Towards Timeout-less Transport in Commodity Datacenter Networks

Hwijoon Lim[1] Wei Bai[2] Yibo Zhu[3] Youngmok Jung[1] Dongsu Han[1]
[1]KAIST [2]Microsoft Research [3]ByteDance

Abstract

Despite recent advances in datacenter networks, timeouts caused by congestion packet losses still remain a major cause of high tail latency. Priority-based Flow Control (PFC) was introduced to make the network lossless, but its Head-of-Line blocking nature causes various performance and management problems. In this paper, we ask if it is possible to design a network that achieves (near) zero timeout only using commodity hardware in datacenters.

Our answer is TLT, an extension to existing transport designed to eliminate timeouts. We are inspired by the observation that only certain types of packet drops cause timeouts. Therefore, instead of blindly dropping (TCP) or not dropping packets at all (RoCEv2), TLT proactively drops some packets to ensure the delivery of more important ones, whose losses may cause timeouts. It classifies packets at the host and leverages color-aware thresholding, a feature widely supported by commodity switches, to proactively drop some less important packets. We implement TLT prototypes using VMA to test with real applications. Our testbed evaluation on Redis shows that TLT reduces 99%-ile FCT up to 91.7% on handling bursts of SET operations. In large-scale simulations, TLT augments diverse datacenter transports, from widely-used (TCP, DCTCP, DCQCN) to state-of-the-art (IRN and HPCC), by achieving up to 81% lower tail latency.

CCS Concepts: • **Networks** → **Transport protocols**.

Keywords: datacenter networking, low-latency transport, RoCE, TCP

ACM Reference Format:
Hwijoon Lim, Wei Bai, Yibo Zhu, Youngmok Jung, Dongsu Han. 2021. Towards Timeout-less Transport in Commodity Datacenter Networks. In *Sixteenth European Conference on Computer Systems (EuroSys '21), April 26–29, 2021, Online, United Kingdom.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3447786.3456227

EuroSys '21, April 26–29, 2021, Online, United Kingdom
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8334-9/21/04…$15.00
https://doi.org/10.1145/3447786.3456227

1 Introduction

Datacenter networks today are mostly built using Ethernet. Its best-effort nature offers attractive advantages. Ethernet scales to datacenter sizes with lower hardware cost and coordination overhead than its reservation-based counterparts. However, unfortunately packet losses due to (transient) congestion became a major cause of high tail latency in production networks [17, 27].

As an (immature) answer to this problem, the community has experimented with "lossless" Ethernet. It eliminates packet drops by triggering Priority-based Flow Control (PFC) before switch buffers overflow. However, it was soon discovered that it causes many side effects from mild (e.g., congestion spreading and unfairness [58]) to severe (e.g., PFC storm and even deadlock [32, 38]). Fundamentally, the Head-of-Line (HoL) blocking nature of PFC violates the best-effort nature of Ethernet, and thus makes large-scale networks hard to understand and manage. Consequently, recent literature [24, 34, 56] shifts back to lossy Ethernet and designs new mechanisms to remove PFC. However, they all require hardware features that are not yet available. The requirement of new hardware features means that legacy deployment cannot use them, and future deployments may not adopt them due to cost issues.

In this paper, we take a step back from the debate between PFC (for lossless networks) and inventing new hardware features (for lossy networks). Instead, we ask, is there a third direction that gets the low tail latency of lossless networks, keeps best-effort nature of lossy networks, and is compatible with today's hardware and existing protocols such as TCP and RDMA over Converged Ethernet (RoCE) v2?

Our answer is positive. We note not all packet drops have equal performance impacts. In some cases, the sender can quickly detect and recover from a loss. In others, the sender may suffer from a timeout. Only the latter seriously impacts the application performance [27]. Inspired by this observation, we believe timeoutless network is the right direction to pursue. To this end, we design TLT[1], a new transport building block using commodity hardware. TLT ensures timeouts do not occur with high probability, while tolerating the drops that can be recovered with fast retransmission. This enables TLT to keep the best-effort nature of Ethernet, while having the tail latency of lossless networks.

At the switch, TLT ensures that congestion drops of packets whose losses may cause timeouts do not happen with high

[1]TLT stands for Timeout-Less Transport.

probability, while exposing the rest to a lossy environment and even dropping them aggressively (to leave headroom to hold more important packets). Then TLT transport guarantees loss detection and recovery succeed with very high probability. TLT only leverages features of commodity switch hardware for deployment friendliness. TLT is designed to be compatible with popular datacenter transport protocols, including DCTCP and RoCEv2, with minimal modifications, making it easier for operators to adopt.

TLT is not an independent transport protocol—instead, it is a building block for existing and future transports. TLT mitigates the harm caused by transient micro-bursts that traditional congestion control often cannot control, but is not a replacement for congestion control or loss recovery. Finally, TLT only concerns congestion losses; losses due to failures and problematic hardware are beyond its scope.

TLT's design involves addressing several challenges. First, we want "important" packets, whose losses may cause transport layer timeouts, to be treated differently at the commodity switches, but at the same time ensure packets are not reordered. Second, we want to mark only a small fraction of packets as "important", while preserving the timeoutless property. We do not want to overrun switch buffers with excessive important packets. Finally, being generic is nontrivial. Indeed, we show different strategies are required in selecting important packets for window- and rate-based congestion control protocols due to their distinct behavior. We demonstrate TLT is applicable to diverse datacenter transports from widely-used TCP, DCTCP [17], and DCQCN [58] to state-of-the-art protocols such as IRN [43] and HPCC [41].

Contributions. TLT is the first mechanism that uses commodity switches to enable timeout-less transport. TLT virtually eliminates timeouts, the primary cause of high tail latency, by making certain important packets lossless. We demonstrate that TLT is generally applicable to many datacenter transport protocols. Our extensive testbed experiments and simulations demonstrate TLT drastically reduces or even eliminates timeouts in a wide variety of realistic traffic mixes. Our testbed evaluation shows TLT reduces 99%-ile FCT (flow completion time) up to 97.2% under incast scenarios. In our simulation, TLT reduces the tail latency by up to 81% in realistic workloads for DCTCP, TCP, DCQCN, IRN, and HPCC.

2 Background and Motivation

2.1 Impact of Timeouts

Although bandwidth is plentiful in datacenters, congestion packet losses are still not uncommon due to the prevalence of microbursts [57] and shallow-buffered switches [23]. As flows come and leave frequently, most congestion incidents in datacenters are short-lived (100s of μs). Traditional datacenter congestion controls [17, 41, 42, 52, 58] require a round-trip-time (RTT) to detect and react to congestion. Hence, they are often too slow to react to microbursts [57], causing timeouts that are harmful to real-time datacenter workloads, including search [17], social networking [50], and retail.

The impact of losses depends on how fast they are recovered. When packets in the middle of a message get lost, the receiver immediately observes out-of-order arrivals. However, when the tail packet, an entire window of data or acknowledgments is lost, timeouts occur. Compared to fast recovery, timeouts are much more detrimental because retransmission timeout (RTO) is set conservatively to avoid spurious retransmissions. For example, TCP sums the RTT estimate with 4x the RTT variance to estimate RTO and has a minimum bound. In Linux, *the lowest possible minimum RTO* is 1 jiffy, which is 4 ms by default in recent kernels [9].

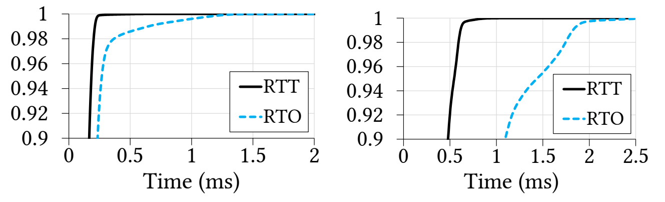
2.2 Using Aggressive Timeout

A seemingly straightforward solution is to use an aggressive RTO; one may use a small minimum RTO (RTO_{min}) [54] or even a small static RTO instead of estimating it. We present reasons why they may not be very effective.

Traffic dynamics lead to a large estimated RTO. Vasudevan et al. [54] proposed using microsecond granularity timers and 100s of microseconds RTO_{min} to avoid incast collapse. However, we find that, even with this solution, the estimated RTO may be still very large because traffic in production datacenters is volatile and bursty [57]. Today's datacenter switching chips [1, 4–6] use dynamic buffer allocation [26] to absorb bursts, thus further increasing the range of RTT variations. For example, Broadcom Trident II [5] has a 12 MB shared buffer and 32 40 GbE ports. During a high-degree incast, RTT may spike to 2.4 ms (12 MB/40 Gbps). Transient bursts also cause queuing and processing delay in middleboxes; the SLB in Microsoft adds a median latency of 196us, while the 90th percentile can reach 1 ms [30].

We use simulations to validate this. We set RTO_{min} to 200μs. We model the shared-buffer switch and generate realistic workloads mixing background flows (Poisson arrival) and incast-like foreground flows (on/off arrival). The complete settings are given in §7.1. Figure 1 shows the cumulative distributions of RTT and estimated RTO under realistic workloads. Though the average network utilization is only 40%, congestion still happens when many foreground flows arrive simultaneously. We find that more than 10% of foreground flows have RTOs larger than 1.1 ms, much larger than RTTs (0.48 ms at 90th percentile). This indicates a small RTO_{min} is not a fundamental solution. In fact, our evaluation (§7) shows that reducing RTO_{min} actually produces worse results than using PFC to make the network lossless.

Aggressive static timeouts may be harmful. A more aggressive approach is to use a small static timeout. Although this can guarantee fast loss recovery, it seriously degrades (DC)TCP's throughput because upon a timeout the congestion window drops to one. We run the same simulation but



(a) Background Flows (b) Foreground Flows

Figure 1. [Simulation] Distribution of RTT and calculated RTO for DCTCP. RTO_{min} is 200μs.

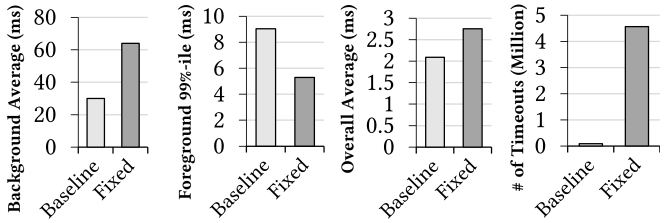


Figure 2. [Simulation] FCTs with fixed RTO of 160 μs. Baseline uses 4 ms RTO_{min}. Transport is DCTCP.

using a small fixed RTO of 160 μs (twice the base RTT). Figure 2 shows flow completion times (FCTs) evaluated with a fixed RTO, compared to the baseline using 4ms RTO_{min}, when the total volume of the foreground traffic is 15%. The 160 μs fixed RTO improves 99%-ile FCTs of foreground flows by 41%. But this gain comes with a large penalty of 113% increase in background average FCTs (and 31% degraded goodput for background flows) due to a 51-fold increase in timeouts.

Undesirable interactions with network diagnosis. A more serious problem of aggressive RTOs is the undesirable interactions with existing datacenter diagnosis systems. Network diagnosis systems use timeouts and retransmissions as key features [22] to infer the root cause of failures or trigger further diagnostic actions upon detecting retransmissions [21]. Excessive spurious timeouts and retransmissions may generate many spurious alerts and interfere with diagnosis, thus degrading network manageability.

Current production practice. Given the above reasons, to the best of our knowledge, most datacenter network operators (including us) still use traditional conservative methods to estimate RTO and configure minimum RTO to several milliseconds for intra-DC traffic.

2.3 Existing Solutions

Loss recovery. Studies on loss recovery are vast, so we only discuss a few representatives here. Tail Loss Probe (TLP) [27] reduces timeouts by converting RTOs into fast recoveries by transmitting a loss probe packet. IRN [43] proposes an improved RoCE NIC design with selective retransmissions. Existing solutions only target a specific protocol and eliminate timeouts for some types of losses. TLP targets TCP, but cannot prevent timeouts once probe packets are lost. TLP and others suffer from timeouts when the retransmit sequence is lost. In contrast, TLT generalizes the idea of converting timeout into fast recovery for both TCP and RoCE-based protocols leveraging features in commodity switches.

Priority-based Flow Control (PFC). Unlike the above *best effort* solutions, PFC [8] makes networks lossless, thus eliminating congestion timeouts. *Even before RoCE deployments, we have deployed PFC with TCP to mitigate incast for a latency-sensitive online service in our datacenters.* PFC is now widely used to support RoCE deployments [32].

However, many side effects manifested when running PFC at scale, ranging from mild (e.g., congestion spreading and unfairness [58]) to severe (e.g., PFC storm and even deadlocks [32, 38]). The main cause is that when PFC pauses ingress ports, it induces collateral damage to other flows traversing the same ingress port but destined to different egress ports. Although congestion control [41, 42, 58] can mitigate some issues, buffer overrun cannot be prevented, especially when many new flows arrive simultaneously.

Novel switch support. Recent research efforts leverage new switch features to provide rapid loss notifications. When a switch queue fills, CP [24] drops packet payloads, but not packet headers, thus achieving a lossless network for metadata. Lossless metadata gives the receiver a complete picture of packet drops. NDP [34] further improves CP to reduce feedback delay and mitigate phase effects [29]. FastLane [56] enhances switches to explicitly send high-priority drop notifications to sources. Although these efforts show promise, they require modifications to switching chips [1, 4–6] and are not readily deployable in production datacenters.

3 TLT Design Overview

Design goals. We focus on reducing tail latency of short bursty flows, often observed from the traffic in user-facing services (e.g., RPC messages), while having a minimal impact on throughput. Motivated by the limitations of existing work, our goal is to design a solution with four properties:

- **Timeout mitigation:** To satisfy the stringent latency requirement of real-time applications, the scheme must *guarantee* congestion packet losses are quickly recovered without triggering timeouts with *a high probability*.
- **Deployment friendliness:** As a new switching chip often takes years to design and implement, our scheme should leverage features of commodity switching chips that are widely deployed in production datacenters.
- **Generality:** The solution should be compatible with a variety of transport protocols and applicable to both window- and rate-based protocols.
- **Minimal side effects:** We must minimize the side effects, such as congestion spreading, deadlocks, and adverse interactions with underlying congestion control.

Design rationale. We observe that certain packets are more *important* (e.g., last packet in the message) than others as their losses are likely to cause timeouts. Inspired by this, we design TLT that selects important packets at the end host and gives these important packets preferential treatment in the network such that they experience near-zero loss, while leaving the rest of the packets subject to loss as usual. This involves solving two main challenges:

- **How to selectively drop packets inside a switch queue?** TLT gives preferential treatment to important packets and drops unimportant ones. To make it readily deployable in production datacenters, the color-aware dropping mechanism should be implemented using commodity switching chips without adversely impacting existing transport.
- **How to select important packets?** Important packets are packets whose losses may cause timeouts. TLT should select as few as possible packets to be important. As a transport building block, TLT must work with various datacenter transport protocols, including TCP and RoCE.

4 TLT Switch Design

4.1 Color Aware Dropping

A naïve approach for enabling preferential treatment to important packets is to use separate queues for important packets and unimportant packets. However, this will incur severe reordering, causing a negative impact on transport protocols. Hence, we aim to realize selective dropping inside a switch queue. To this end, we leverage *color*, a feature that is widely supported by commodity switching chips. Color is a metadata that the chip pipeline attaches to every packet. Most commodity switching chips (e.g., Broadcom Tomahawk and Trident series [1, 4–6]) support three colors, red, yellow, and green, allowing operators to assign packet colors by programming the access control list (ACL).

A buffer threshold can be associated with each color within the same queue. Then the queue build-up for each color is limited to the threshold given. This is known as color aware dropping, originally developed for traffic metering and quality of service (e.g., Two Rate Three Color Marker meters) [16, 35, 36]. In our production datacenters, the feature is not being used for any purpose. To our best knowledge, we are the first to use color aware dropping in conjunction with transport protocols to enhance their performance.

Our idea is to use color aware dropping to "reserve" room for important packets in the egress queue. Marking unimportant packets as red, for example, and setting a color-aware dropping threshold for red packets enable us to limit the queue buildup for unimportant packets. To avoid dropping important packets, TLT allows important packets to queue beyond the color-aware dropping threshold, while limiting the threshold for unimportant packets. We further enable dynamic buffer allocation [26] to accommodate important packets (see §4.2). Finally, the FIFO nature of the queue preserves the ordering of packets.

4.2 Choice of the Dropping Threshold

We refer to the threshold given to unimportant packets as the *color-aware dropping threshold*. When new flows arrive in burst, we want the color-aware dropping to take effect to throttle the queue and protect important packets because traditional congestion control cannot take effect immediately. During this time, color-aware dropping must offer *a fast response to short-term congestion*. However, when congestion controls reach the steady state, color-aware dropping should not interfere or take effect. We can ensure this by setting the dropping threshold larger than the *maximum steady state queue size*. We derive the bounds for widely-used transports:

- **TCP:** To avoid throughput loss, the color-aware dropping threshold must be larger than the bandwidth-delay product (BDP), given the low degree of multiplexing of large concurrent flows in production datacenters [17] (i.e., the large buffer rule rather than the small buffer rule [20]).
- **DCTCP:** In DCTCP [17], all packets are marked if the instantaneous queue length exceeds the single ECN marking threshold, K_{ECN}. At steady state [17, 18], the queue length oscillates around K_{ECN} where the amplitude of oscillation depends on the number of concurrent flows. However, DCTCP falls back to vanilla TCP in the presence of packet losses. Thus, to avoid throughput loss, we should set the color-aware dropping threshold larger than BDP [20]. The minimum color-aware dropping threshold for DCTCP is $max(K_{ECN}, BDP) = BDP$ since $K_{ECN} <$ BDP [17, 18, 55],
- **DCQCN(+IRN):** DCQCN uses RED-like probabilistic ECN marking [58]. It has three switch parameters: minimum threshold (K_{min}), maximum threshold (K_{max}) and maximum probability (P_{max}). Zhu et al. [59] prove that DCQCN has a unique fixed point queue length between K_{min} and K_{max} at steady state. Therefore, the color-aware dropping threshold should be at least as large as K_{max}. Note DCQCN does not adjust the rate when packet loss occurs.

We note, while TLT works with various transport protocols, ones that use less buffer at steady state work better with TLT since they leave larger room for important packets. We characterize the tradeoff for a range of thresholds in §7.

When does TLT drop important packets? Using a model of a shared buffer switch, we characterize when a switch buffer might overflow and drop important packets. Our result below shows TLT ensures lossless delivery of important packets in the vast majority of cases, except for extremes (e.g., 10k concurrent flows at a switch).

To model a switch, we denote the total size of its buffer as B, of which UB is unallocated at a given moment; number of ports as N; and the color-aware dropping threshold as K. The switch buffer is shared by all ports using a dynamic threshold algorithm [26], which uses the parameter, α. An arriving packet to egress queue i is dropped if $Q_i \geq \alpha \times UB$, where Q_i is the length of queue i. Assuming the switch has M ports experiencing congestion simultaneously, each port gets $\frac{\alpha \times B}{1+M\times\alpha}$ [26] from the shared buffer pool and has $\frac{\alpha \times B}{1+M\times\alpha} - K$ to hold important packets. A large α maximizes buffer utilization, while a small α favors short-term fairness among ports [26]. We use α to 1 to balance the trade-off.

We now demonstrate the unlikeliness of important packet loss using numbers from a real ToR switch. Arista 7050QX-32 [2] (based on Broadcom Trident II [5]) with 12 MB shared buffer and 32 40 GbE ports. We assume an extreme case where half the ports ($M = 16$) are congested simultaneously. Given $\alpha = 1$, each port has $\frac{1 \times 12\,MB}{1+16\times 1} = 705.88\,KB$ buffer. For (DC)TCP, we set the select dropping threshold K to 400 kB (assuming a 40 Gbps network with 80 μs RTT). So the per-port buffer "reserved" for important packets is 305.88 kB. Since TLT has at most one important in-flight packet per flow in our design (§5), each port can hold 203 flows, resulting in 3248 flows in total. If only one port is congested, this port can get $1/2 \times 12\,MB - 0.4\,MB = 5.6\,MB$ buffer to hold important packets, thus handling up to 3,733 flows without important packet losses. This suggests TLT can eliminate important packet losses in the vast majority of cases in production datacenters without using PFC. If an operator wants to *eliminate congestion timeouts under any circumstances*, PFC can be used in conjunction with TLT.

5 TLT Host Transport Design

TLT host ensures the underlying transport protocol does not experience any timeout, assuming important packets are not lost. To avoid important packet drop, a TLT host should mark packets as important as few as possible. In the following, we describe how TLT host selects important packets for window-based (§5.1) and rate-based transports (§5.2).

Despite the differences, there is one common mechanism: all control packets (e.g., SYN, FIN, RST, and pure ACK in TCP; and ACK, NACK, and CNP in RoCE) are marked as important given their special purposes and small size.

We assume packets of a flow traverse the same network path, as most production datacenters use Equal-Cost Multi-Path (ECMP) for load balancing [32, 51]. Since out-of-order delivery is very rare, the duplicate ACK or NACK threshold is set to one to reduce retransmission latency. Note Linux kernel implements early retransmit [19] that requires only one duplicate ACK for fast recovery. In most commercial RoCE implementations [10, 11], when the receiver receives an out-of-order packet, it will send a NACK to trigger retransmissions immediately [32].

Finally, TLT does not explicitly deal with non-congestion losses. When important packets are lost due to non-congestive reasons including problematic hardware (e.g., silent packet drops [33]) or packet corruptions, its performance falls back to the underlying transport.

5.1 TLT with Window-based Transport

This section deals with applying TLT to window-based transports, including but not limited to TCP (and its variants such as DCTCP [17]), HPCC [41], and IRN [43]. Protocols with a static window also fall into this category. For example, IRN uses a static window with a size of bandwidth-delay product.

Key Idea: Window-based transports use a sliding window to bound the number of outstanding packets. Packet transmission is self-clocked; an old packet leaving the network triggers an ACK, and the ACK slides the sender's window to inject new packets into the network pipe. They suffer from timeouts if self-clocking is broken or a loss happens at the tail of a flow [27]. Thus, maintaining self-clocking is critical.

A straw man approach to keep self-clocking is to mark the last packet of every window as important. Although this idea seems promising, it will end up marking all packets as important, as every newly transmitted packet is actually at the tail of the current window at that instant.

Instead, TLT uses a novel self-clocking approach to keep one important in-flight packet.

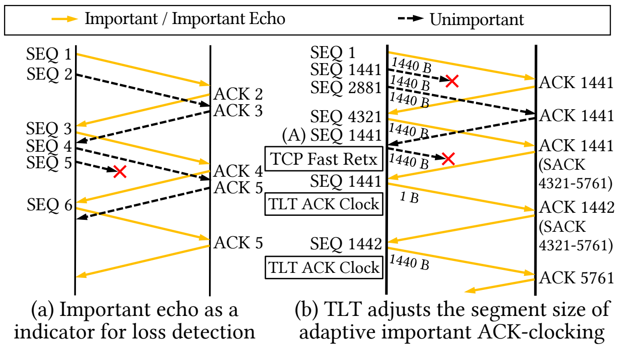
- When a new flow starts, the sender marks the last packet in the initial window as `Important Data`.
- When the receiver receives an `Important Data` packet, it sends an ACK (or SACK) *immediately* marked as important like the other control packets. However, to differentiate with acknowledgements for unimportant packets, we mark this packet with a special tag, `Important Echo`.
- When the sender receives an ACK or SACK with `Important Echo`, it means the only in-flight important data packet has left the network, leaving self-clocking vulnerable. Hence, it transmits another `Important Data` packet immediately.

This ensures there is always one in-flight important packet (either `Important Data` or `Important Echo`) for each flow and enables fast loss detection.

Guaranteed fast loss detection. TLT's important packets naturally act as a reliable *indicator* for loss. Every RTT, an `Important Data` and an `Important Echo` are guaranteed to arrive at the receiver and sender, respectively. Once the sender receives a `Important Echo`, it can detect the losses of unimportant packets sent between two important packets.

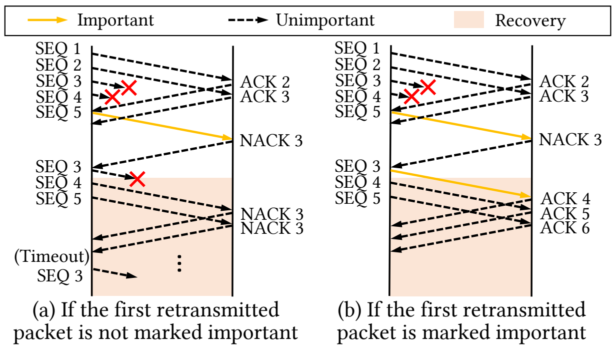
Figure 3 (a) illustrates how `Important Echo` serves as an indicator. Note that SEQ N is acknowledged by ACK N+1. There are three `Important Data` (SEQ 1, SEQ 3, and SEQ 6) and three `Important Echo` (ACK 2, ACK 4 and the second ACK 5) in total. When the sender receives the second ACK 5, it can tell if packet losses happen between two important packets: SEQ 3 and 6, thus detecting the dropped SEQ 5.

Guaranteed self-clocking via important ACK-clocking. When an ACK marked as `Important Echo` opens up a new window, TLT transmits a data packet as `Important Data`. However, a problem arises when the new window does not allow any data transmission, as this prevents further ACK-clocking. This may happen due to window reduction or no available new data in the send buffer. Normally, in this case, sending an extra packet may worsen congestion. However, TLT has already reserved room for important packets at the



(a) Important echo as a indicator for loss detection (b) TLT adjusts the segment size of adaptive important ACK-clocking

Figure 3. TLT on window-based transport ensures loss detection and introduces adaptive important ACK-clocking.



(a) If the first retransmitted packet is not marked important (b) If the first retransmitted packet is marked important

Figure 4. TLT on rate-based transport marks the first retransmitted packet as important.

switches. So we choose to inject an important packet regardless of window or buffer limit to ensure the liveness of ACK-clocking. We call this *important ACK-clocking*.

Nonetheless, important packets also take up buffer space, so we must minimize their footprint. A straightforward solution is transmitting the first sent but unacked byte. This also avoids the loss masking problem [27] where the redundant packet fully repairs the loss and masks it from congestion control. However, this significantly delays loss recovery. Figure 3 (b) illustrates the problem when packet SEQ 1441 (with payload 1441-2880) is lost. We assume SACK is enabled and the initial window is 3. At (A), the duplicate ACK 1441 triggers a retransmission, which is lost again. When the loss is detected, TLT triggers important ACK-clocking with 1 byte payload, which gets delivered and its ACK triggers another important ACK-clocking. As a result, it will take 1440 RTTs to fully recover all the lost bytes (1441 to 2880).

Thus, we use *adaptive* important ACK-clocking that balances the trade-off and minimize adverse interactions with congestion control, as illustrated in Figure 3 (b).

- When an important ACK-clocking is caused by an `Important Echo` that indicates any loss, the sender transmits the first maximum segment size (MSS) worth of lost data as `Important Data` to speed up loss recovery.
- When the `Important Echo` does not indicate any loss, the sender transmits the first unacked byte in the window, to minimize its footprint while ensuring the ACK-clocking.

We also design mechanisms to handle duplicated ACK generated by important ACK-clocking to prevent potential interference with congestion control. For the full algorithm and benefits of each design choice, refer to Appendix A, which summarizes the behavior of a TCP host implementing TLT.

Remark: `Important Data` and their echoes are transport layer message types. They are sent with a network layer tag that switches recognize as "important".

5.2 TLT with Rate-based Transport

Rate-based transport protocols, such as DCQCN, use the internal timers or rate-limiters to explicitly control inter-packet transmission intervals. Commercial RoCE implementations adjust the sending rate based on advanced congestion signal [42, 58] and perform go-back-N [32] recovery in the presence of packet losses. TLT ensures timely loss detection and recovery by carefully selecting important packets.

Timely loss detection. Given that rate-based transports *continuously* transmit packets, we select the last packet of a flow as important. This is because as long as the last packet arrives at the receiver, the receiver can always detect out-of-order arrivals (if packet losses happen) and notify the sender to retransmit (e.g., via NACK [32]). For long flows, however, if all unimportant packets are lost, loss detection may take long. Thus, for each flow, we can further mark an additional important packet in every N transmitted packets[2]. To ensure that important packets do not persistently overwhelm the link capacity, N should be larger than the fan out degree, e.g., the maximum number of hosts sending traffic to a receiver simultaneously in a datacenter application.

Timely loss recovery. However, marking only the last packet is insufficient. A loss of first retransmitted packet is a special case we need to handle. To demonstrate the problem, Figure 4 illustrates a flow with five packets, out of which packets 3 and 4 are lost. TLT transmits packet 5 as important, which generates a NACK indicating packet 3 is missing. This in turn triggers a retransmission of packet 3, which unfortunately is lost again. This triggers another NACK at the receiver. However, after receiving the second NACK 3, the sender cannot distinguish it from the first NACK 3 and hence cannot determine if packet 3 is lost again. As a result, the sender does not start a new round of retransmissions until the retransmission timer expires. To address this, when the rate-based transport starts a round of retransmission, TLT host marks both the first and the last packet as important.

5.3 Discussion

TLT mitigates congestion timeouts at the cost of (unimportant) packet losses and retransmissions. For latency-sensitive workloads, this is acceptable because TLT trades them to avoid the most harmful important packet losses. In addition,

[2]Periodic marking is *optional*, and is an aid for timely loss detection. Thus we can set N to a large value conservatively based on the coarse-grained prior knowledge. We also find TLT is not sensitive to N, as tail FCT differs less than 3% between N=96 and N=384 in our large scale simulations.

unimportant packets are only proactively dropped when congestion does happen, thus avoiding spurious retransmissions which interfere with network diagnosis and management.

Impact on throughput-sensitive workloads. However, the impact of TLT on throughput-sensitive workloads, e.g., data backup and replication, is complex. On the one hand, TLT improves throughput compared to PFC. This is because TLT proactively drops packets to avoid triggering PFC, thus mitigating congestion spreading. On the other hand, TLT slightly degrades throughput as it causes more packet losses and retransmissions. Coarse-grained congestion control and retransmission mechanisms, e.g., go-back-N, will worsen this problem. Therefore, the overall impact of TLT on throughput depends on multiple factors.

To reach a deep and comprehensive understanding, we consider more challenging scenarios in our evaluation with a mix of latency-sensitive and throughput-sensitive traffic. However, we note a potential alternative is to selectively enable TLT for latency-sensitive workloads. Depending on deployment models, operators can use multiple queues to isolate TLT traffic at the switch and only enable color-aware dropping on the queue with TLT traffic; or enable TLT on dedicated clusters only running real-time applications.

Deployment in the public cloud. Our current design is targeted to private datacenters where operators can control both host network stacks and switches. In multi-tenant public cloud, deploying TLT is challenging as operators cannot control tenants' network stacks in the virtual machines. A potential solution is using NetKernel [46] to provide a TLT-enabled network stack as a service to tenants.

Masking packet losses. One potential problem of the approach is masking losses, which has been identified by TLP [27]. For example, consider a two-packet flow whose first packet is marked important. When the second packet gets lost, upon receiving the `Important Echo` of the first packet, TLT will transmit the first byte of the second packet (important ACK-clocking), detect the lost second packet, and then retransmit the second packet. This process successfully masks the packet loss from congestion control.

However, we think TLT's masking losses does not cause much impact for two reasons. First, TLT only masks losses, but does not impact other congestion signals, such as ECN, delay and INT, which datacenter congestion control mainly relies upon. Second, this masking losses problem only happens upon meeting *all* of the three conditions: the last segment of the flow/message is lost, all lost segments are consecutive, and all lost segments reside in the window and are subsequent to the last important packet. And even when the loss is masked, there are no further packets to send by the time the last segment is recovered. Thus, the 'wrong' congestion window does not take any effect.

Incremental deployment. We believe TLT can be deployed incrementally. When TLT is partially deployed, TLT can be used for communication between TLT-enabled machines. At the switch, we can use a dedicated queue to carry TLT-enabled traffic and enable color-aware dropping. Note, non-TLT packets must use a separated queue without color-aware dropping, as it will drop the non-TLT packets (classified as unimportant), leading to performance degradation.

Holistic transport design. TLT is a building block for low-latency datacenter transport. In this paper, we demonstrate its generic applicability by using TLT to augment legacy transports. We believe co-designing TLT with congestion control algorithms is a promising direction.

6 Implementation and Testbed

End-host. We implement TLT on Mellanox Messaging Accelerator (VMA) [12] to accelerate (DC)TCP. VMA is a dynamically-linked user-space Linux library that transparently accelerates socket applications. Unlike RDMA, VMA uses LwIP [28] to implement the stack in user space. Compared to mTCP, VMA provides much lower latency and *does not require any application modification, thus allowing us to run TLT with real applications.* Vanilla VMA only supports basic TCP features, such as New Reno with fast retransmission and timestamp options. We implement SACK and DCTCP on VMA with 481 lines of code (LOC). To test 200 μs RTO_{min}, we also add a high resolution timer with 10 μs granularity into the time subsystem of VMA. The VMA implementation modifies 941 LoC to implement Algorithm 1 in Appendix A.

TLT uses the DSCP field to differentiate important and unimportant packets. We set RTO_{min} to 4 ms unless otherwise noted and initial window to 10, same as Linux default, and use a Linux-like RTO calculation.

Testbed. We connect 9 servers to a 40 GbE Netberg Aurora 720 switch running Openswitch 2.0.5 [3] switch operating system. The switch uses Broadcom Tomahawk chip that has 16 MB shared buffer[3] and 32 40 GbE ports. Each server has either a single Mellanox ConnectX-5 or ConnectX-4 40 Gbps NIC, and 4-6 physical CPU cores.

Switch configuration. For packet classification within the switch, we associate a color to each DSCP value and map important packets to green and unimportant to red. We enable dynamic buffer allocation [26], which will allocate up to ~1.8MB buffer to a single busy port if the other ports are idle. We set the color-aware dropping threshold to 270 KB, which is close to BDP of our testbed. For DCTCP, we set the ECN marking threshold to 200 KB, according to its guideline [17].

7 Evaluation

We evaluate TLT using testbed experiments and large-scale NS-3 simulations. We briefly summarize our main findings:

[3]Tomahawk has 4 memory management units (MMU), each with 4 MB buffer which is dynamically allocated within each MMU. Each egress port is mapped to two MMUs.

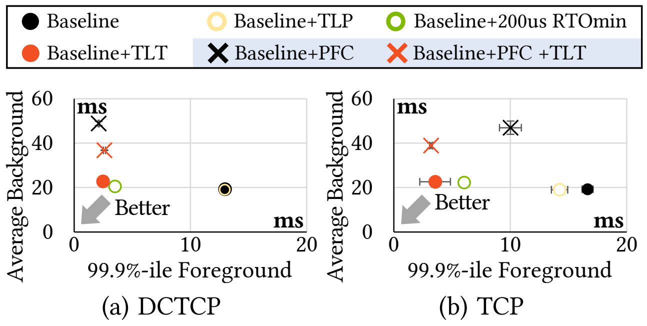


Figure 5. FCT for TCP and DCTCP. Load is 40%. 5% is foreground traffic. Color-aware dropping threshold is 400 kB. The baseline is 4 ms RTO_{min}.

- Our large-scale simulations (§7.1) show TLT virutally eliminates timeouts. When TLT is applied to DCTCP and IRN, it reduces the 99.9%-ile FCT of foreground incast flows by up to 80.9% and 69.1% respectively.
- We characterize TLT's performance under a wide range of threshold setting and network load (§7.2), demonstrating the benefit of timeout mitigation in various settings.
- In application benchmarks (§7.3), we show TLT can reduce the maximum application-level response time by up to 91.5% and 91.7% for in-memory cache with DCTCP and TCP, respectively.
- In testbed experiments (§7.4), we show TLT eliminates timeouts and reduces the 99%-ile FCT by up to 96.8% for DCTCP and 97.2% for TCP under incast scenarios.

7.1 Large-scale Simulations

We demonstrate TLT benefits a wide range of transports from widely deployed schemes, such as DCTCP and DCQCN, to state-of-the-art designs, such as HPCC and IRN, through extensive large-scale simulations. Our results show TLT significantly reduces the foreground tail latency up to 80.9%, without harming background flows.

Settings. We use NS-3 [14] to simulate a 96-host leaf-spine topology with 4 core switches, 12 ToR switches, and 8 servers per ToR switch. The over-subscription ratio is 2:1. Each link has 40 Gbps capacity and 10 μs latency (BDP=80 $\mu s\times$ 40 Gbps=400 kB). For RoCE (DCQCN, IRN, and HPCC), we use 1 μs latency to reflect its low-latency design. To emulate Broadcom Trident II [5] (12 MB buffer and 32 40 GbE ports) which is widely used in our production datacenters, we allocate 12 ports and 4.5 MB total buffer to each switch. We implement buffer mechanisms based on the chip specification and set dynamic threshold parameter α [26] to 1 so that each egress queue can get at most 50% of shared buffer. For RoCE evaluation, we implement TLT, SACK, IRN on the top of HPCC's NS-3 simulator [7]. We also extend HPCC's INT implementation to support 40Gbps.

Benchmark workloads. Based on traffic patterns observed in our datacenters, we model a setting where incast-like foreground traffic serves user-facing workloads, while competing with background traffic. Background flows are sent between a pair of random sender and receiver under a Poisson process, whose size distribution follows the "background traffic" in a web search service [17] with an average flow size of 1.72 MB. We create 10 k background flows for each simulation. Foreground flows are incast traffic from 95 senders to a single receiver. Each sender creates 8 flows of 8 kB. The average network utilization (load) of the links between ToR and core is 40%. We adjust the load of foreground flows by changing the incast frequency. Unless otherwise noted, foreground flows take up 5% of total traffic volume.

Baselines and metrics. For TCP-based transports, we use vanilla TCP (NewReno) and DCTCP. We evaluate their performance with various different loss recovery mechanisms: RTO_{min} of 4 ms (Linux default, baseline in the Figure 5), TLP [27], and 200 μs (high performance timer [54]). We set TLP's minimum probe timeout (PTO) to 10 μs. We enable SACK and set duplicated ACK threshold to 1.

For RoCE-based transports, we choose HPCC, vanilla DCQCN, DCQCN with IRN, and DCQCN with SACK (IRN without the static window limit). Vanilla DCQCN uses go-back-N for loss recovery while the others use SACK. All the schemes except for IRN use a static RTO of 4 ms. For IRN, we use 1930 μs (base latency plus maximum one-hop queueing delay) for RTO_{High} as recommended [43]. For the other parameters, we use the recommended settings in [41, 58].

We set TLT's color-aware dropping threshold for TCP-based transports to 400 kB (equal to BDP). We use 200 kB for RoCE-based transports, which is larger than the steady queue length of DCQCN and HPCC (the recommended K_{max} of DCQCN is 200KB [58]). When we run TLT with vanilla DCQCN, we additionally mark a packet as important for every 96 packets, according to the largest fan-out degree in our topology (§5.2). We run simulations five times with different random seeds and report the average and standard deviation of tail FCT for foreground interactive flows and average FCT for background flows.

Result with TCP-based transports. Figure 5 shows the tail FCT of foreground flows and average FCT of background flows of TCP and DCTCP. We make two key observations:

First, for DCTCP, when PFC is enabled, the tail FCT of foreground flows is significantly reduced compared to baseline (13.0 ms to 2.11 ms), but at the cost of a significant increase (19.3 ms to 48.8 ms) in the FCT of background flows. Background flows often become victims because aggressive foreground flows trigger many PFC PAUSE frames, thus blocking background flows from the same ingress port. TLT, on the other hand, does not have this problem. With or without PFC, the foreground tail FCTs are very similar for TLT.

Second, in lossy networks without PFC, TLT improves the 99.9%-ile tail FCT of foreground flows by 80.9% compared to the original DCTCP (4 ms RTO_{min}) with a slight increase in FCT of the background. TLT also outperforms 200 μs RTO_{min}

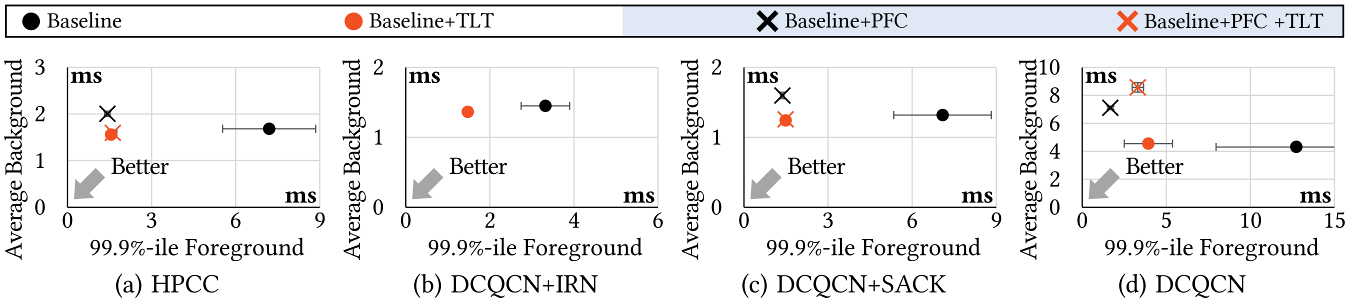


Figure 6. FCT for HPCC and DCQCN (vanilla, with SACK, with IRN). Load is 40%. 5% is foreground traffic. Color-aware dropping threshold is 200 kB.

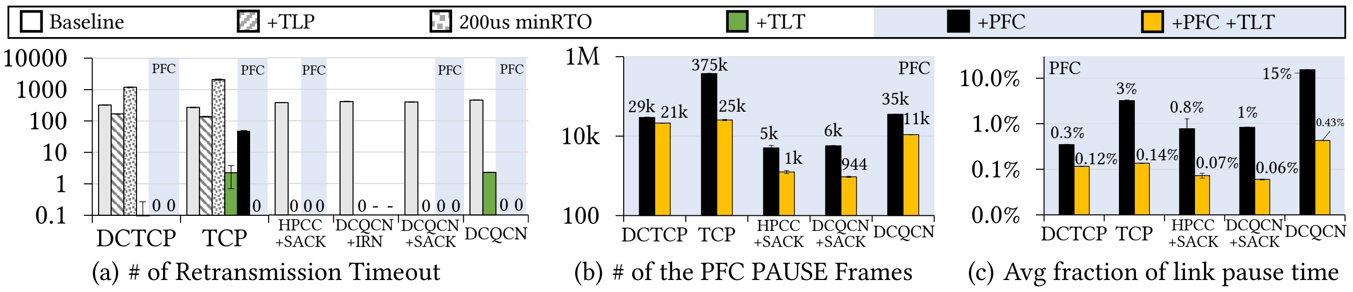


Figure 7. # of timeouts per 1 k flows, # of PAUSE frames per 1 k flows, and the average fraction of link PAUSE time. Load is 0.4, and foreground flows takes 5% of total traffic.

in tail FCTs by up to 29.6%. In contrast, TLP is not very effective in reducing the foreground tail FCT because timeouts can still happen once probe packets are dropped.

Note that trends of result with TCP are similar to those of DCTCP, except that the foreground tail FCT significantly increases with TCP+PFC. This is because TCP background flows can fill the switch buffer to generate many PFC pause frames, thus also blocking foreground flows.

To demonstrate TLT virtually eliminates timeouts, we measure the number of timeouts in Figure 7 (a). DCTCP+TLT nearly eliminates timeout. For TCP+TLT without PFC, only 0.25% of flows experience timeouts, showing the effectiveness of our timely recovery. In contrast, using high-performance timer caused 3.70 times more timeouts (DCTCP). TLP cuts timeouts but 52.2% of timeouts still remain (DCTCP), adversely impacting the tail FCT. Finally, for TLT with DCTCP and TCP, the loss rate of important packets is only 1.33×10^{-7} and 4.30×10^{-6}, explaining why timeouts are rare.

Figure 7 (b) shows the number of PFC PAUSE frames, which TLT reduces by 27.7% for DCTCP and 93.2% for TCP. Without TLT, PFC causes HoL blocking, thus adversely impacting the FCT of background flows. Finally, Figure 7 (c) shows the average fraction of time a link is blocked due to PFC PAUSE. TLT reduces the blocked time by 66.7% for DCTCP and 95.8% for TCP. This demonstrates color-aware dropping is better than blindly ensuring a lossless network.

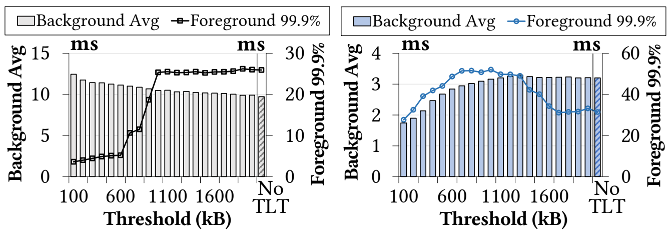
Result with RoCE-based transports. Figure 6 shows the FCT results for HPCC (with SACK), DCQCN with IRN, DCQCN with SACK, and vanilla DCQCN. Except for IRN, we evaluate it with and without PFC.

For HPCC, TLT reduces 99.9%-ile foreground FCT by 78.5% and the average background FCT by 70.5%, when PFC is not used. This is because HPCC cannot handle packet bursts in the first RTT, despite its near zero steady queuing and fast convergence. With PFC, TLT reduces average FCT of background flows by 20.5%, while not degrading the foreground flows heavily. Note although HPCC was designed to be used in lossless network, HPCC with TLT on lossy network shows comparable performance as in lossless network.

For IRN, TLT improves its 99.9%-ile FCT of foreground flows by 55.62% and the average FCT of background flows by 5.92%, as TLT eliminates timeouts as shown in Figure 7 (a). An interesting comparison is between IRN and vanilla DCQCN with PFC. As shown in Figure 6, Compared to RoCE with PFC, IRN achieves comparable tail FCT for foreground flows, and significantly improves throughput with 66.4% improvement on average background FCT. This is consistent with the observation of IRN [43].

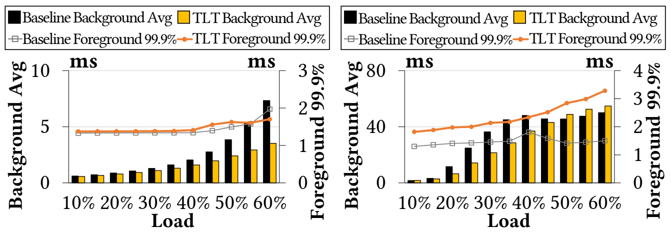
Without PFC, TLT on vanilla DCQCN reduces the 99.9%-ile FCT of foreground flows by 69.1% while slightly increasing the average background FCT only 5.60%. However, with PFC, TLT fails to show performance gain, due to inefficient recovery and congestion control of vanilla DCQCN. On DCQCN with SACK, which has better recovery, TLT reduces the average FCT of background flows by 21.38%, by reducing the number of pause frame by 83.5% and the paused time by 92.8% (Figure 7 b and c). *This also indicates that TLT cannot replace the need of efficient loss recovery.*

Does TLT ensure timely loss recovery? We measure packet delivery time: the time from the first transmission trial to its ACK arrival including all retransmissions if they happen. TLT reduces the 99%-ile latency by 22.8% and 99.9%-ile latency by 57.6% (Figure 16 in Appendix B). Figure 16



(a) DCTCP + TLT (b) DCTCP + TLT +PFC

Figure 8. 99.9%-ile foreground FCT and average background FCT by color-aware dropping threshold.



(a) HPCC + PFC (b) DCTCP + PFC

Figure 9. 99%-ile foreground FCT and average background FCT by different network load.

shows the result for DCTCP without PFC, but we observe similar trends with other transports.

7.2 Deep Dive

In this section, we dig deeper into TLT's design using a series of targeted simulations.

How does the color-aware dropping threshold impact performance? The threshold setting impacts performance in two different directions. On the one hand, as Figure 8 (a) shows, a smaller threshold triggers more (unimportant) packet drops, thus impacting throughput-sensitive long (background) flows. On the other hand, as Figure 8 (b) shows, a larger threshold increases the queue occupancy, thus causing more important packet drops or PAUSE frames.

Figure 8 (a) shows the FCT of foreground and background flows without PFC, for a wide range of threshold settings, for DCTCP. A larger threshold increases the foreground tail FCT, but decreases the average background FCT. If the threshold exceeds 700 kB, we start to observe the effect of timeouts at the tail due to important packet drops.

With PFC (Figure 8 (b)), both foreground tail and background FCT increase with a larger threshold at the beginning. This is because, with a larger threshold, queue size grows and PFC is triggered more often, thus adversely impacting the background FCT. Average background FCT keeps increasing with a larger threshold, but foreground tail FCT starts to decrease after 900 kB, reversing the trend. This is because frequent and wide-spread PAUSE triggers start to victimize background flows so severely and give bandwidth to foreground flows [32, 58].

How sensitive is TLT to network load? As the network load increases, TLT tends to drop more unimportant packets, thus causing more retransmitted packets and rate reductions. Thus, the benefit of TLT may diminish as load increases. Note that production datacenters typically operate at very low loads, e.g., 99% of all links are typically less than 10% loaded [50]. Figure 9 gives the results of tail foreground FCT and average background FCT of HPCC with PFC and DCTCP with PFC. We vary the load from 10% to 60%. We find the sensitivity to the load depends on the transport.

One line of transports, e.g., HPCC, does not reduce the rate in the presence of losses. As shown in Figure 9 (a), when used with HPCC, TLT +PFC keeps low tail FCTs for foreground flows at all the loads, as in pure PFC. In addition, TLT achieves consistently lower average FCT for background flows by mitigating the HoL blocking impact of PFC. The improvement of TLT even increases at higher loads (e.g., 51.9% at 60% load). This is because, at high loads, the penalty of packet retransmissions is much smaller than that of HoL blocking. TLT achieves a good trade-off at high loads.

A second line of transports, e.g., TCP and DCTCP, reduces the rate upon detecting packet losses. Therefore, TLT causes larger performance penalty. Figure 9 (b) shows that DCTCP+PFC benefits from TLT on loads lower than 50%. When the load is higher than 50%, TLT results in a larger average FCT of background flows. This is because at high loads, the penalty of packet retransmissions and throughput losses is larger than that of HoL blocking.

How does the fraction of important packets vary? The fraction of important packets depends on the workload and the color-aware dropping threshold. First, we vary the workload by changing the ratio of incast foreground traffic up to 20%. Figure 10 shows that TLT generates only a small fraction (3.29% by volume) of important packets without foreground traffic. As foreground traffic increases, the fraction of the important packet increases. This is because short foreground flows have a higher fraction of important packets, and a larger fraction of foreground traffic increases congestion and cuts down the congestion window.

Second, we vary the color-aware dropping threshold while fixing the ratio of foreground traffic to 5% in Figure 11 (a). With a 400 KB threshold, 5.90% of packets are marked as important. A smaller color-aware dropping threshold causes loss to a greater fraction of unimportant packets, which triggers retransmissions marked as important.

Does TLT keep queues small? Figure 11 (b) shows the maximum queue size with DCTCP. The ECN marking threshold is set to 200 KB following the DCTCP recommendation. However, without TLT, the maximum queue length reaches 2.18 MB due to the bursty flow arrival. In contrast, TLT effectively controls the queue size and keeps the unimportant queue length under the color-aware dropping threshold. The maximum total queue length is also kept 23.1% lower (with 400 kB color-aware dropping threshold) than vanilla DCTCP.

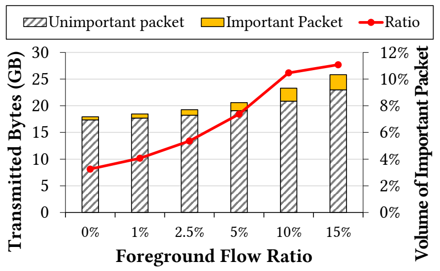
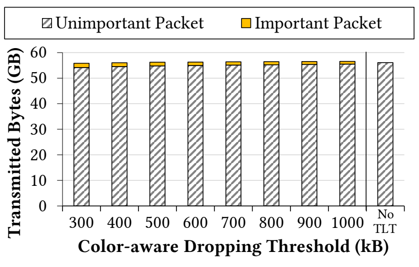
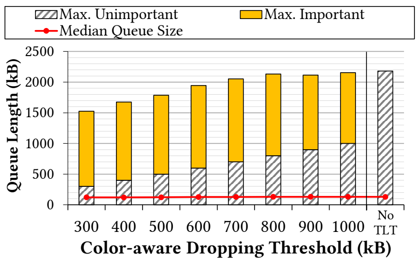


Figure 10. Ratio of important packets by volume for DCTCP+TLT.

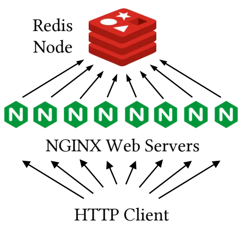


(a) Total bytes transmitted

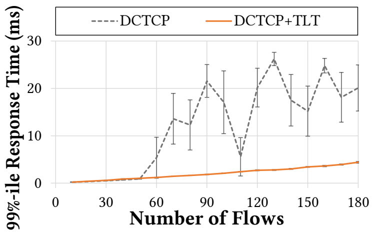


(b) Maximum per-port queue length

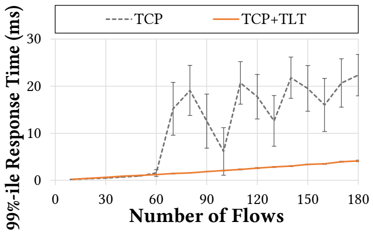
Figure 11. Total bytes transmitted and the maximum per-port queue length for TLT. Base transport is DCTCP.



(a) Topology



(b) 99%-ile Response Time (DCTCP)



(c) 99%-ile Response Time (TCP)

Figure 12. [Testbed] Incast experiment on Redis.

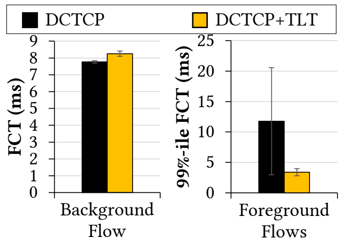


Figure 13. [Testbed] Redis with mixed traffic.

Note the median queue length stays near 130kB, which is lower than the ECN marking threshold.

Additionally, Appendix B presents detailed results that answer the questions below:

- **How often are important packets dropped?** TLT exhibits no important packet drops to loss rate of 3.5x10^{-3} under heavy churn.
- **How effective is adaptive important ACK-clocking?** It achieves fast recovery comparable to full MTU transmission, while using 6.9 times less bandwidth.
- **Is TLT effective with different workloads?** We use three other representative datacenter workloads, and we find that TLT delivers better performance as in §7.1.
- **How does TLT works with different degree of incast?** TLT even works well with higher degree of incast, lowering foreground tail FCT up to 78.9%.

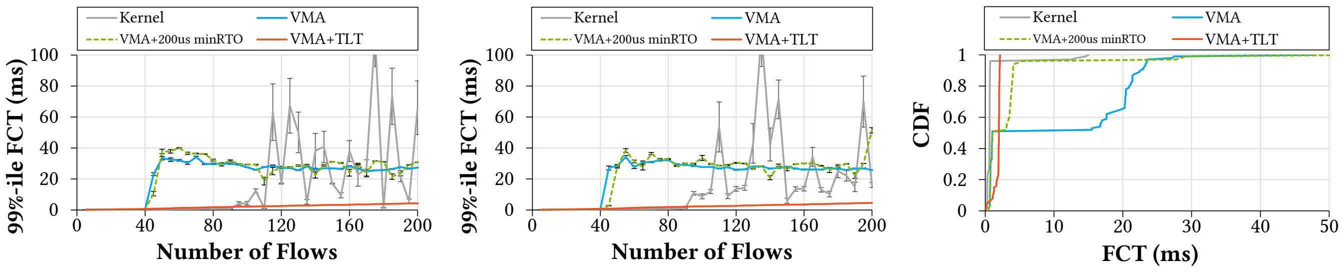
7.3 Application-level Performance

TLT deals with micro-bursts that traditional congestion control cannot handle well. We verify that TLT brings performance improvement on real application through experiments using NGINX and Redis. We look at two such cases.

In-memory caches, e.g., Redis [15] and Memcached [13], are used to reduce the number of disk I/Os. In Facebook production environment [45], all web servers communicate with every Memcached server in a short period of time to satisfy user requests, thus causing incast congestion. To emulate this workload in our 10-node testbed, we set up a HTTP client, eight NGINX web servers, and a Redis node. The client issues up to 180 requests in total at the same time, evenly distributed toward eight web servers. We build persistent connections between web servers and the Redis node, and each HTTP request triggers the web server to send a set operation with 32 KB data to the Redis node. Hence, the requests from the web servers to the Redis node cause incast. We vary the number of HTTP requests to control the incast degree and measure the response time of HTTP messages. We evaluate the application level latency by measuring the client-perceived response time at the HTTP clients.

We enable TLT and (DC)TCP (on VMA) between web servers and the Redis node. Figure 12 (b) and (c) respectively show the average of 99%-ile HTTP response times measured during twelve runs with (DC)TCP. The error bar shows ±0.5×standard deviation. Both DCTCP and TCP show very high variance, depending on how well the requests from web servers to the Redis node happen to be synchronized. DCTCP supports a slightly higher degree of incast than TCP. On the other hand, TLT supports a much higher fan-out. Both TCP and DCTCP with TLT show very steady changes in response time as the number of the flows increases. It maintains a very low tail response time between 213μs and 4.40 ms. The maximum response time improvement over TCP and DCTCP is up to 91.7% and 91.5% respectively. Tail FCTs on (DC)TCP without TLT show much larger variance as the number of timeouts differs depending on how the flows are synchronized, while (DC)TCP with TLT does not experience timeouts. Note TLT slightly worsens the performance of TCP, due to the retransmission caused by color-aware dropping when the number of flows are small.

In-memory cache with mixed traffic. We conduct an experiment that shows TLT minimizes the tail flow completion times of latency-sensitive foreground flows, while minimizing performance degradation of throughput-sensitive background flows. For this, we generate a 8 MB background flow



(a) Tail FCT of DCTCP (b) Tail FCT of TCP (c) CDF of FCT for 100 DCTCP flows

Figure 14. [Testbed] Microbenchmark (Incast experiment) on VMA.

that competes with 152 foreground flows of 32 KB accessing the Redis node from 8 servers. We report the average 99%-ile FCTs from four runs in Figure 13. The error bar shows 1 standard deviation. We find that while DCTCP suffers from high 99%-ile FCT up to 11.3 ms, DCTCP+TLT achieves 3.39ms of 99%-ile FCT (71.2% improvement), which comes with slight decrease (5.58%) in the goodput of the background flow.

7.4 Testbed Microbenchmark Experiments

Using our testbed, we show that TLT handles bursts of small flows without experiencing any timeout. TLT reduces tail FCTs up to 97.2% by effectively eliminating timeout.

Microbenchmark with incast. We create incast and measure the 99%-ile FCT with VMA implementations while increasing the fan out. We use a client which generates synchronized requests for 32 KB of data to the other eight servers. We measure the FCT by the time from the request is transmitted to receiving all data of the response. We establish all TCP connections before conducting each experiment run. We report the average and standard deviation of five runs. For a realistic baseline, we also provide measurements on Linux kernel 5.4.0 with SACK and DCTCP.

Figure 14 (a) and (b) respectively show the 99%-ile FCT for TCP and DCTCP implemented on VMA, customized to have 4 ms and 200 μs RTO_{min}. The error bar shows 1 standard deviation of the entire distribution. Figure 14 (c) shows the CDF of FCT for TCP flows with 100 flows. Both TCP and DCTCP cannot avoid timeouts because the burst flow arrival renders reactive congestion control ineffective. In contrast, TLT accommodates a higher degree incast (at least four times as many flows) gracefully, without introducing any timeout. It allows a flow to progress using the reserved buffer at the switch. Note that kernel measurements show much larger variance and high tail FCTs (beyond 90 flows). This is because kernel TCP flows are less synchronized than the VMA-based implementation. In Figure 14 (c), reducing minimum RTO shows improvement on some flows, but it does not show much improvement on 99%-ile FCT. We find that the aggressive timeout shows unexpected high FCTs from 96%-ile. We suspect this is because consecutive losses of retransmitted packets cause exponential backoff in RTO.

We also measure queueing delay. The maximum queuing delay (not shown in figure) of DCTCP was 262 μs. In contrast, DCTCP+TLT shows lower queuing delay bounded below 173 μs, which is 34% lower than DCTCP.

Mixed traffic with PFC. We run an experiment with a mix of background and foreground flows in a network that uses PFC. We build a dumbbell topology using two switches. For this we add a Netberg Aurora 420 switch with Broadcom Trident II that has 12 MB shared buffer, 48 10GbE + 6 40GbE ports. The link capacity between the two switches is 40 Gbps. Seven hosts (senders) are attached to one switch, and two hosts (receivers) are connected to the other. Six senders generate total 600 flows of 32kB foreground traffic, and the other sender generates a long-running background traffic. We first start the background flow and observe that the goodput closely matches the link capacity. When the foreground flows start, however, the background flow experiences performance degradation due to PFC PAUSEs. We find that the PFC PAUSE duration is reduced from 6.24 ms to 3.26 ms, since TLT alleviates triggering PFC frames by keeping queue short. As a result, the average goodput of TLT is much higher than that of regular DCTCP, demonstrating the effectiveness of TLT.

8 Related Work

"Try and backoff" congestion control. Traditional datacenter congestion control algorithms [17, 40–42, 53, 55, 58] react to various congestion signals, e.g. ECN, delay, and INT, to reduce queue delay, packet losses and/or PFC pause frames. Despite their performance improvement compared to loss-based congestion control, they still suffer from losses, timeouts, and/or PFC storms, when many new flows arrive simultaneously. TLT is complementary to all of them.

Credit-based transports. Recently, many credit-based datacenter transports [25, 31, 34, 44, 47] are emerging. In theory, they can achieve zero congestion packet losses as they *proactively* allocate bandwidth (credits) to schedule packet transmissions. However, credits allocation takes one extra RTT, which may not be acceptable for small flows. To avoid this delay, many solutions [31, 34, 44] actually allow packet transmissions in the first RTT, thus still causing losses and timeouts. NDP [34] mitigates this by using "lossless headers", which requires hardware features that are not widely available on commodity switches. Aeolus [37] enables credit-based transports to use spare bandwidth to transmit the first RTT packets. It de-prioritizes the first RTT packets using selective dropping and uses tail probe packets to effectively detect the first-RTT packet losses. Aeolus can only handle losses in the first RTT while relying on proactive congestion control [25] to eliminate the other congestion losses. TLT and Aeolus target traditional reactive transports and credit-based transports, respectively. Aeolus realizes selective dropping by re-interpreting RED/ECN feature. In contrast, TLT uses color-aware dropping instead as RED/ECN is used by ECN-based congestion control to detect congestion.

Loss recovery. Vasudevan et al. [54] show that reducing minimum RTO can mitigate incast. TLP [27] transmits redundant loss probe packets to avoid timeouts. CP [24] trims packet payloads but not packet headers to achieve rapid loss notifications. FastLane [56] enhances switches to explicitly send high-priority drop notifications to sources. Mittal et al. IRN [43] propose an improved RoCE NIC design to support selective retransmissions.

Flow control. PFC is known to cause diverse issues at scale [32, 58]. Some efforts [38, 39, 48] have been made to study and prevent PFC deadlocks. GFC [48] is a new flow control which manipulates the port rate at a fine granularity. Although the efforts show promise, they cannot change the HoL blocking nature of PFC, thus still leaving many problems unsolved, e.g., congestion spreading and unfairness.

9 Conclusion

TLT presents a timeout-less transport for commodity datacenters. We demonstrate when we prioritize the delivery of certain packets such that they do not experience congestion loss, we can significantly cut down the tail latency by reducing or even eliminate timeouts. We show the idea applies to diverse datacenter transports, while minimizing undesirable interactions with existing transport (e.g., reordering). Our extensive evaluation shows TLT mitigates timeouts in conventional transport and delivers low tail flow completion times for bursty foreground traffic, without causing any significant adverse impact on larger background flows. Finally, this paper does not raise any ethical concerns.

Acknowledgments

We would like to thank our sheperd Geoffrey M. Voelker and anonymous reviewers for their valuable feedback. This work was supported by Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Government of the Republic of Korea (Ministry of Science and ICT) [No.2018-0-00693].

Algorithm 1 TLT algorithm on TCP host

```
1: function ReceiveData(packet)
2:     if packet.mark = ImportantData then
3:         recvState ← Important
4:     else if packet.mark = ImportantClockData then
5:         recvState ← ImportantClock
6:     TcpForwardUp(packet)
7: function ReceiveAck(packet)
8:     remove ACKed (SACKed) segment from uimpQ
9:     if packet.mark = ImportantEcho then
10:        sendState ← Important
11:    else if packet.mark = ImportantClockEcho then
12:        sendState ← Important
13:        if packet.ack < SND.UNA then return
14:    TcpForwardUp(packet)
15:    if recvState ≠ Idle and sendBuffer is not empty then
16:        importantAckClocking()
17: function sendData(packet)
18:    if TLT detects loss and sendState = Important then
19:        packet ← pop 1 MSS of lost segment from uimpQ
20:        sendState ← Idle
21:        L3Transmit(packet)
22:        return
23:    if sendState = Important then
24:        packet.mark ← ImportantData
25:        sendState ← Idle
26:    else uimpQ.push(packet)
27:    L3Transmit(packet)
28: function sendAck(packet)
29:    if recvState = Important then
30:        packet.mark ← ImportantEcho
31:        recvState ← Idle
32:    else if recvState = ImportantClock then
33:        packet.mark ← ImportantClockEcho
34:        recvState ← Idle
35:    L3Transmit(packet)
36: function importantAckClocking
37:    if unimportant packet loss detected then
38:        packet ← pop 1 MSS from uimpQ
39:    else packet ← pop 1 Byte from uimpQ
40:    packet.mark ← ImportantClock
41:    sendData(packet)
```

Appendix A TCP + TLT

Some window-based transports (e.g., TCP) regard duplicated ACK as a congestion signal. However, `Important Data` packets on important ACK-clocking may trigger duplicated ACKs if earlier packets were not lost. This causes the underlying congestion control to misleadingly reduce its window. To prevent this, we mark the data packet on important ACK-clocking as `Important Clock Data`, and its ACK as `Important Clock Echo` (recognized as "important" at switches). If an `Important Clock Echo` packet whose acknowledge number is no larger than the last unacked sequence number arrives, the sender drops it at the TLT layer without passing it to the transport layer.

Appendix B TLT Deep Dive

Large-scale simulations in Appendix were conducted with 30% load with four 16 kB-sized foreground flows per host, except indicated otherwise.

How often are important packets dropped? Table 1 shows the loss rate of important packets (when PFC is not enabled). We measure this while changing the color-aware dropping threshold and the fraction of foreground traffic.

Work load	Load	DCTCP				TCP				DCQCN+SACK		DCQCN+IRN		HPCC+SACK	
		Baseline	+TLP	200us RTO_{min}	+TIRE	Baseline	+TLP	200us RTO_{min}	+TIRE	Baseline +PFC	+TIRE	Baseline	+TIRE	Baseline +PFC	+TIRE
Web Search	0.2	12.32	4.85	1.91	1.15	5.69	5.41	2.44	1.26	0.64	0.69	1.05	0.69	0.66	0.71
	0.3	8.48	5.30	2.51	1.56	12.89	5.95	3.81	1.99	0.65	0.71	1.05	0.70	0.67	0.71
	0.4	13.06	12.52	2.88	1.84	13.43	12.64	4.84	2.30	0.65	0.78	1.14	0.77	0.67	0.82
	0.5	9.79	8.64	3.12	2.06	14.81	13.00	5.79	2.41	0.66	0.89	1.08	0.87	0.68	0.94
Web Server	0.2	5.07	4.68	2.44	1.22	12.25	5.53	2.30	1.40	0.65	0.71	1.04	0.70	0.68	0.75
	0.3	12.79	5.42	2.73	1.77	13.12	12.98	5.14	2.30	0.66	0.74	1.05	0.71	0.68	0.76
	0.4	5.83	5.88	3.66	2.12	12.98	10.83	6.86	2.69	0.66	0.76	1.07	0.73	0.69	0.78
	0.5	5.79	5.70	4.57	2.38	12.92	12.88	7.78	5.56	0.67	0.80	1.08	0.77	0.69	0.80
Cache Follower	0.2	12.28	12.32	1.84	1.20	6.04	5.26	2.42	1.30	0.64	0.69	1.05	0.69	0.67	0.71
	0.3	8.82	5.35	2.63	1.57	13.67	9.67	4.41	2.08	0.65	0.70	1.05	0.69	0.67	0.72
	0.4	12.78	12.47	3.03	1.87	17.71	13.86	6.69	2.40	0.65	0.73	1.13	0.72	0.67	0.73
	0.5	9.38	8.93	2.99	1.99	19.90	14.87	8.37	2.59	0.66	0.80	1.08	0.78	0.68	0.75

Figure 15. 99.9%-ile foreground FCT for various workloads.

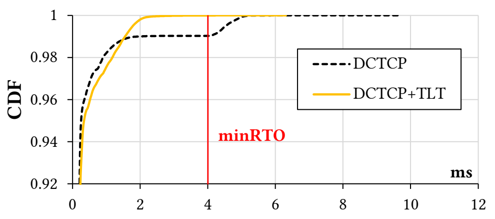


Figure 16. CDF of segment delivery time.

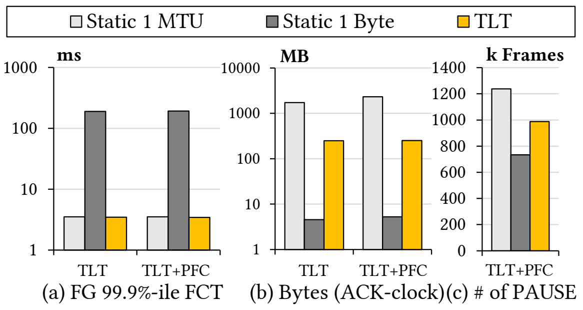


Figure 17. (a) Foreground tail FCT (b) Bytes (ACK-clocking) (c) # of PAUSE frames. Base transport is DCTCP.

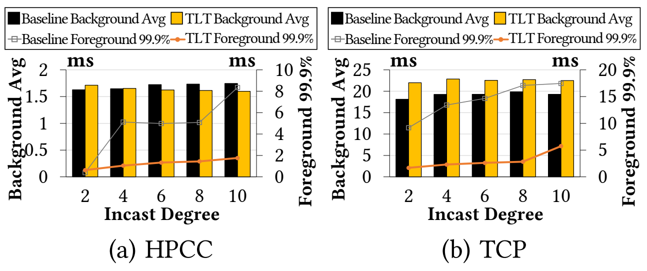


Figure 18. 99%-ile foreground FCT and average background FCT by different incast degree.

		Color-aware Dropping Threshold		
		400 kB	500 kB	600 kB
TLT +DCTCP	5% Foreground	0	0	0
	10% Foreground	0	1.02×10^{-5}	3.49×10^{-3}
TLT +TCP	5% Foreground	8.36×10^{-7}	3.4×10^{-5}	1.04×10^{-4}
	10% Foreground	9.29×10^{-5}	9.07×10^{-5}	3.24×10^{-4}

Table 1. Packet loss rate of important packets under various color-aware dropping threshold with (DC)TCP.

With 400 KB threshold, DCTCP exhibits no drops. TCP has a small loss rate, explaining the small timeouts we had in Figure 7 (a). With a larger threshold, less room is reserved for important packets. As a result, when the flow churn rate gets high (i.e., when the fraction of bursty foreground traffic increases), the loss rate for important packets increases.

Is adaptive important ACK-clocking effective? To minimize the adverse interaction with congestion control, TLT performs 1-byte (re)transmission when there's no usable window. It reverts to a full MTU retransmission when a loss is detected. To quantify its benefit, we compare with two alternative designs: one in which we always send 1 MTU, and the other that always sends 1 B for important ACK-clocking.

Figure 17 (a), (b), and (c) respectively show the 99.9%-ile foreground FCT, total bytes sent by important ACK-clocking, and the number of PFC PAUSE frames triggered. The baseline transport is DCTCP. The full MTU transmission delivers the best foreground tail FCT (Figure 17 (a)), but it incurs large bandwidth overhead (Figure (b)) and thus triggers 1.25x more frequent PFC PAUSE frames (Figure (c)). On the other hand, 1-Byte transmission makes recovery much slower (Figure (a)), but the overhead is low. TLT takes the benefit of both—it results in fast recovery (55 times shorter 99%-ile FCT than 1-Byte and similar to 1-MTU) while incurring 6.90 times smaller bandwidth overhead than a full MTU transmission.

Performance under diverse workloads. We show that TLT reduces 99.9%-ile foreground FCT up to 90.1% under various workloads. We use three other representative datacenter workloads as background traffic while keeping the same 8 kB incast as foreground traffic. Figure 15 shows the tail FCT for foreground flows for four transports. We use workloads from web search [17], web server [49], and cache follower [49]. For (DC)TCP and IRN, we observe that TLT gives the best performance for all workloads regardless of the load factor. For DCQCN and HPCC with SACK, TLT is not as good as PFC in the aspect of tail FCT of foreground flows. This is because DCQCN and HPCC can throttle background flows to maintain low queues, thus not generating PFC PAUSE frames to block foreground flows. Note TLT still reduces the average FCT of background flows by up to 54.7% (not shown in figure) compared to PFC.

Foreground traffic with different degrees of incast. We also demonstrate that TLT brings reduction in the 99%-ile foreground FCT regardless of the incast degree of foreground traffic. We use same configuration as Figure 5 and Figure 6, but with different incast degree for foreground flows. We vary the number of foreground flows generated from each host at a time, from 2 to 10. For both HPCC and TCP, TLT shows the best performance when the incast degree is high. TLT reduces the 99.9%-ile FCT of foreground flows up to 78.9% and 67.0% for HPCC and TCP respectively.

References

[1] 12.8 Tb/s StrataXGS Tomahawk 3 Ethernet Switch Series. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56980-series.

[2] Arista 7050QX-32. https://www.arista.com/assets/data/pdf/Datasheets/7050QX-32_Datasheet.pdf.

[3] Aurora 720. https://netbergtw.com/products/aurora-720/.

[4] Broadcom First to Deliver 64 Ports of 100GE with Tomahawk II 6.4Tbps Ethernet Switch. https://www.broadcom.com/news/product-releases/broadcom-first-to-deliver-64-ports-of-100ge-with-tomahawk-ii-ethernet-switch.

[5] High-Capacity StrataXGS® Trident II Ethernet Switch Series. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56850-series.

[6] High-Density 25/100 Gigabit Ethernet StrataXGS Tomahawk Ethernet Switch Series. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56960-series.

[7] HPCC simulation. https://github.com/alibaba-edu/High-Precision-Congestion-Control.

[8] IEEE DCB. 802.1Qbb - Priority-based Flow Control. http://www.ieee802.org/1/pages/802.1bb.html.

[9] Linux Kernel Timer Systems. https://elinux.org/Kernel_Timer_Systems.

[10] Mellanox ConnectX-3 Pro. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_ConnectX-3_Pro_Card_EN.pdf.

[11] Mellanox ConnectX-4 EN. https://www.mellanox.com/related-docs/prod_adapter_cards/PB_ConnectX-4_EN_Card.pdf.

[12] Mellanox Messaging Accelerator (VMA). https://github.com/Mellanox/libvma.

[13] Memcached. http://memcached.org/.

[14] ns-3 Network Simulator. https://www.nsnam.org.

[15] Redis. https://redis.io/.

[16] O. Aboul-Magd and S. Rabie. RFC 4115, "A Differentiated Service Two-Rate, Three-Color Marker with Efficient Handling of in-Profile Traffic". https://tools.ietf.org/html/rfc4115.

[17] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data center TCP (DCTCP). In *SIGCOMM 2010*.

[18] Mohammad Alizadeh, Adel Javanmard, and Balaji Prabhakar. 2011. Analysis of DCTCP: stability, convergence, and fairness. In *SIGMETRICS 2011*.

[19] M Allman, K Avrachenkov, U Ayesta, J Blanton, and P Hurtig. 2010. Early retransmit for TCP and stream control transmission protocol (SCTP). *Internet RFCs, ISSN 2070-1721, RFC* 5827 (2010).

[20] Guido Appenzeller, Isaac Keslassy, and Nick McKeown. 2004. Sizing Router Buffers. In *SIGCOMM 2004*.

[21] Behnaz Arzani, Selim Ciraci, Luiz Chamon, Yibo Zhu, Hongqiang (Harry) Liu, Jitu Padhye, Boon Thau Loo, and Geoff Outhred. 2018. 007: Democratically Finding the Cause of Packet Drops. In *NSDI 2018*.

[22] Behnaz Arzani, Selim Ciraci, Boon Thau Loo, Assaf Schuster, and Geoff Outhred. 2016. Taking the Blame Game out of Data Centers Operations with NetPoirot. In *SIGCOMM 2016*.

[23] Wei Bai, Kai Chen, Shuihai Hu, Kun Tan, and Yongqiang Xiong. 2017. Congestion Control for High-speed Extremely Shallow-buffered Datacenter Networks. In *APNet 2017*.

[24] Peng Cheng, Fengyuan Ren, Ran Shu, and Chuang Lin. 2014. Catch the Whole Lot in an Action: Rapid Precise Packet Loss Notification in Data Centers. In *NSDI 2014*.

[25] Inho Cho, Keon Jang, and Dongsu Han. 2017. Credit-Scheduled Delay-Bounded Congestion Control for Datacenters. In *SIGCOMM 2017*.

[26] Abhijit K. Choudhury and Ellen L. Hahne. 1998. Dynamic Queue Length Thresholds for Shared-memory Packet Switches. *IEEE/ACM Trans. Netw.* 6, 2 (April 1998), 130–140. https://doi.org/10.1109/90.664262

[27] Nandita Dukkipati, Neal Cardwell, Yuchung Cheng, and Matt Mathis. 2013. *Tail Loss Probe (TLP): An Algorithm for Fast Recovery of Tail Losses*. Internet-Draft draft-dukkipati-tcpm-tcp-loss-probe-01. IETF Secretariat. http://www.ietf.org/internet-drafts/draft-dukkipati-tcpm-tcp-loss-probe-01.txt http://www.ietf.org/internet-drafts/draft-dukkipati-tcpm-tcp-loss-probe-01.txt.

[28] Adam Dunkels. 2001. Design and Implementation of the lwIP TCP/IP Stack. *Swedish Institute of Computer Science* 2, 77 (2001).

[29] Sally Floyd and Van Jacobson. 1991. Traffic phase effects in packet-switched gateways. *ACM SIGCOMM Computer Communication Review* 21, 2 (1991), 26–42.

[30] Rohan Gandhi, Hongqiang Harry Liu, Y. Charlie Hu, Guohan Lu, Jitendra Padhye, Lihua Yuan, and Ming Zhang. 2014. Duet: Cloud Scale Load Balancing with Hardware and Software. In *SIGCOMM 2014*.

[31] Peter X. Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2015. pHost: Distributed Near-optimal Datacenter Transport over Commodity Network Fabric. In *CoNEXT 2015*.

[32] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. 2016. RDMA over Commodity Ethernet at Scale. In *SIGCOMM 2016*.

[33] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. 2015. Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis. In *SIGCOMM 2015*.

[34] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W Moore, Gianni Antichi, and Marcin Wójcik. 2017. Re-architecting datacenter networks and stacks for low latency and high performance. In *SIGCOMM 2017*.

[35] J. Heinanen and R. Guerin. RFC 2697, "A Single Rate Three Color Marker". https://tools.ietf.org/html/rfc2697.

[36] J. Heinanen and R. Guerin. RFC 2698, "A Two Rate Three Color Marker". https://tools.ietf.org/html/rfc2698.

[37] Shuihai Hu, Wei Bai, Baochen Qiao, Kai Chen, and Kun Tan. 2018. Augmenting Proactive Congestion Control with Aeolus. In *APNet 2018*.

[38] Shuihai Hu, Yibo Zhu, Peng Cheng, Chuanxiong Guo, Kun Tan, Jitendra Padhye, and Kai Chen. 2016. Deadlocks in Datacenter Networks: Why Do They Form, and How to Avoid Them. In *HotNets 2016*.

[39] Shuihai Hu, Yibo Zhu, Peng Cheng, Chuanxiong Guo, Kun Tan, Jitendra Padhye, and Kai Chen. 2017. Tagger: Practical PFC Deadlock Prevention in Data Center Networks. In *CoNEXT 2017*.

[40] Changhyun Lee, Chunjong Park, Keon Jang, Sue Moon, and Dongsu Han. 2015. Accurate Latency-based Congestion Feedback for Datacenters. In *ATC 2015*.

[41] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. 2019. HPCC: High Precision Congestion Control. In *SIGCOMM 2019*.

[42] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. 2015. TIMELY: RTT-based Congestion Control for the Datacenter. In *SIGCOMM 2015*.

[43] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. 2018. Revisiting Network Support for RDMA. In *SIGCOMM 2018*.

[44] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. 2014. Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities. In *SIGCOMM 2018*.

[45] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani.

2013. Scaling Memcache at Facebook. In *NSDI 2013*.

[46] Zhixiong Niu, Hong Xu, Dongsu Han, Peng Cheng, Yongqiang Xiong, Guo Chen, and Keith Winstein. 2017. Network Stack as a Service in the Cloud. In *HotNets 2017*.

[47] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Deverat Shah, and Hans Fugal. 2014. Fastpass: A Centralized "Zero-queue" Datacenter Network. In *SIGCOMM 2014*.

[48] Kun Qian, Wenxue Cheng, Tong Zhang, and Fengyuan Ren. 2019. Gentle Flow Control: Avoiding Deadlock in Lossless Networks. In *SIGCOMM 2019*.

[49] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. 2015. Inside the Social Network's (Datacenter) Network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (London, United Kingdom) *(SIGCOMM '15)*. Association for Computing Machinery, New York, NY, USA, 123–137. https://doi.org/10.1145/2785956.2787472

[50] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. 2016. Inside the Social Network's (Datacenter) Network. In *SIGCOMM 2015*.

[51] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, et al. 2015. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *SIGCOMM 2015*.

[52] Amin Vahdat, Behnam Montazeri, Christopher Alfeld, David J. Wetherall, Gautam Kumar, Hassan Wassel, Keon Jang, Kevin Springborn, Mike Ryan, Nandita Dukkipati, Xian Wu, and Yaogong Wang. 2020. Swift: Delay is Simple and Effective for Congestion Control in the Datacenter.

[53] Balajee Vamanan, Jahangir Hasan, and TN Vijaykumar. 2012. Deadline-aware datacenter tcp (d2tcp). In *SIGCOMM 2012*.

[54] Vijay Vasudevan et al. 2009. Safe and effective fine-grained TCP retransmissions for datacenter communication. In *SIGCOMM 2009*.

[55] Haitao Wu, Jiabo Ju, Guohan Lu, Chuanxiong Guo, Yongqiang Xiong, and Yongguang Zhang. 2012. Tuning ECN for data center networks. In *CoNEXT 2012*.

[56] David Zats, Anand Padmanabha Iyer, Ganesh Ananthanarayanan, Rachit Agarwal, Randy Katz, Ion Stoica, and Amin Vahdat. 2015. Fast-Lane: Making Short Flows Shorter with Agile Drop Notification. In *SOCC 2015*.

[57] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. 2017. High-resolution measurement of data center microbursts. In *IMC 2017*.

[58] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. 2015. Congestion Control for Large-Scale RDMA Deployments. In *SIGCOMM 2015*.

[59] Yibo Zhu, Monia Ghobadi, Vishal Misra, and Jitendra Padhye. 2016. ECN or Delay: Lessons Learnt from Analysis of DCQCN and TIMELY. In *CoNEXT 2016*.