Investigating DCTCP Beyond the Data Center

Introduction

Modern data centers are characterized by high-speed links in dense leaf-spine topology, extremely low and stable latency, and shallow-buffered switches. These networks are prone to sudden bursts in traffic, heavy background traffic, and an in-cast traffic pattern where many senders simultaneously target a single receiver which can overwhelm small switch buffers.

While standard TCP protocols like CUBIC use binary signals like a single packet loss to trigger aggressive window reductions, DCTCP uses proportional feedback loop using the Explicit Congestion Notification (ECN) bit to calculate the fraction of congestion (α). During initialization, the connection starts with a high α and monitors the network. Once per RTT, the algorithm updates α using an exponentially weighted moving average: α = (1 - g) * α + g * F, where F is the fraction of packets marked in the last window and g is the smoothing parameter that determines reaction speed. When congestion is detected, DCTCP scales its window reduction based on α. Rather than cutting the window in half, it sets the slow start threshold (ssthresh) to ssthresh = cwnd (1 - α / 2). This makes DCTCP maintain a small, stable queue depth without overflowing the buffer.

DCTCP is designed to be highly sensitive to every ECN marked packet which makes it very fragile when exposed to the non-idealities of the public internet, such as high jitter, and variable round-trip times. This paper investigates whether the DCTCP can be optimized to handle these impairments through systemic parameter tuning or fundamental algorithmic modifications.

Method

We implemented a modular framework using Mininet on a GCP hosted Linux VM to evaluate congestion control algorithms across various network environments. The framework leveraged the native kernel-level implementations of congestion control algorithms, mainly DCTCP and CUBIC. We instantiated a star topology to simulate many-to-one in-cast pattern and a dumbbell topology to measure shared bottleneck performance. Workloads were generated using iperf3 to simulate long-lived elephant flows and bursts of mice traffic, representing typical data center and wide-area workloads. Network environments like ECN awareness, delayed acks, and queue management (QM) policies were configured using tc (Traffic Control) utilities. The experiment also analyzed our own variations of DCTCP algorithms by overriding the kernel-level implementation to study any improvements.

Experiment Setup 1: DCTCP in the Data Center

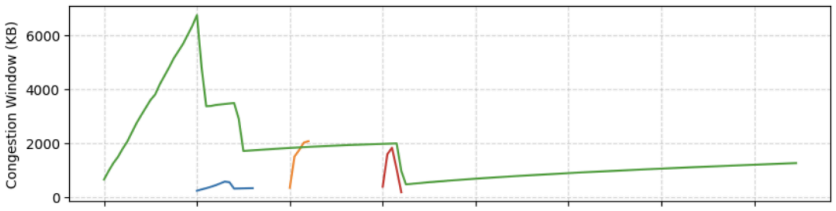
We utilize a star topology to represent multiple senders communicating with a Top-of-Rack switch linked to a single receiver. The bottleneck link between the switch and receiver is 950 Mbps, the propagation delay on the link is 500us (for ~1ms RTT), and the packet queue size is 100 packets. Our default DCTCP uses g = 4 (default) and our switch ECN configuration sets marking zones K with K_{min}=30 and K_{max}= 31. DCTCP assumes a step function for the switch's ECN configuration, so K_{min} and K_{max} are roughly equal [1].

DCTCP versus CUBIC

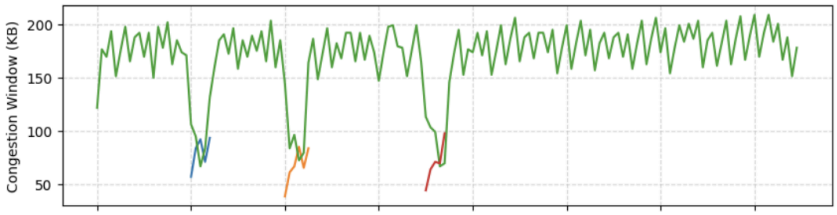
Unlike standard TCP protocols like CUBIC slashes its congestion window by 30% upon a single loss or ECN mark [2], DCTCP tracks the fraction of packet marked with ECN bit and adjusts the congestion window proportionally. This allows DCTCP to maintain a high throughput while leaving more room in switch buffers

to absorb bursts. CUBIC is often too aggressive for data centers since it causes a massive throughput drop on loss, and its recovery is too slow to re-utilize the link effectively.

We ran elephant-vs-mice traffic with 1 elephant and 3 mice on our star topology with two configurations: DCTCP senders, switch with ECN enabled, and receiver with ECN-Echo; and CUBIC senders, switch with Taildrop QM and no ECN enabled, and receiver without ECN-Echo. Compared to CUBIC, DCTCP's proportional response yields an 88% reduction in RTT CoV (coefficient-of-variation; the lower the CoV, the less variance), a 97% reduction in P99 RTT, and a 76% reduction in congestion window CoV for the elephant flow. Crucially, these stability wins are achieved with zero retransmissions and only a negligible 2% reduction in average throughput compared to CUBIC. In the graphs below, we compare congestion windows of DCTCP and CUBIC. CUBIC follows the usual sawtooth pattern, while DCTCP keeps a much tighter bound on the congestion window. This was consistent with the original paper [1].



Graph 1: CUBIC congestion window over time in the datacenter.



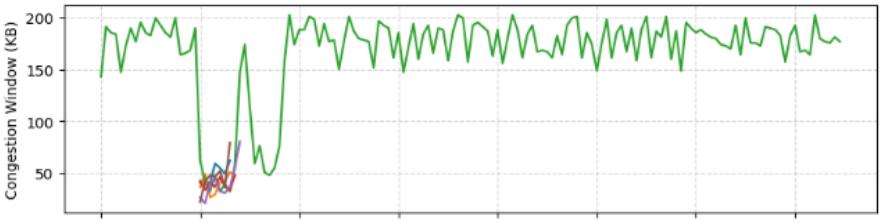
Graph 2: DCTCP congestion window over time in the datacenter.

Introduction of External Factors

The precision of DCTCP is highly dependent on a stable feedback loop, which can be disrupted by real-world network "noise" as seen in some of the experiments we run below.

Delayed ACKs

In standard TCP, delayed ACKs are commonly used to improve efficiency by reducing traffic on the reverse path. However, for DCTCP, they introduce a feedback granularity problem, since it needs to know exactly which packets were marked with the ECN bit. In our in-cast scenario with 5 simultaneous mice and 1 elephant flow, delayed ACKs cause a significant regression in fairness and predictability. Mice flows experience a 35-165% increase in throughput CoV and up to a 90% increase in RTT CoV. As seen in the graph below, the elephant flow also takes significantly longer to recover its window, as the feedback granularity required to calculate α is inaccurately halved and it oscillates between too aggressive or too passive.



Graph 3: DCTCP congestion window over time in the datacenter with delayed ACKs.

Jitter

Jitter temporarily confuses DCTCP's control loop. Since α is updated once per RTT, jitter leads to inaccurate α and oscillations. Adding 200μs of jitter to our 3-mice, 1-elephant experiment results in three out of four mice suffering from a 20% reduction in average throughput. The elephant flow sees an 8% increase in tail RTT and the emergence of non-zero retransmissions, as jitter-induced ACK bunching triggers false congestion signals.

Random Packet Loss

For DCTCP, actual packet loss is a "critical failure" indicating the ECN loop was too slow. Any packet loss is assumed to mean that the switch queue is full and the switch has started dropping packets. Since DCTCP tries to keep the queue size small, it takes packet loss as a sign of high congestion. When we added a 0.1% random loss rate to our 3-mice, 1-elephant experiment it introduced a jagged pattern in the window. It was forced to exit its stable proportional mode, with the elephant flow experiencing an 18% increase in RTT CoV and a 30% increase in window CoV.

Improving DCTCP with External Factors

Given DCTCP's reactive behavior towards external factors, we decided to explore potential ways to stabilize it in this new environment.

Parameter Tuning

We first focused on tuning the available parameters for DCTCP, which are the smoothing parameter g and the K_{min} / K_{max} in the ECN configuration for the switch. Increasing g can help filter out the noise caused by jitter, forcing the algorithm to focus on long-term queue trends rather than fleeting bursts. while increasing K_{min} provides some buffer that accommodates larger bursts and prevents random noise from triggering ECN marks. However, testing of over 20 combinations of K_{min} / K_{max} and g values revealed that parameter tuning alone is insufficient. The "best" tuned configuration (g=4, K=30/30) still resulted in a 10% throughput loss and a 43% increase in window CoV compared to the noise-free baseline.

Changing DCTCP Congestion Control Algorithm

We realized a more intrusive approach might be required to help DCTCP handle the new noise better, so we tried multiple modifications of DCTCP's congestion control algorithm. Modern implementations of DCTCP already improve resilience against delayed ACKs by using byte counts instead of ACK counts for α calculation, which mitigates the impact of delayed ACKs. Therefore, we focused mostly on reducing α sensitivity to low jitter and random packet loss in our changes.

The first iteration, *dctcp_v1*, addressed the fundamental inability of DCTCP to differentiate between actual congestion and random packet loss. To prevent the elephant flow from collapsing under a 0.1% loss rate, which normally triggers a 50% window cut every few milliseconds, *dctcp_v1* reduced the congestion window by only 1/8 instead of 1/2 if packet loss occurred while the α value remained low (<5%). Furthermore, the switch threshold was increased to K=60/61 to provide an additional buffer against jitter.

In *dctcp_v2*, to counter high-frequency jitter oscillations, the smoothing gain was increased to g=6. The threshold for gentler congestion window reduction was lowered from 5% to 1% to ensure that even minor real congestion triggered a serious back-off, addressing the retransmission spikes seen in *dctcp_v1*. *dctcp_v2* also replaced the flat 1/8 reduction with a scaled reduction. During loss events, the window reduction was scaled proportionally to α, mirroring normal DCTCP behavior but remaining less aggressive than Reno, DCTCP's fallback congestion control algorithm.

The final iteration, *dctcp_v3*, attempted to accelerate recovery for the elephant flow. Since a high g value provides stability, it also causes α to decay slowly even after congestion has cleared, potentially keeping the elephant flow throttled after mice flows have finished. Instead, in *dctcp_v3*, if the number of ECN-marked bytes in an RTT dropped to zero (no congestion detected), the algorithm switched to a lower g (g=4); if ECN marks were present, it remained at g=5 (a compromise from g=6) for stability.

For our comparison, we first ran the default DCTCP with all external noise (delayed ACKs, 200μs jitter, and 0.1% packet loss) as a baseline. Results of all three versions are shown below.

Version	**Change**	**Results (compared to default DCTCP with all noise)**
DCTCP v1	Gentler cwnd reduction from ½ to ⅛.	Sawtooth pattern for congestion window, increased retransmissions, lenient to actual congestion events, constant bufferbloat
DCTCP v2	Scaled cwnd reduction by increasing g=6	Stabilized elephant throughput, significant reduction in reduced transmissions, faster mice completion times, slow elephant recovery
DCTCP v3	Faster recovery by dynamic smoothing.	Faster elephant recovery, mice retransmissions remained high because elephant did not back off quickly enough for sudden bursts

Conclusion

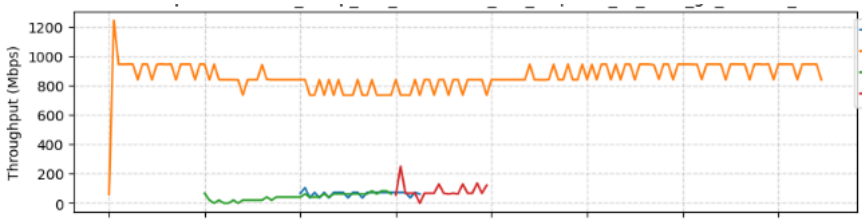
We see a 40-60% improvement in average throughput for two mice flows with *dctcp_v2*, but also a 29% increase in congestion window CoV, 14% increase in p99 RTT, and 16% increase in throughput CoV for our elephant flow compared to default DCTCP with all external factors. However, *dctcp_v2* actually generally performs worse than the default DCTCP. It seems the main issue in our approach with these changes is that we assume that a low α means that congestion cannot be happening. We allow α to change more slowly, which means that real congestion, which can occur extremely fast in this low-latency, shallow-buffer environment, is not being accounted for anymore. Further work might look into trying to use RTT trends to better differentiate between loss and congestion.

Experiment Setup 2: DCTCP beyond the Data Center

Moving DCTCP beyond the data center and into the Wide Area Network (WAN) introduces a much higher Bandwidth-Delay Product (BDP). In these environments, the RTT is significantly higher (e.g., 20ms vs. 1ms) so the feedback loop is much slower. By the time a sender receives an ECN mark, the congestion event that triggered it may have already passed, or alternatively, the queue may have already overflowed into massive packet loss. For our experiments beyond the data center, we utilize a dumbbell topology to represent congestion between two switches on the path from source to receiver. The bottleneck link between the two switches is still 950 Mbps, the propagation delay on the link is 10ms (for ~20ms RTT), and the packet queue size is 200 packets.

Tuning DCTCP

In the WAN, our data center default DCTCP's marking threshold of K_{min}=30 severely underutilizes the link. Because the BDP is high, K_{min}=30 represents a tiny fraction of the necessary in-flight data.K must be scaled linearly with the BDP to ensure the marking point occurs only when the physical link is actually congested. We did some experimentation to find the best g and K values, and decided to use g = 6 and K_{min} / K_{max} = 200/201. A 3 mice and 1 elephant traffic on this dumbbell topology resulted in:

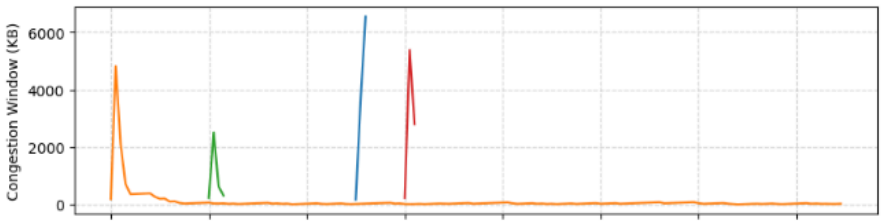


Graph 4: DCTCP throughput over time beyond the datacenter.

We can see that the elephant flow's throughput is fairly stable, though the mice seem to take longer to complete (a result of increasing g). The RTT also took longer to stabilize but eventually reached stability at a little over 20ms. We observe a slight creep in RTT at the end because the queue builds gradually over time due to the sender's slower response to congestion.

Introduction of External Factors

The combination of high RTT and noise creates a highly unfavorable environment for DCTCP. We demonstrate this by introducing delayed ACKs, 2ms jitter, and 0.1% packet loss into our dumbbell topology. In this scenario, as can be seen below, the elephant flow essentially dies because DCTCP interprets every ECN mark and every packet loss as a definitive signal to reduce sender rate. The sender is forced into a state of permanent back-off, and this instability is further worsened by jitter, which makes the estimation of α highly unstable. Since the RTT is 20 ms, by the time the sender realizes a packet has been lost, it has already added hundreds of unnecessary packets into the switch buffer. The result is a collapse of the elephant's congestion window to almost zero, where it remains stuck.



Graph 5: DCTCP congestion window over time beyond the datacenter with all "noise".

Changing DCTCP Congestion Control Algorithm

We experimented with modifying DCTCP to better support the longer RTT and unexpected noise. The first iteration, *dctcp_v4*, attempted to adapt *dctcp_v2* to the WAN by increasing the threshold for gentler congestion window reduction to 5%, accounting for the higher volume of non-congestion loss.

To speed up recovery, *dctcp_v5* introduced custom congestion avoidance logic that we hoped would be more resilient than standard Reno. Reno increases the window by only 1/cwnd per ACK; when combined with delayed ACKs, this growth rate is effectively halved. In our approach, while the slow-start phase remained the same, DCTCP would double its number of counted ACKs if α was very low (<2.5%). The goal was to amplify the signal of no congestion, so DCTCP would ramp up quicker.

Finally, *dctcp_v6* attempted to prevent the window from collapsing when α was low by increasing the noise threshold within ssthresh. If α was below 5%, the algorithm could only reduce the window by up to two packets, based on the assumption that the loss was random noise rather than a full buffer. Results of all three versions are shown below.

Version	Change	Results (compared to default DCTCP with all noise in Graph 5)
DCTCP v4	Adjusted gentler cwnd reduction	80% reduction in mice RTT CoV, 15% reduction in mice p99 RTT, 48% decrease in elephant cwnd CoV, 21% decrease in elephant RTT CoV, elephant flow killed by slow speed of loss recovery rather than noise
DCTCP v5	Custom congestion avoidance	12% improvement in elephant throughput, 9% drop in elephant window CoV, 4% drop in elephant throughput CoV, 30% improvement in mice flow completion times, up to 80% decrease in mice RTT CoV; elephant flow still unable to sustain high throughput
DCTCP v6	Increased noise threshold	10% improvement in elephant throughput, 9% reduction in elephant RTT CoV, up to 80% reduction in mice RTT CoV, up to 62% reduction in mice congestion window CoV, up to 97% regression in mice throughput; elephant flow still collapsed

Conclusion

Congestion control algorithms like CUBIC are designed to handle a 30-50% reduction in window size upon loss, but DCTCP does not have the foundation necessary to handle this additional noise. Frequent packet loss means DCTCP spends all of its time trying to recover from an initial stall and never successfully transitions back to congestion avoidance, regardless of the smoothing or threshold parameters used.

Conclusion and Related Work

The experimental results from both the low-noise data-center topology and high-noise WAN topology suggest that DCTCP is essentially infeasible for general-purpose internet or WAN traffic. Its fundamental design assumes an accurate, low-noise signal. When jitter, 0.1% loss, and delayed ACKs are introduced, the proportional nature of DCTCP that brings it huge wins in the data center becomes its downfall outside of the data center. It becomes too sensitive to signals that don't represent real congestion. Though not presented, we also found in our experiments that DCTCP easily succumbs to smaller CUBIC flows, allowing these smaller flows to starve a larger DCTCP flow.

To address these limitations in heterogeneous networks, recent research has shifted toward the L4S (Low Latency, Low Loss, and Scalable Throughput) architecture [3]. The core of this framework is the Dual Queue Coupled AQM, which is designed to allow Scalable traffic (like DCTCP) to coexist with Classic traffic (like CUBIC) without the performance trade-offs we have observed in our experiments. L4S provides a path towards an efficient DCTCP beyond the data center.

How We Used AI

We used AI to generate graphs, parse data, and visualize our Mininet topology.

References

[1] **Alizadeh, M., et al. (2010).** "Data Center TCP (DCTCP)." *ACM SIGCOMM Computer Communication Review*.
[2] **Ha, S., Rhee, I., & Xu, L. (2008).** "CUBIC: A New TCP-Friendly High-Speed TCP Variant." *ACM SIGOPS Operating Systems Review*, 42(5), 64-74. DOI: 10.1145/1400097.1400105.
[3] **Briscoe, B., et al. (2023)**. "Dual-Queue Coupled Active Queue Management (AQM) for Low Latency, Low Loss, and Scalable Throughput (L4S)." RFC 9332.