloads in three production clusters related to web search and other services. The measurements serve to illuminate the nature of data center traffic, and they provide the basis for understanding why TCP behaves poorly and for the creation of benchmarks for evaluating DCTCP.

We instrumented a total of over 6000 servers in over 150 racks. The three clusters support soft real-time *query traffic*, integrated with urgent *short message traffic* that coordinates the activities in the cluster and continuous *background traffic* that ingests and organizes the massive data needed to sustain the quality of the query responses. We use these terms for ease of explanation and for analysis, the developers do not separate flows in simple sets of classes. The instrumentation passively collects socket level logs, selected packet-level logs, and app-level logs describing latencies – a total of about 150TB of compressed data over the course of a month.

Each rack in the clusters holds 44 servers. Each server connects to a Top of Rack switch (ToR) via 1Gbps Ethernet. The ToRs are shallow buffered, shared-memory switches; each with 4MB of buffer shared among 48 1Gbps ports and two 10Gbps ports.

Query Traffic. *Query traffic* in the clusters follows the Partition/Aggregate pattern. The query traffic consists of very short, latency-critical flows, with the following pattern. A high-level aggregator (HLA) partitions queries to a large number of mid-level aggregators (MLAs) that in turn partition each query over the 43 other servers in the same rack as the MLA. Servers act as both MLAs and workers, so each server will be acting as an aggregator for some queries at the same time it is acting as a worker for other queries. Figure 3(a) shows the CDF of time between arrivals of queries at mid-level aggregators. The size of the query flows is extremely regular, with queries from MLAs to workers being 1.6KB and responses from workers to MLAs being 1.6 to 2KB.

Background Traffic. Concurrent with the query traffic is a complex mix of *background traffic*, consisting of both large and small flows. Figure 4 presents the PDF of background flow size, illustrating how most background flows are small, but most of the bytes in background traffic are part of large flows. Key among background flows are large, 1MB to 50MB, *update* flows that copy fresh data to the workers and time-sensitive *short message* flows, 50KB to 1MB in size, that update control state on the workers. Figure 3(b) shows the time between arrival of new background flows. The inter-arrival time between background flows reflects the superposition and diversity of the many different services supporting the application: (1) the variance in interarrival time is very high, with a very heavy tail; (2) embedded spikes occur, for example the 0ms inter-arrivals that explain the CDF hugging the y-axis up to the 50^{th} percentile; and (3) relatively large numbers of outgoing flows occur periodically, resulting from workers periodically polling a number of peers looking for updated files.

Flow Concurrency and Size. Figure 5 presents the CDF of the number of flows a MLA or worker node participates in concurrently (defined as the number of flows active during a 50ms window). When all flows are considered, the median number of concurrent flows is 36, which results from the breadth of the Partition/Aggregate traffic pattern in which each server talks to 43 other servers. The 99.99th percentile is over 1,600, and there is one server with a *median* of 1,200 connections.

When only large flows ($>$ 1MB) are considered, the degree of statistical multiplexing is very low — the median number of concurrent large flows is 1, and the 75th percentile is 2. Yet, these flows are large enough that they last several RTTs, and can consume significant buffer space by causing queue buildup.

In summary, throughput-sensitive large flows, delay sensitive short flows and bursty query traffic, co-exist in a data center network. In the next section, we will see how TCP fails to satisfy the performance requirements of these flows.

2.3 Understanding Performance Impairments

We found that to explain the performance issues seen in the production cluster, we needed to study the interaction between the long and short flows in the cluster and the ways flows interact with the switches that carried the traffic.

2.3.1 Switches

Like most commodity switches, the switches in these clusters are *shared memory switches* that aim to exploit statistical multiplexing gain through use of logically common packet buffers available to all switch ports. Packets arriving on an interface are stored into a high speed multi-ported memory shared by all the interfaces. Memory from the shared pool is dynamically allocated to a packet by a MMU. The MMU attempts to give each interface as much memory as it needs while preventing unfairness [1] by dynamically adjusting the maximum amount of memory any one interface can take. If

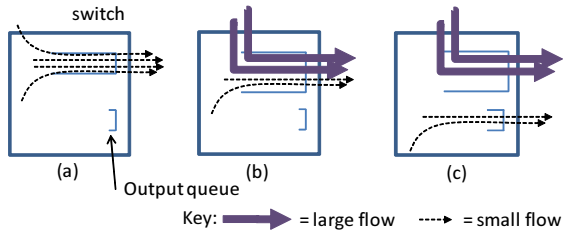


Figure 6: Three ways in which flows interact on a multi-ported switch that result in performance problems.

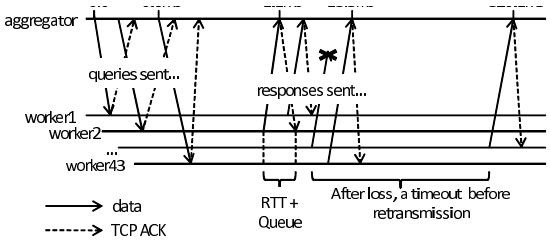


Figure 7: A real incast event measured in a production environment. Timeline shows queries forwarded over 0.8ms, with all but one response returning over 12.4ms. That response is lost, and is retransmitted after $RTO_{\min}$ (300ms). RTT+Queue estimates queue length on the port to the aggregator.

a packet must be queued for an outgoing interface, but the interface has hit its maximum memory allocation or the shared pool itself is depleted, then the packet is dropped. Building large multi-ported memories is very expensive, so most cheap switches are *shallow buffered*, with packet buffer being the scarcest resource. The shallow packet buffers cause three specific performance impairments, which we discuss next.

2.3.2 *Incast*

As illustrated in Figure 6(a), if many flows converge on the same interface of a switch over a short period of time, the packets may exhaust either the switch memory or the maximum permitted buffer for that interface, resulting in packet losses. This can occur even if the flow sizes are small. This traffic pattern arises naturally from use of the Partition/Aggregate design pattern, as the request for data synchronizes the workers' responses and creates *incast* [32] at the queue of the switch port connected to the aggregator.

The incast research published to date [32, 13] involves carefully constructed test lab scenarios. We find that incast-like problems do happen in production environments and they matter — degrading both performance and, more importantly, user experience. The problem is that a response that incurs incast will almost certainly miss the aggregator deadline and be left out of the final results.

We capture incast instances via packet-level monitoring. Figure 7 shows timeline of an observed instance. Since the size of each individual response in this application is only 2KB (2 packets) [1], loss of a packet almost invariably results in a TCP time out. In our network stack, the $RTO_{\min}$ is set to 300ms. Thus, whenever a timeout occurs, that response almost always misses the aggregator's deadline.

Developers have made two major changes to the application code to avoid timeouts on worker responses. First, they deliberately limited the size of the response to 2KB to improve the odds that all

[1]Each query goes to all machines in the rack, and each machine responds with 2KB, so the total response size is over 86KB.

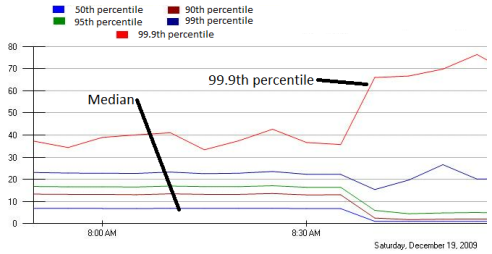


Figure 8: Response time percentiles for a production application having the incast traffic pattern. Forwarded requests were jittered (deliberately delayed) over a 10ms window until 8:30am, when jittering was switched off. The 95th and lower percentiles drop 10x, while the 99.9th percentile doubles.

the responses will fit in the memory of the switch. Second, the developers added application-level jittering [11] to desynchronize the responses by deliberating delaying them by a random amount of time (typically a mean value of 10ms). The problem with jittering is that it reduces the response time at higher percentiles (by avoiding timeouts) at the cost of increasing the median response time (due to added delay). This is vividly illustrated in Figure 8.

Proposals to decrease $RTO_{\min}$ reduce the impact of timeouts [32], but, as we show next, these proposals do not address other important sources of latency.

2.3.3 *Queue buildup*

Long-lived, greedy TCP flows will cause the length of the bottleneck queue to grow until packets are dropped, resulting in the familiar sawtooth pattern (Figure 1). When long and short flows traverse the same queue, as shown in Figure 6(b), two impairments occur. First, packet loss on the short flows can cause incast problems as described above. Second, there is a *queue buildup* impairment: even when no packets are lost, the short flows experience increased latency as they are in queue behind packets from the large flows. Since every worker in the cluster handles both query traffic and background traffic (large flows needed to update the data structures on the workers), this traffic pattern occurs very frequently.

A closer look at Figure 7 shows that arrivals of the responses are distributed over ∼12ms. Since the total size of all responses is only 43 × 2KB = 86KB — roughly 1ms of transfer time at 1Gbps — it is surprising that there would be any incast losses in such transfers. However, the key issue is the occupancy of the queue caused by other flows - the background traffic - with losses occurring when the long flows and short flows coincide.

To establish that long flows impact the latency of query responses, we measured the RTT between the worker and the aggregator: this is the time between the worker sending its response and receiving a TCP ACK from the aggregator labeled as "RTT+Queue" in Figure 7. We measured the intra-rack RTT to approximately 100μs in absence of queuing, while inter-rack RTTs are under 250μs. This means "RTT+queue" is a good measure of the the length of the packet queue headed to the aggregator during the times at which the aggregator is collecting responses. The CDF in Figure 9 is the distribution of queue length for 19K measurements. It shows that 90% of the time a response packet sees $<$ 1ms of queueing, and 10% of the time it sees between 1 and 14ms of queuing (14ms is the maximum amount of dynamic buffer). This indicates that query flows are indeed experiencing queuing delays. Further, note that answering a request can require multiple iterations, which magnifies the impact of this delay.

Note that this delay is unrelated to incast. No packets are being lost, so reducing RTO_{min} will not help. Further, there need not

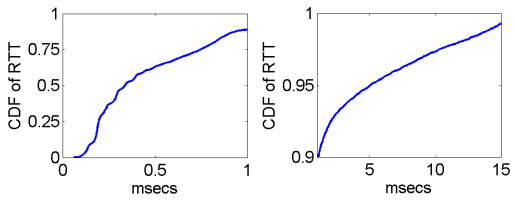


Figure 9: CDF of RTT to the aggregator. 10% of responses see an unacceptable queuing delay of 1 to 14ms caused by long flows sharing the queue.

even be many synchronized short flows. *Since the latency is caused by queueing, the only solution is to reduce the size of the queues.*

2.3.4 *Buffer pressure*

Given the mix of long and short flows in our data center, it is very common for short flows on one port to be impacted by activity on any of the many other ports, as depicted in Figure 6(c). Indeed, the loss rate of short flows in this traffic pattern depends on the number of long flows *traversing other ports*. The explanation is that activity on the different ports is coupled by the shared memory pool.

The long, greedy TCP flows build up queues on their interfaces. Since buffer space is a shared resource, the queue build up reduces the amount of buffer space available to absorb bursts of traffic from Partition/Aggregate traffic. We term this impairment *buffer pressure*. The result is packet loss and timeouts, as in incast, but without requiring synchronized flows.

3. THE DCTCP ALGORITHM

The design of DCTCP is motivated by the performance impairments described in § 2.3. The goal of DCTCP is to achieve high burst tolerance, low latency, and high throughput, with commodity shallow buffered switches. To this end, DCTCP is designed to operate with small queue occupancies, without loss of throughput.

DCTCP achieves these goals primarily by reacting to congestion in proportion to the extent of congestion. DCTCP uses a simple marking scheme at switches that sets the Congestion Experienced (CE) codepoint of packets as soon as the buffer occupancy exceeds a fixed small threshold. The DCTCP source reacts by reducing the window by a factor that depends on the *fraction* of marked packets: the larger the fraction, the bigger the decrease factor.

It is important to note that the key contribution here is not the control law itself. It is the act of *deriving multi-bit feedback from the information present in the single-bit sequence of marks*. Other control laws that act upon this information can be derived as well. Since DCTCP requires the network to provide only single-bit feedback, we are able to re-use much of the ECN machinery that is already available in modern TCP stacks and switches.

The idea of reacting in proportion to the extent of congestion is also used by delay-based congestion control algorithms [5, 31]. Indeed, one can view path delay information as implicit multi-bit feedback. However, at very high data rates and with low-latency network fabrics, sensing the queue buildup in shallow-buffered switches can be extremely noisy. For example, a 10 packet backlog constitutes 120μs of queuing delay at 1Gbps, and only 12μs at 10Gbps. The accurate measurement of such small increases in queueing delay is a daunting task for today's servers.

The need for reacting in proportion to the extent of congestion is especially acute in the absence of large-scale statistical multiplexing. Standard TCP cuts its window size by a factor of 2 when it receives ECN notification. In effect, TCP reacts to presence of

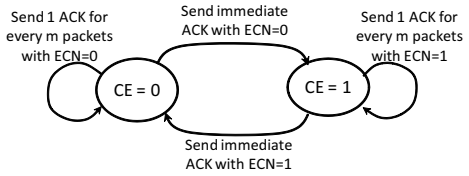


Figure 10: Two state ACK generation state machine.

congestion, not to its *extent* [2]. Dropping the window in half causes a large mismatch between the input rate to the link and the available capacity. In the high speed data center environment where only a small number of flows share the buffer (§ 2.2), this leads to buffer underflows and loss of throughput.

3.1 Algorithm

The DCTCP algorithm has three main components:

(1) Simple Marking at the Switch: DCTCP employs a very simple active queue management scheme. There is only a single parameter, the marking threshold, K. An arriving packet is marked with the CE codepoint if the queue occupancy is greater than K upon it's arrival. Otherwise, it is not marked. This scheme ensures that sources are quickly notified of the queue overshoot. The RED marking scheme implemented by most modern switches can be re-purposed for DCTCP. We simply need to set both the low and high thresholds to K, and mark based on instantaneous, instead of average queue length.

(2) ECN-Echo at the Receiver: The only difference between a DCTCP receiver and a TCP receiver is the way information in the CE codepoints is conveyed back to the sender. RFC 3168 states that a receiver sets the ECN-Echo flag in a series of ACK packets until it receives confirmation from the sender (through the CWR flag) that the congestion notification has been received. A DCTCP receiver, however, tries to accurately convey the exact sequence of marked packets back to the sender. The simplest way to do this is to ACK every packet, setting the ECN-Echo flag if and only if the packet has a marked CE codepoint.

However, using Delayed ACKs is important for a variety of reasons, including reducing the load on the data sender. To use delayed ACKs (one cumulative ACK for every m consecutively received packets [3]), the DCTCP receiver uses the trivial two state state-machine shown in Figure 10 to determine whether to set ECN-Echo bit. The states correspond to whether the last received packet was marked with the CE codepoint or not. Since the sender knows how many packets each ACK covers, it can exactly reconstruct the runs of marks seen by the receiver.

(3) Controller at the Sender: The sender maintains an estimate of the fraction of packets that are marked, called α, which is updated once for every window of data (roughly one RTT) as follows:

$$\alpha \leftarrow (1-g) \times \alpha + g \times F, \quad (1)$$

where F is the *fraction* of packets that were marked in the last window of data, and $0 < g < 1$ is the weight given to new samples against the past in the estimation of α. Given that the sender receives marks for *every packet* when the queue length is higher than K and does not receive *any marks* when the queue length is below K, Equation (1) implies that α estimates the probability that the queue size is greater than K. Essentially, α close to 0 indicates low, and α close to 1 indicates high levels of congestion.

[2]Other variants which use a variety of fixed factors and/or other fixed reactions have the same issue.

[3]Typically, one ACK every 2 packets.

Prior work [26, 20] on congestion control in the small buffer regime has observed that at high line rates, queue size fluctuations become so fast that you cannot control the queue size, only its *distribution*. The physical significance of α is aligned with this observation: it represents a single point of the queue size distribution at the bottleneck link.

The only difference between a DCTCP sender and a TCP sender is in how each reacts to receiving an ACK with the ECN-Echo flag set. Other features of TCP such as slow start, additive increase in congestion avoidance, or recovery from packet lost are left unchanged. While TCP always cuts it's window size by a factor of 2 in response[4] to a marked ACK, DCTCP uses α:

$$cwnd \leftarrow cwnd \times (1 - \alpha/2). \quad (2)$$

Thus, when α is near 0 (low congestion), the window is only slightly reduced. In other words, DCTCP senders start gently reducing their window as soon as the queue exceeds K. This is how DCTCP maintains low queue length, while still ensuring high throughput. When congestion is high ($\alpha = 1$), DCTCP cuts window in half, just like TCP.

3.2 Benefits

DCTCP alleviates the three impairments discussed in § 2.3 (shown in Figure 6) as follows.

Queue buildup: DCTCP senders start reacting as soon as queue length on an interface exceeds K. This reduces queueing delays on congested switch ports, which minimizes the impact of long flows on the completion time of small flows. Also, more buffer space is available as *headroom* to absorb transient micro-bursts, greatly mitigating costly packet losses that can lead to timeouts.

Buffer pressure: DCTCP also solves the buffer pressure problem because a congested port's queue length does not grow exceedingly large. Therefore, in shared memory switches, a few congested ports will not exhaust the buffer resources harming flows passing through other ports.

Incast: The incast scenario, where a large number of synchronized small flows hit the same queue, is the most difficult to handle. If the number of small flows is so high that even 1 packet from each flow is sufficient to overwhelm the buffer on a synchronized burst, then there isn't much DCTCP—or any congestion control scheme that does not attempt to schedule traffic—can do to avoid packet drops.

However, in practice, each flow has several packets to transmit, and their windows build up over multiple RTTs. It is often bursts in subsequent RTTs that lead to drops. Because DCTCP starts marking early (and aggressively – based on instantaneous queue length), DCTCP sources receive enough marks during the first one or two RTTs to tame the size of follow up bursts. This prevents buffer overflows and resulting timeouts.

3.3 Analysis

We now analyze the steady state behavior of the DCTCP control loop in a simplified setting. We consider N infinitely long-lived flows with identical round-trip times RTT, sharing a single bottleneck link of capacity C. We further assume that the N flows are synchronized; i.e., their "sawtooth" window dynamics 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 (§ 2.2).

Because the N window sizes are synchronized, they follow identical sawtooths, and the queue size at time t is given by

$$Q(t) = NW(t) - C \times RTT, \quad (3)$$

[4]Both TCP and DCTCP cut their window size at most once per window of data [27].

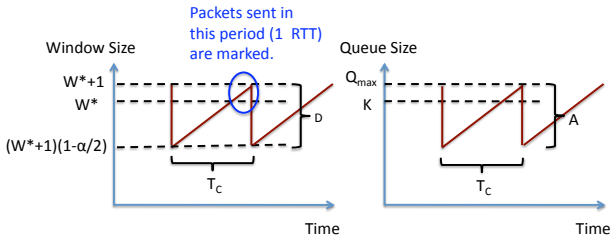


Figure 11: Window size of a single DCTCP sender, and the queue size process.

where $W(t)$ is the window size of a single source [4]. Therefore the queue size process is also a sawtooth. We are interested in computing the following quantities which completely specify the sawtooth (see Figure 11): the maximum queue size (Q_{max}), the amplitude of queue oscillations (A), and the period of oscillations (T_C). The most important of these is the amplitude of oscillations, which quantifies how well DCTCP is able to maintain steady queues, due to its gentle proportionate reaction to congestion indications.

We proceed to computing these quantities. The key observation is that with synchronized senders, the queue size 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 can compute the fraction of marked packets, α, by simply dividing the number of packets sent during the last RTT of the period by the total number of packets sent during a full period of the sawtooth, T_C.

Let's consider one of the senders. Let $S(W_1, W_2)$ denote the number of packets sent by the sender, while its window size increases from W_1 to $W_2 > W_1$. Since this takes $W_2 - W_1$ round-trip times, during which the average window size is $(W_1 + W_2)/2$,

$$S(W_1, W_2) = (W_2^2 - W_1^2)/2. \quad (4)$$

Let $W^* = (C \times RTT + K)/N$. This is the critical window size at which the queue size reaches K, and the switch starts marking packets with the CE codepoint. During the RTT it takes for the sender to react to these marks, its window size increases by one more packet, reaching $W^* + 1$. Hence,

$$\alpha = S(W^*, W^* + 1)/S((W^* + 1)(1 - \alpha/2), W^* + 1). \quad (5)$$

Plugging (4) into (5) and rearranging, we get:

$$\alpha^2(1 - \alpha/4) = (2W^* + 1)/(W^* + 1)^2 \approx 2/W^*, \quad (6)$$

where the approximation in (6) is valid when $W^* >> 1$. Equation (6) can be used to compute α as a function of the network parameters C, RTT, N and K. Assuming α is small, this can be simplified as $\alpha \approx \sqrt{2/W^*}$. We can now compute A and T_C in Figure 11 as follows. Note that the amplitude of oscillation in window size of a single flow, D, (see Figure 11) is given by:

$$D = (W^* + 1) - (W^* + 1)(1 - \alpha/2). \quad (7)$$

Since there are N flows in total,

$$A = ND = N(W^* + 1)\alpha/2 \approx \frac{N}{2}\sqrt{2W^*}$$
$$= \frac{1}{2}\sqrt{2N(C \times RTT + K)}, \quad (8)$$

$$T_C = D = \frac{1}{2}\sqrt{2(C \times RTT + K)/N} \text{ (in RTTs)}. \quad (9)$$

Finally, using (3), we have:

$$Q_{max} = N(W^* + 1) - C \times RTT = K + N. \quad (10)$$

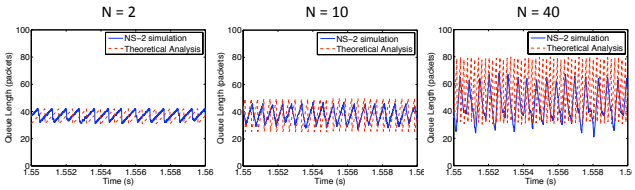


Figure 12: Comparison between the queue size process predicted by the analysis with NS-2 simulations. The DCTCP parameters are set to $K = 40$ packets, and $g = 1/16$.

We have evaluated the accuracy of the above results using NS-2 simulations in a variety of scenarios. Figure 12 shows the results for $N = 2, 10$, and 40 long-lived DCTCP flows sharing a 10Gbps bottleneck, with a $100\mu s$ round-trip time. As seen, the analysis is indeed a fairly accurate prediction of the actual dynamics, especially when N is small (less than 10). For large N, as in the $N = 40$ case, de-synchronization of the flows leads to smaller queue variations than predicted by the analysis.

Equation (8) reveals an important property of DCTCP; when N is small, the amplitude of queue size oscillations with DCTCP is $O(\sqrt{C \times RTT})$, and is therefore much smaller than the $O(C \times RTT)$ oscillations of TCP. This allows for a very small marking threshold K, without loss of throughput in the low statistical multiplexing regime seen in data centers. In fact, as we verify in the next section, even with the *worst* case assumption of synchronized flows used in this analysis, DCTCP can begin marking packets at $(1/7)^{th}$ of the bandwidth-delay product without losing throughput.

3.4 Guidelines for choosing parameters

In this section, C is in packets/second, RTT is in seconds, and K is in packets.

Marking Threshold. The minimum value of the queue occupancy sawtooth is given by:

$$Q_{min} = Q_{max} - A \tag{11}$$

$$= K + N - \frac{1}{2}\sqrt{2N(C \times RTT + K)}. \tag{12}$$

To find a lower bound on K, we minimize (12) over N, and choose K so that this minimum is larger than zero, i.e. the queue does not underflow. This results in:

$$K > (C \times RTT)/7. \tag{13}$$

Estimation Gain. The estimation gain g must be chosen small enough to ensure the exponential moving average (1) "spans" at least one congestion event. Since a congestion event occurs every T_C round-trip times, we choose g such that:

$$(1 - g)^{T_C} > 1/2. \tag{14}$$

Plugging in (9) with the worst case value $N = 1$, results in the following criterion:

$$g < 1.386/\sqrt{2(C \times RTT + K)}. \tag{15}$$

3.5 Discussion

AQM is not enough: Before designing DCTCP, we evaluated Active Queue Management (AQM) schemes like RED [5] and PI [17] that do not modify TCP's congestion control mechanism. We found they do not work well when there is low statistical multiplexing and traffic is bursty—both of which are true in the data center. Essentially, because of TCP's conservative reaction to congestion indications, any AQM scheme operating with TCP in a data-center-like environment requires making a tradeoff between throughput and delay [9]: either accept large queue occupancies (and hence delay), or accept loss of throughput.

We will examine performance of RED (with ECN) in some detail in § 4, since our testbed switches are RED/ECN-capable. We have evaluated other AQM schemes such as PI extensively using NS-2. See [3] for detailed results. Our simulation results show that with few flows (< 5), PI suffers from queue underflows and a loss of utilization, while with many flows (20), queue oscillations get worse, which can hurt the latency of time-critical flows.

Convergence and Synchronization: In both analysis and experimentation, we have found that DCTCP achieves both high throughput and low delay, all in an environment with low statistical multiplexing. In achieving this, DCTCP trades off convergence time; the time required for a new flow to grab its share of the bandwidth from an existing flow with a large window size. This is expected since a DCTCP source must make incremental adjustments to its window size based on the accumulated multi-bit feedback in α. The same tradeoff is also made by a number of TCP variants [22, 23].

We posit that this is not a major concern in data centers. First, data center round-trip times are only a few 100μsec, 2 orders of magnitudes less than RTTs in the Internet. Since convergence time for a window based protocol like DCTCP is proportional to the RTT, the actual differences in time cause by DCTCP's slower convergence compared to TCP are not substantial. Simulations show that the convergence times for DCTCP is on the order of 20-30ms at 1Gbps, and 80-150ms at 10Gbps, a factor of 2-3 more than TCP [6]. Second, in a data center dominated by microbursts, which by definition are too small to converge, and big flows, which can tolerate a small convergence delay over their long lifetimes, convergence time is the right metric to yield.

Another concern with DCTCP is that the "on-off" style marking can cause synchronization between flows. However, DCTCP's reaction to congestion is not severe, so it is less critical to avoid synchronization [10].

Practical considerations: While the recommendations of § 3.4 work well in simulations, some care is required before applying these recommendations in real networks. The analysis of the previous section is for idealized DCTCP sources, and does not capture any of the burstiness inherent to actual implementations of window-based congestion control protocols in the network stack. For example, we found that at 10G line rates, hosts tend to send bursts of as many as 30-40 packets, whenever the window permitted them to do so. While a myriad of system details (quirks in TCP stack implementations, MTU settings, and network adapter configurations) can cause burstiness, optimizations such as *Large Send Offload* (LSO), and interrupt moderation increase burstiness noticeably [7]. One must make allowances for such bursts when selecting the value of K. For example, while based on (13), a marking threshold as low as 20 packets can be used for 10Gbps, we found that a more conservative marking threshold larger than 60 packets is required to avoid loss of throughput. This excess is in line with the burst sizes of 30-40 packets observed at 10Gbps.

Based on our experience with the intrinsic burstiness seen at 1Gbps and 10Gbps, and the total amount of available buffering in our switches, we use the marking thresholds of $K = 20$ packets for

[5] We *always* use RED with ECN: i.e. random early *marking*, not random early drop. We call it RED simply to follow convention.

[6] For RTTs ranging from 100μs to 300μs.

[7] Of course, such implementation issues are not specific to DCTCP and affect any protocol implemented in the stack.

	Ports	Buffer	ECN
Triumph	48 1Gbps, 4 10Gbps	4MB	Y
Scorpion	24 10Gbps	4MB	Y
CAT4948	48 1Gbps, 2 10Gbps	16MB	N

Table 1: Switches in our testbed

1Gbps and $K = 65$ packets for 10Gbps ports in our experiments, unless otherwise noted. g is set to 1/16 in all experiments.

4. RESULTS

This section is divided in three parts. First, using carefully designed traffic patterns, we examine the basic properties of the DCTCP algorithm, such as convergence, fairness, and behavior in a multi-hop environment. Second, we show a series of microbenchmarks that explain how DCTCP ameliorates the specific performance impairments described in §2.3. Finally, we evaluate DCTCP using a benchmark generated from our traffic measurements. *No simulations are used in this section.* All comparisons are between a full implementation of DCTCP and a state-of-the-art TCP New Reno (w/ SACK) implementation. Unless otherwise noted, we use the parameter settings discussed at the end of § 3.

Our testbed consists of 94 machines in three racks. 80 of these machines have 1Gbps NICs, and the remaining 14 have 10Gbps NICs. The CPU and memory of these machines were never a bottleneck in any of our experiments. We connect these machines together in a variety of configurations using the set of switches shown in Table 1. CAT4948 is a Cisco product, the rest are from Broadcom. The Triumph and Scorpion are "shallow buffered," while the CAT4948 is a deep-buffered switch. Except where noted, the switches used their default dynamic buffer allocation policy.

4.1 DCTCP Performance

The experiments in this subsection are microbenchmarks, with traffic patterns specifically designed to evaluate particular aspects of DCTCP's performance.

Throughput and queue length: We begin by evaluating whether DCTCP achieves the same throughput as TCP on long-lived flows when recommended values of K are used. To determine this, we use machines connected to the Triumph switch with 1Gbps links. One host is a receiver; the others are senders. The senders establish long-lived connections to the receiver and send data as fast as they can. During the transfer, we sample the instantaneous queue length at the receiver's switch port every 125ms. We repeat the experiment for both DCTCP and TCP. For DCTCP, we set $K = 20$, as recommended before. For TCP, the switch operates in standard, drop-tail mode.

We find that both TCP and DCTCP achieve the maximum throughput of 0.95Gbps, and link utilization is nearly 100%. The key difference is queue length at the receiver interface. The CDF in Figure 13 shows that DCTCP queue length is stable around 20 packets (i.e., equal to $K + n$, as predicted in § 3), while the TCP queue length is 10X larger and varies widely. In fact, Figure 1 is based on the data from this experiment. It shows the time series of queue length (with 2 flows) for a representative period. The sawtooth behavior of TCP is clearly visible. The upper limit for TCP's queue length is dictated by the switch's dynamic buffer allocation policy, which will allocate up to ∼700KB of buffer to a single busy interface if no other port is receiving traffic. In contrast, DCTCP maintains a stable, low queue length.

We also tried various other values of K, and found that the performance is insensitive to value of K at 1Gbps, even for values as low as $K = 5$. We then repeated the experiment with 10Gbps link. Figure 14 shows the throughput results. Once the value of

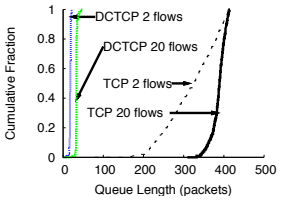


Figure 13: Queue length CDF (1Gbps)

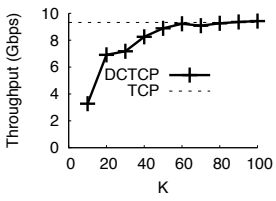
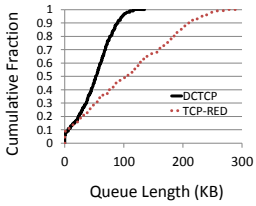
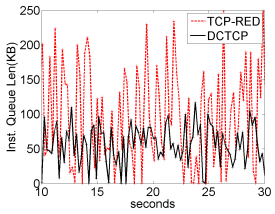


Figure 14: Throughput (10Gbps)



(a) CDF of queue length



(b) Time series of queue length

Figure 15: DCTCP versus RED at 10Gbps

K exceeds the recommended value of 65, DCTCP gets the same throughput as TCP, and is no longer sensitive to the value of K.

RED: In §3, we argued that even AQM schemes like RED would show queue length oscillations, as these oscillations stem from the way TCP adjusts its window in response to congestion indications. We also argued that DCTCP does not suffer from this problem.

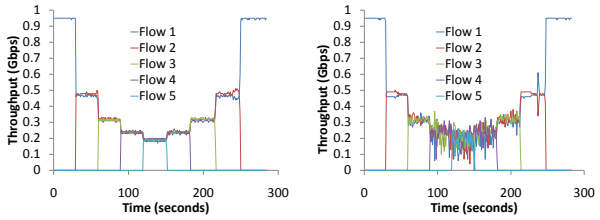
To verify this, we repeated the 10Gbps experiment. We again ran DCTCP with $K = 65$. For TCP, the switch marked packets using RED.[8] It was difficult to set RED parameters correctly: following the guidelines in [7] (max_p=0.1, weight=9, min_th=50, max_th=150) caused TCP throughput to drop to 7.8Gbps. To get a fair comparison with DCTCP, we increased the value of the min_th and max_th RED parameters until TCP achieved 9.2Gbps at min_th = 150. Figure 15 shows the distribution of queue length observed at the receiver's switch port.

We see that RED causes wide oscillations in queue length, often requiring twice as much buffer to achieve the same throughput as DCTCP. This transient queue buildup means there is less room available to absorb microbursts, and we will see the impact of this on our real benchmark in §4.3. However, given the difficulty of setting RED parameters, we use TCP with drop tail as the baseline for all other experiments. The baseline also reflects what our production clusters (§ 2) actually implement.

The key takeaway from these experiments is that *with K set according to the guidelines in §3.4, DCTCP achieves full throughput, even at very small queue length. TCP with drop tail or RED causes queue lengths to oscillate widely.*

Fairness and convergence: To show that DCTCP flows quickly converge to their fair share, we set up 6 hosts connected via 1Gbps links to the Triumph switch. K was set to 20. One of the hosts acts as a receiver, while the others act as senders. We start a single long-lived flow, and then we sequentially start and then stop the other senders, spaced by 30 seconds. The timeseries depicting the overall flow throughput is shown in Figure 16(a). As DCTCP flows come and go, they quickly converge to their fair share. For comparison, the corresponding timeseries for TCP is shown in Figure 16(b). TCP throughput is fair on average, but has much higher variation. We have repeated this test with up to 90 flows, and we find that DCTCP converges quickly, and all flows achieve their

[8]Our switches do not support any other AQM scheme. RED is implemented by setting the ECN bit, not dropping.



(a) DCTCP (b) TCP

Figure 16: Convergence test

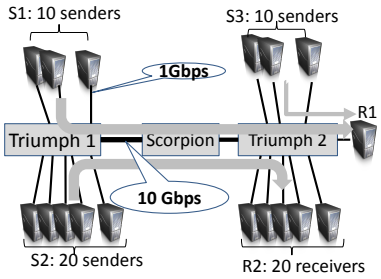


Figure 17: Multi-hop topology

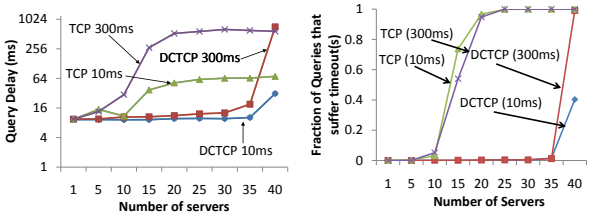


Figure 18: DCTCP performs better than TCP and then converges at 35 senders (log scale on Y axis; 90% confidence intervals for the means are too small to be visible).

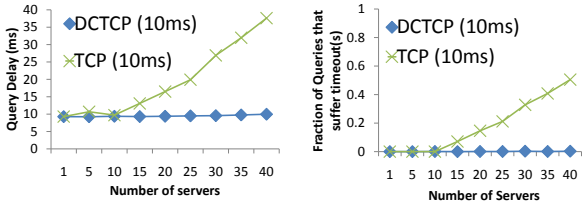


Figure 19: Many-to-one: With dynamic buffering, DCTCP does not suffer problems at even high load.

fair share (Jain's fairness index is 0.99). We make no claims that DCTCP is fair to TCP. If need be, DCTCP flows can be isolated from TCP flows, which can be easily accomplished in most data centers, as discussed in (§ 1).

Multi-hop networks: To evaluate DCTCP performance in a multi-hop, multi-bottleneck environment, we created the topology in Figure 17. The senders in the S1 and S3 groups, totaling 20 machines, all send to receiver R1. The 20 senders in S2 each send to an assigned receiver in group R2. There are two bottlenecks in this topology: both the 10Gbps link between Triumph 1 and the Scorpion and the 1Gbps link connecting Triumph 2 to R1 are oversubscribed. The flows from the senders in group S1 encounter both these bottlenecks. We find that with DCTCP, each sender in S1 gets 46Mbps and S3 gets 54Mbps throughput, while each S2 sender gets approximately 475Mbps — these are within 10% of their fair-share throughputs. TCP does slightly worse: the queue length fluctuations at the two bottlenecks cause timeouts for some of the TCP connections. This experiment shows that DCTCP can cope with the multiple bottlenecks and differing RTTs that will be found in data centers.

4.2 Impairment microbenchmarks

We now show a series of microbenchmarks that show how DCTCP addresses the impairments described in § 2.3.

4.2.1 Incast

In this section, we examine the *incast* impairment (§ 2.3). In [32] value of RTO_{min} was reduced to address the incast problem. We compare this approach with DCTCP's approach of *avoiding* timeouts, as explained in 3.2.

Basic incast: We start with an experiment that repeats the conditions in [32]. Forty-one machines are connected to the Triumph switch with 1Gbps links, and, to duplicate the prior work, for this one experiment the dynamic memory allocation policy in the switch was replaced with a static allocation of 100 packets to each port.

One machine acts as a client, others act as servers. The client requests ("queries") 1MB/n bytes from n different servers, and each server responds immediately with the requested amount of data. The client waits until it receives all the responses, and then issues another, similar query. This pattern is repeated 1000 times. The metric of interest is the completion time of the queries. The minimum query completion time is around 8ms: the incoming link at the receiver is the bottleneck, and it takes 8ms to deliver 1MB of data over a 1Gbps link. We carry out the experiment for both TCP and DCTCP, and with two timeout values: the default 300ms, and 10ms. The latter value is the tick granularity of our system, so it is the smallest timeout value we can use without major changes to timer handling.

Figure 18(a) shows the mean query completion time. DCTCP performs much better than both TCP variants — eventually converging to equivalent performance as incast degree increases. Figure 18(b) makes the reason evident by showing the fraction of queries that suffered at least one timeout.[9] TCP begins to suffer timeouts when the number of servers exceeds 10, as no sender receives a signal to lower its sending rate until enough packets are in flight to cause a full window loss. DCTCP senders receive ECN marks, slow their rate, and only suffer timeouts once the number of senders is large enough so that each sending (around) 2 packets exceeds the static buffer size ($35 \times 2 \times$ 1.5KB > 100KB).

Importance of dynamic buffering: Recall that the above experiment used a static buffer of 100 packets at each port. Would using the switch's default dynamic buffer allocation algorithm (§ 2.3.1) solve the incast problem? To answer this question, we repeated the above experiment with dynamic buffering enabled. Given the poor performance of TCP with 300ms timeout (top curve in Figure 18(a)), we use 10ms timeout here and for the rest of the paper.

Figure 19 shows that DCTCP no longer suffers incast timeouts even when the number of servers grows to 40. On the other hand, TCP continues to suffer from incast timeouts, although the dynamic buffering algorithm mitigates the impact by allocating as much as 700KB of buffer to the receiver's port (it does not allocate all 4MB for fairness). The allocated buffer is sufficient for DCTCP to avoid timeouts even with a large number of servers.

All-to-all incast: In the previous experiments, there was only a single receiver. This is an easy case for the dynamic buffer management to handle, as there is only one receiver, so only one port

[9]Each query elicits a response from several servers, any of which can suffer a timeout. However, if multiple responses suffer timeouts, the delay does not increase proportionately, since the responses are delivered in parallel.

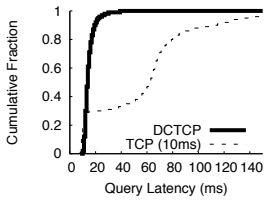


Figure 20: All-to-all: Dynamic buffering works well for DCTCP.

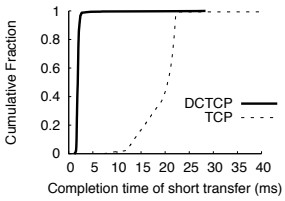


Figure 21: Short transfers see low delay with DCTCP.

gets congested. But what happens when there are multiple simultaneous incasts going on, as occurs in the production clusters? To investigate, we use the same setup as before, except all 41 machines participate. Each machine requests 25KB of data from the remaining 40 machines (total 1MB). In other words, 40 incast experiments happen all at once. The CDF of the response time is shown in Figure 20. We see that DCTCP keeps the demand on buffer space low enough that the dynamic buffering is able to cover all requests for memory and DCTCP suffers no timeouts at all. TCP, on the other hand, performs poorly, because over 55% of the queries suffer from at least one timeout.

Other settings: We also investigated TCP and DCTCP's performance with incast traffic patterns in scenarios including 10Gbps links and larger (10MB) and smaller response sizes (100KB). The results are qualitatively similar to those presented here. Repeating the experiments with our CAT4948 deep-buffered switch, we found a reduction in TCP's incast problem for small response sizes (< 1MB), but the problem resurfaces for larger response sizes (10MB). DCTCP performs well at all response sizes. Besides, the deep buffers allow for larger queue buildups, which hurts performance of other traffic — we examine this in detail in §4.3.

In summary, *DCTCP handles incast without timeouts until there are so many senders that the traffic sent in the first RTT overflows the buffers.* Its performance is further enhanced by the dynamic buffering offered by modern switches.

4.2.2 Queue buildup

The second impairment scenario in § 2.3 involves big and small flows mixing in the same queue, with the queue build-up caused by the big flow increasing the latency of the small flows. To measure DCTCP's impact on this impairment, we connect 4 machines to the Triumph switch with 1 Gbps links. One machine is a receiver and the other three are senders. We start one big TCP flow each from two senders to the receiver — the 75th percentile of concurrent connections measured in our data center (Figure 5). The receiver then requests 20KB chunks of data from the third sender. The sender responds immediately, and the receiver sends the next request as soon as it finishes receiving data. All communication is over long-lived connections, so there is no three-way handshake for each request.

Figure 21 shows the CDF of request completion times for 1000 20KB transfers. DCTCP's completion time (median delay < 1 ms) is much lower than TCP (median delay 19ms). No flows suffered timeouts in this scenario, so reducing RTO_{min} would not reduce the delay. Since the amount of data transferred is small, the completion time is dominated by the round trip time, which is dominated by the queue length at the switch.

Thus, *DCTCP improves latency for small flows by reducing queue lengths, something reducing RTO_{min} does not affect.*

4.2.3 Buffer pressure

The third impairment scenario in § 2.3 involves flows on one set of interfaces increasing the rate of packet loss on other interfaces. Recall that these are shared-memory switches. With TCP, long flows use up the shared buffer space, which leaves less headroom to absorb incast bursts. DCTCP should alleviate this problem by limiting the queue build up on the interfaces used by long flows. To evaluate this, we connected 44 hosts to a Triumph switch with 1Gbps links. 11 of these hosts participate in a 10-1 incast pattern, with 1 host acting as a client, and 10 hosts acting as servers. The client requested a total of 1MB data from the servers (100KB from each), repeating the request 10,000 times. We have seen (Figure 19), that the switch can easily handle 10:1 incast with both TCP and DCTCP, without inducing any timeouts.

	Without background traffic	With background traffic
TCP	9.87ms	46.94ms
DCTCP	9.17ms	9.09ms

Table 2: 95^{th} percentile of query completion time. DCTCP prevents background traffic from affecting performance of query traffic. $RTO_{min} = 10ms$, $K = 20$.

Next, we use the remaining 33 hosts to start "background" traffic of long-lived flows to consume the shared buffer. We start a total of 66 big flows between the 33 hosts, with each host sending data to two other hosts.[10] Table 2 shows the 95th percentile of query completion times.

We see that with TCP, query completion time is substantially worse in the presence of long-lived flows, while DCTCP's performance is unchanged. This performance difference is due to timeouts: about 7% of queries suffer from timeouts with TCP, while only 0.08% do so under DCTCP. Long-lived flows get the same throughput under both TCP and DCTCP.

This experiment shows *DCTCP improves the performance isolation between flows by reducing the buffer pressure that would otherwise couple them.*

In summary, these microbenchmarks allow us to individually test DCTCP for fairness, high throughput, high burst tolerance, low latency and high performance isolation while running at 1 and 10Gbps across shared-buffer switches. On all these metrics, DCTCP significantly outperforms TCP.

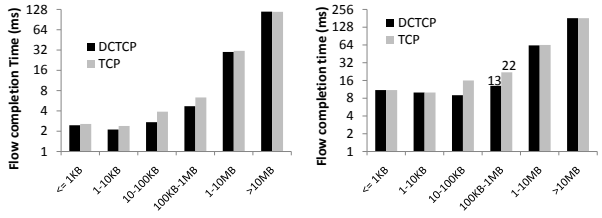
4.3 Benchmark Traffic

We now evaluate how DCTCP would perform under the traffic patterns found in production clusters (§2.2). For this test, we use 45 servers connected to a Triumph top of rack switch by 1Gbps links. An additional server is connected to a 10Gbps port of the Triumph to act as a stand-in for the rest of the data center, and all inter-rack traffic is directed to/from this machine. This aligns with the actual data center, where each rack connects to the aggregation switch with a 10Gbps link.

We generate all three types of traffic found in the cluster: query, short-message, and background. Query traffic is created following the Partition/Aggregate structure of the real application by having each server draw from the interarrival time distribution and send a query to *all* other servers in the rack, each of which then send back a 2KB response ($45 \times 2\text{KB} \approx 100\text{KB}$ total response size). For the short-message and background traffic, each server draws independently from the interarrival time and the flow size distributions, choosing an endpoint so the ratio of inter-rack to intra-rack flows is the same as measured in the cluster.[11] We carry out these experiments using TCP and DCTCP, with RTO_{min} set to 10ms in both.

[10] Queue buildup only occurs when there is more than 1 flow, which measurements of our cluster show happens 25% of the time (§ 2).

[11] Background flows have some structure (e.g., pattern of polling other workers for updates), so using two independent distributions instead of a joint distribution is an approximation.



(a) Mean (b) 95th Percentile

Figure 22: Completion time of background traffic. Note the log scale on the Y axis.

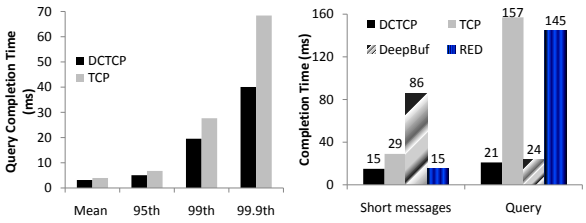


Figure 23: Completion time: query traffic

Figure 24: 95th percentile completion time: 10x background and 10x query

For DCTCP experiments, K was set to 20 on 1Gbps links and to 65 on the 10Gbps link. Dynamic buffer allocation was used in all cases. We generate traffic for 10 minutes, comprising over 200,000 background flows and over 188,000 queries.

Both query and short-message flows are time critical, their metric of interest is completion time. The RTT (i.e. queue length) and timeouts affect this delay the most. For large flows in the background traffic (e.g., updates), the throughput of the network is the main consideration.

Figure 22 shows the mean and 95th percentile of completion delay for background traffic, classified by flow sizes. The 90% confidence intervals of the mean are too tight to be shown. Short-messages benefit significantly from DCTCP, as flows from 100KB-1MB see a 3ms/message benefit at the mean and a 9ms benefit at 95th percentile. The background traffic did not suffer any timeouts with either protocol in this experiment. Thus, the *lower latency for short-messages is due to DCTCP's amelioration of queue buildup.*

Figure 23 shows query completion time statistics. DCTCP performs better than TCP, especially at the tail of the distribution. The reason is a combination of timeouts and high queueing delay. With TCP, 1.15% of the queries suffer from timeout(s). No queries suffer from timeouts with DCTCP.

Scaled traffic: The previous benchmark shows how DCTCP performs on today's workloads. However, as explained in §2.3, the traffic parameters we measured reflect extensive optimization conducted by the developers to get the existing system into the tight SLA bounds on response time. For example, they restrict the size of query responses and update frequency, thereby trading off response quality for response latency. This naturally leads to a series of "what if" questions: how would DCTCP perform if query response sizes were larger? Or how would DCTCP perform if background traffic characteristics were different? We explore these questions by scaling the traffic in the benchmark.

We begin by asking if using DCTCP instead of TCP would allow a 10X increase in both query response size and background flow size without sacrificing performance. We use the same testbed as before. We generate traffic using the benchmark, except we increase the size of update flows larger than 1MB by a factor of 10 (most bytes are in these flows, so this effectively increases the volume of background traffic by a factor of 10). Similarly, we generate queries as before, except that the total size of each response is 1MB (with 44 servers, each individual response is just under 25KB). We conduct the experiment for both TCP and DCTCP.

To see whether other solutions, besides DCTCP, may fix TCP's performance problems, we tried two variations. First, we replaced the shallow-buffered Triumph switch with the deep-buffered CAT4948 switch.[12] Second, instead of drop tail queues, we used RED with ECN. It was as difficult to tune RED parameters at 1Gbps as it was previously at 10Gbps: after experimentation, we found that setting $min\_th = 20$, $max\_th = 60$ and using [7] for the remaining parameters gave the best performance.

Figure 24 shows the 95th percentile of response times for the short messages (100KB-1MB) and the query traffic (mean and other percentiles are qualitatively similar). The results show DCTCP performs significantly better than TCP for both update and query traffic. The 95th percentile of completion time for short-message traffic improves by 14ms, while query traffic improves by 136ms. With TCP, over 92% of the queries suffer from timeouts, while only 0.3% suffer from timeouts with DCTCP.

In fact, short message completion time for DCTCP is essentially unchanged from baseline (Figure 22(b)) and, even at 10X larger size, only 0.3% of queries experience timeouts under DCTCP: in contrast TCP suffered 1.15% timeouts for the baseline.[13] Thus, DCTCP can handle substantially more traffic without any adverse impact on performance.

Deep buffered switches have been suggested as a fix for TCP's incast problem, and we indeed see that with CAT4948 TCP's query completion time is comparable to DCTCP, since less than 1% of queries suffer from timeout. However, if deep buffers are used, the short-message traffic is penalized: their completion times are over 80ms, which is substantially higher than TCP without deep buffers (DCTCP is even better). The reason is that deep buffers cause queue buildup.

We see that RED is not a solution to TCP's problems either: while RED improves performance of short transfers by keeping average queue length low, the high variability (see Figure 15) of the queue length leads to poor performance for the query traffic (95% of queries experience a timeout). Another possible factor is that RED marks packets based on average queue length, and is thus slower to react to bursts of traffic caused by query incast.

These results make three key points: *First, if our data center used DCTCP it could handle 10X larger query responses and 10X larger background flows while performing better than it does with TCP today. Second, while using deep buffered switches (without ECN) improves performance of query traffic, it makes performance of short transfers worse, due to queue build up. Third, while RED improves performance of short transfers, it does not improve the performance of query traffic, due to queue length variability.*

We have tried other variations of our benchmarks - either by increasing the arrival rate of the flows or by increasing their sizes. We see that DCTCP consistently ourperforms TCP.

5. RELATED WORK

The literature on congestion control is vast. We focus on recent developments most directly related to our work. Application-level solutions to incast problem include jittering responses (§ 2.3.2) or batching responses in small groups. These solutions avoid incast, but increase the median response time. Recently, researchers

[12]CAT4948 does not support ECN, so we can't run DCTCP with it.

[13]Note that with 10X larger queries, the minimum query completion time increases 10X.

have shown [32, 13] that lowering the $RTO_{\min}$ to 1ms, and using high-resolution retransmission timers alleviates the impact of incast-induced timeouts. But this does not prevent queue buildup; and hence does not address latency issues (§ 2.3). DCTCP, *avoids* queue buildup, and hence prevents timeouts.

QCN [24] is being developed as an optional standard for Ethernet. QCN requires hardware rate limiters (implemented on the NICs). This adds to hardware complexity, and hence increases server cost. To reduce cost, QCN rate limiters must lump flows into (a single or few) flow sets, which then share fate, leading to collateral damage to flows that do not share bottleneck links with congested flows. Moreover, QCN cannot cross layer-3 boundaries, which abound in many data centers today. Special transport schemes for data centers or even optical networks like E-TCP [14] seek to maintain high utilization in face of small buffers, deliberately inducing loss, and not reacting to it. E-TCP does not address delays experienced by short flows or incast.

Several TCP variants aim to reduce queue lengths at routers: delay based congestion control (e.g., Vegas [5] and high speed variants such as CTCP [31]), explicit feedback schemes (e.g., RCP [6], and the ECN-based VCP [34] and BMCC [25]) AQM schemes (e.g., RED [10] and PI [17]). In data centers, queuing delays are comparable to sources of noise in the system, hence do not provide a reliable congestion signal. We have shown that AQM schemes like RED and PI do not perform well in absence of statistical multiplexing. Schemes like RCP, VCP and BMCC require switches to do more complex operations, and are not commercially available.

DCTCP differs from one of the earliest ECN schemes, DECbit [28], in the way AQM feedback is smoothed (filtered) across time. In DECbit, the router averages the queue length parameter over recent cycles, while DCTCP uses a simple threshold and delegates the smoothing across time of the feedback to the host (sender).

Much research has been devoted, with success, to improving TCP performance on paths with high bandwidth-delay product, including High-speed TCP [8], CUBIC [30] and FAST [33]. While many of these variants respond less drastically to packet loss, just like DCTCP does, they do not seek to maintain small queue length and their analysis often assumes a high degree of statistical multiplexing, which is not the norm in a data center environment.

6. FINAL REMARKS

In this paper, we proposed a new variant of TCP, called Data Center TCP (DCTCP). Our work was motivated by detailed traffic measurements from a 6000 server data center cluster, running production soft real time applications. We observed several performance impairments, and linked these to the behavior of the commodity switches used in the cluster. We found that to meet the needs of the observed diverse mix of short and long flows, switch buffer occupancies need to be persistently low, while maintaining high throughput for the long flows. We designed DCTCP to meet these needs. DCTCP relies on Explicit Congestion Notification (ECN), a feature now available on commodity switches. DCTCP succeeds through use of the multi-bit feedback derived from the series of ECN marks, allowing it to react early to congestion. A wide set of detailed experiments at 1 and 10Gbps speeds showed that DCTCP meets its design goals.

Acknowledgments

Mohammad Alizadeh is supported by Caroline and Fabian Pease Stanford Graduate Fellowship. Balaji Prabhakar thanks Abdul Kabbani for collaborating on the ECN-hat algorithm [18] which influenced the development of DCTCP. We thank our shepherd Srini Seshan and SIGCOMM reviewers whose comments helped us improve the paper.

7. REFERENCES

[1] P. Agarwal, B. Kwan, and L. Ashvin. Flexible buffer allocation entities for traffic aggregate containment. US Patent 20090207848, August 2009.
[2] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM*, 2008.
[3] M. Alizadeh et al. Data Center TCP (DCTCP). Technical report.
[4] G. Appenzeller, I. Keslassy, and N. McKeown. Sizing router buffers. In *SIGCOMM*, 2004.
[5] L. Brakmo, S. O'Malley, and L. Peterson. TCP Vegas: New techniques for congestion detection and avoidance. In *SIGCOMM*, 1994.
[6] N. Dukkipati, M. Kobayashi, R. Zhang-Shen, and N. McKeown. Processor sharing flows in the internet. In *IWQOS*, 2006.
[7] S. Floyd. RED: Discussions of setting parameters. http://www.icir.org/floyd/REDparameters.txt.
[8] S. Floyd. RFC 3649: HighSpeed TCP for large congestion windows.
[9] S. Floyd, R. Gummadi, and S. Shenker. Adaptive RED: An algorithm for increasing the robustness of RED's active queue management. Technical report, ACIRI, 2001.
[10] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM ToN*, 1993.
[11] S. Floyd and V. Jacobson. The synchronization of periodic routing messages. *IEEE/ACM ToN*, 1994.
[12] A. Greenberg et al. VL2: A scalable and flexible data center network. In *SIGCOMM*, 2009.
[13] R. Griffith, Y. Chen, J. Liu, A. Joseph, and R. Katz. Understanding TCP incast throughput collapse in datacenter networks. In *WREN Workshop*, 2009.
[14] Y. Gu, D. Towsley, C. Hollot, and H. Zhang. Congestion control for small buffer high bandwidth networks. In *INFOCOM*, 2007.
[15] C. Guo et al. Bcube: High performance, server-centric network architecture for data centers. In *SIGCOMM*, 2009.
[16] J. Hamilton. On designing and deploying Internet-scale services. In *USENIX LISA*, 2007.
[17] C. V. Hollot, V. Misra, D. Towsley, and W.-B. Gong. On designing improved controllers for AQM routers supporting TCP flows. In *INFOCOM*, April 2001.
[18] A. Kabbani and B. Prabhakar. In defense of TCP. In *The Future of TCP: Train-wreck or Evolution*, 2008.
[19] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. The nature of datacenter traffic: Measurements and analysis. In *IMC*, 2009.
[20] F. Kelly, G. Raina, and T. Voice. Stability and Fairness of Explicit Congestion Control with Small Buffers. *CCR*, July 2008.
[21] R. Kohavi et al. Practical Guide to Controlled Experiments on the Web: Listen to Your Customers not to the HiPPO. *KDD*, 2007.
[22] D. Leith, R. Shorten, and G. McCullagh. Experimental evaluation of cubic-TCP. In *Proc. Protocols for Fast Long Distance Networks 2007*, 2007.
[23] Y.-T. Li, D. Leith, and R. N. Shorten. Experimental evaluation of TCP protocols for high-speed networks. *IEEE/ACM Trans. Netw.*, 15(5):1109–1122, 2007.
[24] R. Pan, B. Prabhakar, and A. Laxmikantha. QCN: Quantized congestion notification an overview. http://www.ieee802.org/1/files/public/docs2007/au_prabhakar_qcn_overview_geneva.pdf.
[25] I. A. Qazi, L. Andrew, and T. Znati. Congestion control using efficient explicit feedback. In *INFOCOM*, 2009.
[26] G. Raina, D. Towsley, and D. Wischik. Part II: Control theory for buffer sizing. *CCR*, July 2005.
[27] K. Ramakrishnan, S. Floyd, and D. Black. RFC 3168: the addition of explicit congestion notification (ECN) to IP.
[28] K. K. Ramakrishnan and R. Jain. A binary feedback scheme for congestion avoidance in computer networks. *ACM Trans. Comp. Systems*, 1990.
[29] J. Rothschild. High performance at massive scale: Lessons learned at facebook. mms://video-jsoe.ucsd.edu/calit2/JeffRothschildFacebook.wmv.
[30] I. R. Sangtae Ha and L. Xu. Cubic: A new TCP-friendly high-speed TCP variant. *SIGOPS-OSR*, July 2008.
[31] K. Tan, J. Song, Q. Zhang, and M. Sridharan. A Compound TCP Approach for High-speed and Long Distance Networks. In *INFOCOM*, 2006.
[32] V. Vasudevan et al. Safe and effective fine-grained TCP retransmissions for datacenter communication. In *SIGCOMM*, 2009.
[33] D. X. Wei, C. Jin, S. H. Low, and S. Hegde. FAST TCP: motivation, architecture, algorithms, performance. *IEEE/ACM ToN*, Dec. 2006.
[34] Y. Xia, L. Subramanian, I. Stoica, and S. Kalyanaraman. One more bit is enough. In *SIGCOMM*, 2005.