Time-division TCP for Reconfigurable Data Center Networks

Shawn Shuoshuo Chen[0*], Weiyang Wang[1*], Christopher Canel[0]
Srinivasan Seshan[0], Alex C. Snoeren[2], Peter Steenkiste[0]
Carnegie Mellon University[0], MIT[1], UC San Diego[2]

ABSTRACT

Recent proposals for reconfigurable data center networks have shown that providing multiple time-varying paths can improve network capacity and lower physical latency. However, existing TCP variants are ill-suited to utilize available capacity because their congestion control cannot react quickly enough to drastic variations in bandwidth and latency.

We present Time-division TCP (TDTCP), a new TCP variant designed for reconfigurable data center networks. TDTCP recognizes that communication in these fabrics happens over a set of paths, each having its own physical characteristics and cross traffic. TDTCP multiplexes each connection across multiple independent congestion states—one for each distinct path—while managing connection-wide tasks in a shared fashion. It leverages network support to receive timely notification of path changes and promptly matches its local view to the current path. We implement TDTCP in the Linux kernel. Results on an emulated network show that TDTCP improves throughput over both traditional TCP variants, such as DCTCP and CUBIC, and multipath TCP by 24–41% without requiring significant in-network buffering to hide path variations.

CCS CONCEPTS

• **Networks** → **Transport protocols**; **Network protocol design**; **Data center networks**;

KEYWORDS

Data Center, Congestion Control, Transport Protocol

ACM Reference Format:
Shawn Shuoshuo Chen, Weiyang Wang, Christopher Canel, Srinivasan Seshan, Alex C. Snoeren, Peter Steenkiste. 2022. Time-division TCP for Reconfigurable Data Center Networks. In *ACM SIGCOMM 2022 Conference (SIGCOMM '22), August 22–26, 2022, Amsterdam, Netherlands.* ACM, Amsterdam, Netherlands, 17 pages. https://doi.org/10.1145/3544216.3544254

1 INTRODUCTION

Data center networks face the task of connecting a large number of end hosts over a high bandwidth, low latency fabric. Most data center networks are based on the popular Clos topology and provide the "big-switch" abstraction of a single full-bisection network. However, this approach is reaching its limit as the number of hosts increases and switching chip manufacturers contend with the slowdown of Moore's Law. Optical circuit switch (OCS)–based reconfigurable data center networks (RDCNs) are a promising solution to bridge the widening gap between bandwidth supply and demand. Compared to statically wired topologies [16, 24, 37, 38], RDCNs dynamically allocate the available network resources over time, providing greater scalability at lower cost and energy consumption. However, reconfigurable networks introduce unique challenges to existing transport layer protocols.

While TCP congestion control is designed to optimize data transfers between end points with competing traffic, it assumes that the channel between them is relatively stable in terms of bandwidth and latency. In the context of RDCNs, however, this assumption does not hold. In hybrid electrical/optical networks, the path between a pair of hosts may alternate rapidly between a fast optical network and a slower electrical network, with orders-of-magnitude differences in bandwidth and/or latency. Even in proposals that are purely optical [29], indirect forwarding—where packets hop across several hosts when a direct link is unavailable—creates drastic latency changes. Our experiments show that existing TCP variants are unable to take full advantage of the additional capacity provided by these RCDN technologies. Because TCP infers network state and updates its congestion model iteratively on the timescale of multiple RTTs, senders cannot converge to the current path conditions before the network reconfigures again in highly dynamic RCDNs.

To exploit the full potential of RDCNs, we present Time-division TCP (TDTCP), a new variant of TCP designed specifically for reconfigurable networks. We build on the insight that in RDCNs, as long as the network moves crisply between distinct configurations, it suffices to operate efficiently in each configuration separately; the full algorithm is then a piecewise function of individual models. In other words, in

[*]Equal contribution.

TDTCP, we multiplex independent network states over time, much like how MPTCP [41] multiplexes subflows over space. Unlike MPTCP, however, in TDTCP only one subflow is active at a given time. As the network moves from one configuration to the next, senders switch between network models.

While this overall approach may seem straightforward, we must solve several challenges before TDTCP is a complete protocol. First, in today's RDCNs, the duration in which a single optical configuration is in place is on the order of a few RTTs, leaving little time for the host to react to the new network environment. We address this issue by leveraging the fact that the top-of-rack (ToR) switches to which end hosts connect are directly involved in reconfiguring the network fabric. Therefore, we enlist ToR switches to directly notify their attached hosts when paths change. This signaling enables end hosts to view the reconfigurable network not as a single fabric, but rather as a series of time-division networks (TDNs) that transition periodically.

Second, when the network reconfigures, in-flight packets may not belong entirely to one TDN or the other. For example, data packets may traverse a different TDN than their ACKs, or a decrease in latency may reorder packets. Careful accounting is required to disambiguate transient reordering from true loss and credit the appropriate TDN, preventing unnecessary retransmissions. TDTCP tracks the sequence space associated with each TDN and uses heuristics and selective acknowledgments to avoid severe performance penalties during TDN transitions.

Finally, implementing TDTCP in a modern operating system kernel is challenging because the congestion control state is tightly integrated into the networking stack. We identify the subset of CUBIC congestion control state that must be duplicated across TDNs and optimize the TDN switching process to ensure end hosts are promptly notified of network transitions. We show that our open-source implementation in Linux kernel 5.8 scales to 100 Gbps and supports reconfigurations on microsecond timescales.

Using our prototype, we evaluate the performance of TDTCP against existing TCP variants in the kernel as well as MPTCP and reTCP [32], an RDCN-specific approach that requires extensive switch support. Results show that long-lived flows achieve better throughput under TDTCP: 24% higher than single-path CUBIC and DCTCP in one representative RDCN setting, and 41% higher than MPTCP. Additionally, TDTCP matches the throughput performance of reTCP while exhibiting lower switch buffer occupancy and does not rely on active switch buffer management.

This paper makes three contributions:

- Our measurement study on an emulated RDCN identifies TCP's shortcomings: single-path variants react too slowly to changing conditions, but multipath TCP is crippled by flow control stalls. We learn that a new TCP should leverage predictable reconfigurations to model TDNs separately and use a subflow scheme that bridges the gap between single and multipath TCP.
- We present the design of Time-division TCP, which maintains a separate congestion control state corresponding to each TDN but uses a unified sequence number space to (1) simplify loss recovery and bookkeeping across reconfigurations and (2) minimize the need for receiver-side changes.
- We present a full-featured implementation of TDTCP in Linux kernel 5.8, including performance optimizations that support aggressive reconfiguration frequencies and modern (e.g., 100-Gbps) link rates.

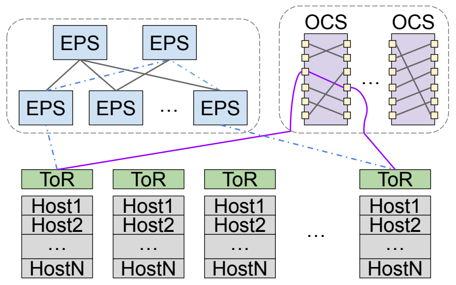


Figure 1: A hybrid, demand-oblivious RDCN topology. The OCS network provides temporary high-bandwidth links that supplement the EPS network.

[This work does not raise any ethical concerns.]

2 MOTIVATION

The underlying technology in optical networks—such as MEMS mirrors [9]—necessitates the use of communication circuits, so switches must periodically reconfigure to guarantee all-to-all connectivity at scale. Such designs are known as reconfigurable data center networks (RDCNs). In this section, we first give an overview of how RDCNs differ from electrical packet-switched networks and then explore why these differences yield poor TCP performance. We leave a full description of RDCN variants to §6.

2.1 Hybrid demand-oblivious RDCNs

In this paper, we consider hybrid, demand-oblivious RDCNs. As illustrated in Figure 1, such networks are *hybrids* because they consist of both electrical packet switches (EPSes) and optical circuit switches (OCSes). Each ToR is connected to several upstream EPSes as well as one or more OCSes. The EPS and OCS networks are isolated from each other.

Time-division multiplexing. When forwarding a packet, a ToR chooses either the EPS network or the OCS network. We assume that for a given destination, only one network is in use at a time: if a circuit exists, then the optical network is chosen, otherwise the ToR falls back to the packet network. During its lifetime, a flow may traverse either the packet network, the optical network, or both, depending on timing and flow duration. From a sender's perspective, this is a time-division multiplexed path, which is conceptually in-between a single path and true multipath routing. We refer to each discrete network state as a *time-division network* (TDN).

Different network characteristics. The packet and optical networks have different bandwidth and end-to-end latency due to their physical characteristics and network architectures. EPS networks typically use 10–400 Gbps links, while OCS links have the potential to be an order of magnitude greater than that [26, 30]. Traversing an EPS topology may take 100 μs with queuing, whereas the end-to-end OCS latency can be shorter because it includes fewer intermediate switches, does not queue inside the network, and only the destination switch decodes packets.

Stable characteristics within a TDN. Compared to the orders-of-magnitude differences across TDNs, within a TDN conditions are relatively stable: subject to background traffic, the bandwidth, latency, and loss rate on a path oscillate within a *comparatively* small range.

Predictable TDN pattern. OCSes follow a schedule that consists of several independent configurations, which we refer to as *days*, each of which individually provide connectivity to a disjoint subset of ToR pairs. The full schedule is known as a *week*. Between two days, there is a blackout period we call a *night* during which no packets can be sent while the optical network reconfigures. The term *demand-oblivious* refers to how the configurations are generated [27]: the complete schedule provides full-mesh connectivity over a week that can accept any traffic demand.

Frequent path switches. To keep overall utilization high, the optical network must maintain a high duty cycle, defined as the ratio of uptime to downtime. Related work [32] assumes a 9:1 duty cycle, so a state-of-the-art reconfiguration time of 20 μs [26] yields a circuit day of 180 μs, striking a balance between frequent reconfiguration (to serve all rack pairs often) and stability (to give flows time to converge). Assuming an RTT of $\approx$40 μs, a circuit day is on the order of five RTTs. This is a concern since TCP receives congestion feedback and adjusts its rate on the timescale of RTTs, so a short circuit day means that common TCP variants like CUBIC and DCTCP may not have enough time to adapt to path characteristics before switching paths again.

2.2 Measuring TCP performance in RDCNs

TCP is the dominant transport protocol in data centers, so it is important to understand TCP's performance on hybrid demand-oblivious RDCNs. In this section, we investigate the performance of two representative, widely-used TCP variants, single-path TCP CUBIC [19] and multipath TCP (MPTCP) [35, 41]. Our measurement and analysis addresses the following questions: *Can TCP take full advantage of RDCN capacity? Why or why not?*

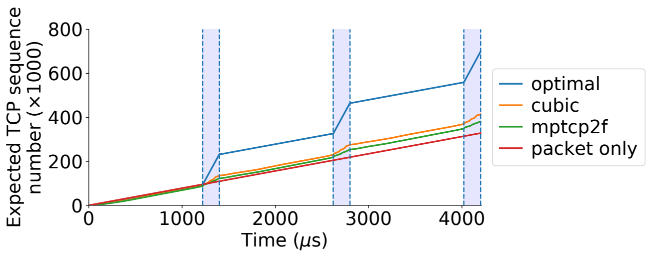


Figure 2: Sequence graph of TCP variants running in hybrid RDCN for 3 optical weeks.

We conduct all experiments using the Etalon RDCN emulator [32]. Specific testbed parameters are described in §5.1. We run CUBIC and MPTCP flows in the RDCN topology depicted in Figure 1. The open-source MPTCP implementation does not support our intended use: pinning one subflow (index 0) to the packet network and one subflow (index 1) to the optical network. Hence, we extended MPTCP with a prototype `tdm_schd` scheduler that steers packets between two subflows according to the RDCN schedule. For example, when the packet network is active, `tdm_schd` sends all packets to subflow 0, which is pinned to the packet network, and vice versa.

Figure 2 shows the throughput[1] of single-path TCP CUBIC and MPTCP in the form of a sequence-number graph. To remove startup and transient effects, results across thousands of optical weeks are averaged to create this visualization. The shaded regions show when the optical path is active while the unshaded regions correspond to the packet network. The slope of a sequence number curve gives the throughput. For example, for the *optimal* curve—which is computed analytically and assumes that a flow uses all available bandwidth—the slope is steeper in shaded regions than in unshaded ones, as one would expect due to the higher bandwidth of the optical network. On the other hand, the *packet only* curve specifies the best-case performance when the end host uses only the packet network. It has a constant slope equal to the rate of the packet network.

Both single-path TCP CUBIC and MPTCP fall far below the optimal line, showing that neither takes full advantage of the RDCN bandwidth. While their sequence number curves in the packet network (unshaded) are parallel to the optimal line, their curves in the optical network (shaded) are shallow and only slightly better than in the packet network. MPTCP's performance is lower than that of CUBIC, despite the fact that it has been made RDCN-aware, likely because of the

[1]See Appendix A.3 for an analysis of in-network queuing.

overhead of subflow management and the reinjection of lost segments that could not be ACKed before a path change.

Single-path TCP CUBIC has two shortcomings that contribute to its sub-optimal performance. First, it assumes a single path with conditions that change relatively slowly, but this assumption does not hold in RDCNs. After a path change, single-path TCP must spend multiple RTTs probing the path to converge to a new state that is appropriate for the new bandwidth and RTT, which can differ by orders of magnitude. Given the short duration of each optical day (≈5 RTTs in this example), there is insufficient time for single-path TCP to ramp up. Second, it is ignorant of the cyclic, deterministic RDCN schedule; instead, it must rediscover each change and adapt using mechanisms designed for traditional networks.

MPTCP's low throughput is caused by the fact that the connection is frequently stalled due to the strict isolation between the two subflows. Packets sent on one subflow may not be acknowledged for a while if the receiver is blocked from sending on that subflow because the corresponding TDN is not active. This means that the sender stalls until MPTCP's connection-level reinjection mechanism [36] attempts to retransmit the unacknowledged packets from one subflow on the other—currently active—subflow. This retransmission also introduces additional overhead.

In summary, our investigation reveals three findings: (1) TCP should model the characteristics of each TDN separately, (2) TCP should leverage the repeated, deterministic RDCN schedule to speed up congestion control convergence, (3) MPTCP-style independent subflows are not suitable for the RDCN environment where only one path is active at a time.

3 DESIGN

In light of the lessons learned above, there is certainly room for improvement. We hereby ask the question: how can we improve TCP performance in a network with fast and periodic changes in path characteristics? Taking a first-principles approach, we believe the answer is a new Time-division TCP (TDTCP) designed specifically for such use cases. Since these networks have multiple paths with different characteristics, we need a TCP variant that can handle all of these paths efficiently, but in an interleaved—as opposed to simultaneous—fashion. TDTCP takes advantage of the best of both worlds—leveraging building blocks and concepts from both single-path TCP and MPTCP. In this section, we explore our design decisions for TDTCP.

3.1 Per-TDN state variables

A time-division multiplexed path, with distinctive traits setting itself apart from a stable single path, should be treated like distinct paths. Particularly, TDTCP should model the path characteristics of different TDNs independently. Traditional single-path TCP lacks the ability to distinguish its own operating states in different TDNs. Upon a path change, traditional single-path TCP has no accumulated state information about the new TDN; all of its state is modeling the previous TDN. Instead of invalidating this model, it tries to reconverge with samples taken in the new TDN. For instance, using TCP's standard exponentially weighted moving average technique to merge bandwidth estimates for two different TDNs results in a value that is too large for the packet TDN and too small for the optical TDN. A natural fix is to maintain independent copies of internal state and dynamically swap them in when the relevant TDN is active. These sets are kept isolated so that samples from different TDNs do not pollute each other.

TCP models path state using many different internal variables. We group these into three categories according to their main functions and for brevity only list a few of the variables themselves. "Pipe" variables such as `pkts_out`, `lost_out`, and `retx_out` are used to estimate the number of bytes in flight, lost, and retransmitted, respectively. This gives TCP a sense of how full the "pipe" is. Congestion control variables such as `cwnd`, `ssthresh`, and `ca_state` are used to control how many more bytes are allowed into the network and whether the sender should slow down. Delay/RTT variables such as `srtt`, `rttvar`, and `mdev` are used to track the path latency. All three categories of path state variables require per-TDN copies to make TDTCP truly model each TDN separately.

In stable state, each per-TDN set of variables closely tracks its corresponding TDN. When the network path changes, TDTCP swiftly replaces the variables in use with another set tracking the new TDN. The new set of variables already contains a snapshot view of the new TDN when it was last active. After the replacement, TDTCP continues to operate on the new TDN as if it has just resumed from a checkpoint. In principle, variables of inactive TDNs should not be modified when that TDN is inactive to avoid unexpected state corruption. However, there are certain key exceptions that require updates to the state of inactive TDNs. These exceptions are to handle reordering scenarios and to make sure that when the inactive TDN becomes active again, it is modeled accurately. As a simple example, suppose that a packet is sent on TDN 0 and its ACK returns on TDN 1. TDTCP needs to decrease the in-flight count for TDN 0 even though TDN 1 is active. This ensures that when TDN 0 becomes active again, TDTCP has accurate knowledge about the in-flight packets. In addition to the above state variables, TDTCP also tags each packet and ACK with a TDN ID and keeps track of it throughout the lifetime of the packet. TDTCP can find out on which TDN a particular packet is sent by examining this field. This information enables some heuristics for managing cross-TDN reordering and spurious retransmissions. We discuss reordering scenarios in more detail in §3.4.

3.2 TDN change notification

By modeling path states separately, TDTCP implicitly requires the ability to promptly determine when network transitions occur. Instead of relying on in-band notifications like packet marks (e.g., ECN [12]) or iteratively probing the path, TDTCP relies on an out-of-band hint to decide when to use which set of states. Given that all switches in the RDCN know when the path change will happen and what the new path will be, it is reasonable to have the switches at the network edge, i.e., ToRs, propagate this information to the end hosts using proactive notifications. We employ a dedicated Internet Control Message Protocol (ICMP) [34] packet to carry this notification. We prefer this alternative to an in-band signal—such as piggybacking the notification in packet headers—because our out-of-band ICMP approach avoids potentially deferring the notification indefinitely in the absence of a continuous flow of packets. The ICMP packet simply carries an integer index indicating which network path is currently active. We refer to this path index as the TDN ID.

One major concern is whether TDTCP can obtain the most up-to-date TDN ID in a timely fashion, as this governs TDTCP’s operating efficiency. For example, if a packet-to-optical TDN change notification arrives at the sender when the optical day is about to end, then TDTCP will miss the opportunity to send at higher speeds. Vice versa, TDTCP could experience excessive packet drops if it continues to send at the optical bandwidth after the network has transitioned into a packet TDN. While the delivery latency differs network-by-network, our design of ToR-generated notifications should keep it generally low since ToRs and end hosts are located within the same rack. In §5.4, we evaluate several optimizations that further minimize this latency.

3.3 Sequence numbering

TDTCP maintains a single, connection-level sequence number space despite viewing each TDN independently. This decision contrasts with MPTCP’s two-level design for its subflows. Three reasons lead to this design decision: (1) flow control stalls that occur due to ACK delays after a switch, (2) it does not require inter-subflow coordination to facilitate loss recovery, and (3) only minimal receiver-side modifications are needed.

Before we discuss the benefits of TDTCP’s single sequence number design, let us summarize why MPTCP uses a separate sequence number space for each path. MPTCP benefits from this design in two ways. First, MPTCP assumes multiple paths that operate (if one ignores potentially shared bottlenecks) completely independently, thus using full-fledged TCP connections as subflows is a natural fit. Second, and perhaps more pragmatically MPTCP sought to keep each subflow compatible with various Internet middleboxes. Subflows with discontinuous sequence numbers may be incorrectly identified by middleboxes and blackholed or reordered. Fortunately, such universal compatibility is not a design requirement for TDTCP. Data center operators have significant control over network functions, and we assume they can make whatever changes may be necessary to support TDTCP, as they have for DCTCP and other recent TCP variants.

Avoiding flow control stalls after a TDN switch. As we observe in §2.2, MPTCP has poor performance after a TDN switch because of flow control. The reason is that ACKs for packets sent over the old TDN right before the switch may be stuck in the send queue of the receiver, since they must transmitted over the same TDN as the data packets they acknowledge. It is only after the packets have been reinjected over the active TDN that transmission can continue. With a single uniform sequence number space packets receive on one TDN can be ACKed on another TDN, and they naturally contribute to the overall sequence-space progress and drive transmission forward regardless of the currently active TDN.

Coordination. TDTCP’s single sequence number space eliminates the need for inter-subflow coordination. MPTCP requires a scheduler to serve as the central point of coordination. It makes a decision on which subflow to use when sending out a packet. In TDTCP, the active TDN is dictated by the network: there is only one TDN available at any point in time. If one TDN is slowed due to packet loss, it may block the progress during other TDNs due to the shared sender buffer. MPTCP employs connection-level reinjection to work around this situation by remapping lost segments onto another subflow [36]; TDTCP’s design avoids this unnecessary implementation complexity and performance overhead.

Receiver processing. MPTCP’s two-level sequence number space design requires runtime translation between data sequence number and subflow sequence number. This means the receiver also needs extensive modifications. However, with a single uniform sequence number space, no receiver-side modification is needed to support TDTCP sequencing. The major downside of TDTCP’s single sequence space and simple receiver behavior is cross-TDN reordering. We describe the impact of this phenomenon and how it can be mitigated below.

3.4 Packet reordering

Since the days of Reno, TCP has employed a fast-recovery mechanism to ameliorate the impact of packet loss. Specifically, segments deemed lost due to the receipt of three consecutive dupACKs are queued for immediate retransmission. This time-tested heuristic builds upon an assumption that no longer holds true in RDCNs: that a network path only slightly reorders packets, if at all. In fact, most reordering seen in RDCNs is cross-TDN reordering, where packets and their corresponding ACKs traverse TDNs with different

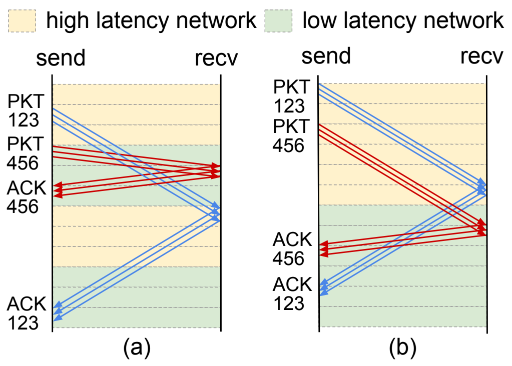


Figure 3: Cross-TDN reordering scenarios. Time proceeds from top to bottom; arrows represent TDTCP segments transmitted between sender and receiver.

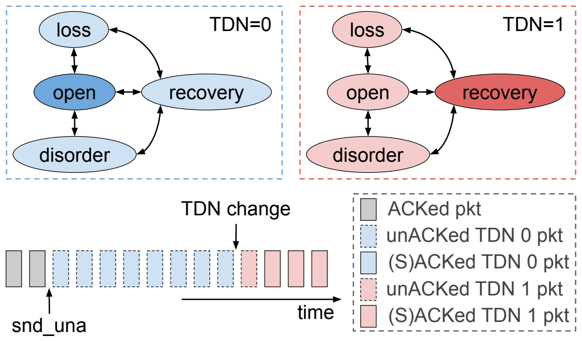


Figure 4: An example of relaxed reordering detection.

latencies. Cross-TDN reordering does not signal loss; ACKs are simply delayed due to increased path latency. If TDTCP were to continue following TCP Reno's fast-retransmit heuristics, it would spuriously retransmit a large number of packets at each TDN transition (indeed, over 100 in our setting according to measurements in §5.3).

Cross-TDN reordering. Occurrences of cross-TDN reordering can be grouped into two categories: data reordering and ACK reordering. Figure 3 shows examples[2] of each: scenario (a) corresponds to data reordering, and scenario (b) represents ACK reordering. Every horizontal slot in Figure 3 represents a unit of time t. One-way delays of the high-latency (yellow) network and low-latency (green) network are $5t$ and t, respectively. Consecutive slots of identical color represent the active duration of a TDN. For example, in scenario (a), the active TDN switches every $3t$. We label each data segment and its corresponding ACK with individual sequence numbers for clarity. (In practice TDTCP continues to use payload-byte-based sequence numbering and cumulative acknowledgments.)

Data reordering typically happens when the last few segments of a sending episode enter the network after the new TDN becomes active. In scenario (a), segments 4–6 are sent later but arrive before segments 1–3, causing reordering at the receiver. The receiver acknowledges segments 4–6 and 1–3 in the order they are received. The arrival of ACKs 4–6 before 1–3 would be interpreted by TCP Reno as a loss event, causing the sender to enter fast recovery mode and retransmit spuriously.

In scenario (b), the receiver sends out ACKs in the expected order. However, half of the ACKs are transported by a different TDN with lower latency. ACKs 4–6 end up arriving at the sender before ACKs 1–3. Unlike data reordering, ACK reordering is largely harmless because the latter ACKs carry a higher acknowledgment number than the former ACKs. Following TCP's cumulative ACK semantics, upon receiving ACKs 4–6, the sender assume segments 1–6 have all been received, which implicitly nullifies the in-flight ACKs 1–3. When ACKs 1–3 arrive, they are simply discarded.

[2]See Appendix A.1 for more reordering scenarios.

Relaxed detection. In order to avoid frequent spurious retransmission, TDTCP relaxes the triple dupACK heuristic with the help of per-TDN state and SACK support [11]. Whenever TDTCP detects a gap in the sequence number space, it first inspects the associated TDN IDs of all segments falling in the missing sequence range and compares them with a TDN change pointer. The TDN change pointer tracks the final sequence number of the previous TDN/the first sequence number of the new TDN. When packets in the "hole" have a different TDN ID than the ACK(s) that triggered the heuristics, the sender suspects cross-TDN reordering. As a result, TDTCP does not consider such segments to be lost and retransmits only those with a matching TDN ID (if any). TDTCP also updates the per-TDN congestion state machine associated with the lost segments; TDNs with retransmissions will enter the recovery state. For cases where lost segments with a different TDN ID are true tail losses (i.e., the last segments of a prior TDN really were dropped by the network), TDTCP relies on RACK-TLP [7] to recover efficiently.

Figure 4 illustrates TDTCP's relaxed reordering detection heuristic. When (pink) segments sent over TDN 1 are acknowledged before outstanding (blue) segments from TDN 0, SACK marks the successfully acknowledged segments and invokes fast recovery. All segments between the `snd_una` pointer and the highest SACKed sequence number (dashed rectangles) might be lost and potentially need to be retransmitted. However, TDTCP inspects the associated TDN IDs of these packets and compares them with the TDN change pointer. (Dashed blue) segments from TDN 0 are ignored since they belong to a different TDN and their ACKs are very likely just delayed. Only one (dashed pink) segment belonging to TDN 1 is confirmed as a true loss, which will be retransmitted. TDN 0 remains in `Open` state and is allowed to continue sending at full speed; TDN 1, on the other hand, enters `Recovery` state due to the loss.

3.5 Fairness, generality, and limitations

Fairness. TDTCP does not propose a new congestion control algorithm (CCA). Rather, it simply implements one of the available CCAs in each TDN. In principle, TDTCP could use multiple, different CCAs within a single flow. In our current implementation, however, CUBIC is used in all TDNs. We expect CCAs used within each TDN to have similar fairness properties as their single-path siblings, over a long period of time. Short-term anomalies could however exist. An in-depth investigation regarding fairness across TDNs and disparate CCAs remains as future work.

Generality. Though hybrid, demand-oblivious RDCNs are the motivation for our work, TDTCP is designed to be generic so that it can service a variety of use cases. The fundamental assumption TDTCP makes is that the network fabric moves between a fixed set of internally-coherent network conditions, where each condition is likely to reappear during the lifetime of a connection. Satellite-based communications appear to be another good use case for TDTCP: Satellite signal coverage has a periodic strong-weak pattern as satellites orbit the earth. Satellite links are used if a strong signal can be detected. When the signal falls weak, fiber links between ground stations are often used as a backup [22]. At any time, only one link is selected. TDTCP is particularly suitable for a network with this pattern.

Limitations. The main limitation of TDTCP is its operating regime. TDTCP is only useful when the network conditions change with a certain frequency. Consider two extreme cases: (1) TDNs change every a few hours, (2) TDNs change every few nanoseconds. In case 1, a regular single-path TCP performs well because few connections will ever experience a change in TDN, and those that do can amortize any performance degradation over a very long period of time. In case 2, traditional TCP variants can perform near-optimally because nanosecond-level TDN changes will appear similar to per-packet load balancing. Without loss of generality, TDTCP is most suitable to operate in networks where the periods between TDN changes are 1–100× path RTT.

4 IMPLEMENTATION

We have implemented TDTCP in Linux 5.8 kernel with 11,798 lines of C code changes. The code is open source and available at `https://github.com/shuoshuc/TDTCP`. In this section, we describe a few implementation-specific considerations. More details can be found in the source code, accompanying comments and corresponding documentation.

4.1 Packet format

TDTCP defines three types of packets for its operation. An ICMP packet is used for path change notifications. A TCP handshake option is used to negotiate and establish TDTCP connections. Another TCP option is used to inform the peer of the TDN path taken by the packet.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     Type=7    |     Code=0    |           Checksum            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Active TDN ID |                  (Reserved)                   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

(a)

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---------------+---------------+-------+-------+---------------+
|      Kind     |    Length=4   |SubType|       |    # TDNs     |
+---------------+---------------+-------+-------+---------------+
```

(b)

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---------------+---------------+-------+-------+---------------+
|      Kind     |   Length=16   |SubType|  (reserved)     |B|D|A|
+---------------+---------------+-------+-------+---------------+
|  Data TDN ID  |   (reserved)  |  ACK TDN ID   |   (reserved)  |
+---------------+---------------+-------+-------+---------------+
```

(c)

Figure 5: TDTCP packet formats: (a) ICMP for TDN change notification, (b) TCP option for the `TD_CAPABLE` handshake, (c) TCP option of `TD_DATA_ACK` exchange.

As discussed in §3.2, TDTCP uses a switch-generated path change notification to learn about RDCN schedule changes. The active TDN ID field is carried as the first byte of the ICMP payload. We assume the number of distinct paths in an RDCN is no more than 256, hence allocating a single byte in the packet. Should the use case require a larger integer, we can easily allocate more bytes for it. Figure 5a shows the ICMP packet format of the notification.

4.2 Connection establishment

A TDTCP-speaking end host uses TCP header options inside of normal TCP connections. TDTCP exposes the same application-level API as traditional TCP so it is fully compatible with existing applications. TDTCP is also transparent to the application layer: it does not require multiple network interfaces nor require the applications to make any decisions regarding path selection.

To enable hosts to establish a TDTCP connection, TDTCP defines a new TCP option (Figure 5b) of subtype `TD_CAPABLE`, indicating that the end host supports TDTCP. The option also carries the number of TDNs observed by the end host. In our current implementation, the two ends must both agree on the number of TDNs. This is to align the TDN IDs used by both ends—a TDN ID must refer to the same network condition at both parties. On an established[3] TDTCP connection, all segments carry a TCP option with subtype `TD_DATA_ACK` (shown in Figure 5c) that contains the ID of the TDN on which the segment was sent. If the segment contains data, the data TDN ID field should be properly filled and the `D` bitflag set, likewise for an ACK segment. This option is used by the TDTCP state machines and heuristics to identify segment reordering.

[3]See Appendix A.2 for details on TDN management *during* the handshake.

Our current TDTCP implementation supports runtime RDCN schedule changes, but the new number of TDNs seen by both the sender and receiver must remain equal. TDTCP automatically initializes a new set of state variables upon being notified of a new TDN for the first time. We have also implemented the ability for a host to downgrade established TDTCP connections to regular TCP. When doing so, only the local side is affected; the peer can continue to send TDTCP-enabled segments but only regular ACKs will be returned. We find this feature quite helpful for debugging.

4.3 TDN state management

Mapping per-TDN state variables from design to implementation is not entirely straightforward. Due to the way connection state is maintained in the Linux TCP stack, there are subtle semantic differences affecting how per-TDN state should be managed, requiring case-by-case inspection. At a high level, state variables fall into one of four different classes: *current TDN*, *all TDNs*, *any TDN* and *specific TDN*. We provide examples of each below.

Current TDN. The most common and natural semantic is for state to be accounted entirely within the current TDN. For example, when transmitting new data, each segment is tagged with the current TDN ID. Relevant state variables that track in-flight data such as `packets_out` should be incremented for the current TDN only.

All TDNs. On the other hand, sometimes the state of other TDNs must also be considered. Linux uses the variable `packets_out` in its `tcp_ack()` function to filter out ACKs that do not need to be processed. If `packets_out` is 0, the ACK must be either stale or malicious because no data is pending; hence, no ACK is expected. In the context of TDTCP, `packets_out` must be 0 across all TDNs because an ACK can acknowledge data sent in any TDN. This means the sum of all per-TDN `packets_out` should be used to verify ACKs.

Any TDN. TDTCP retransmits lost segments at the earliest opportunity, regardless of which TDN was used originally to transmit them. The Linux TCP stack maintains variables `ca_state` and `lost_out` to decide if a retransmission should be scheduled. If `ca_state` is Loss or Recovery and `lost_out` is non-zero, Linux prioritizes retransmitting the lost segment(s). In the context of TDTCP, retransmission needs to be scheduled if the above condition is true for *any* TDN, which means a logical OR should be applied to the two variables of all TDNs.

Specific TDN. In the receiving path, an incoming (cumulative) ACK can acknowledge data sent over multiple TDNs. When segments are acknowledged, their corresponding tracking variables such as `packets_out` and `retrans_out` must be decreased to reflect the progress. In the context of TDTCP, we identify the set of segments being acknowledged by scanning the retransmission queue and obtaining the TDN ID associated with each one of them. The variables `packets_out` and `retrans_out` are updated for the specific TDN associated with each.

4.4 RTT estimation

RTT estimation faces the same problem as in §3.4. Regular TCP assumes stable routing when collecting RTT measurements, i.e., that the samples always reflect properties of the same network condition. However, TDTCP may sample three distinct conditions even with only two TDNs. Let the one-way delay of TDN i and TDN j be denoted as $\frac{1}{2}RTT_i$ and $\frac{1}{2}RTT_j$, respectively. Type-1 samples measure RTT_i, type-2 samples measure RTT_j, and type-3 samples measure $\frac{1}{2}RTT_i + \frac{1}{2}RTT_j$. Type-3 samples arise when a segment and its ACK traverse different TDNs.

We seek to discard type-3 samples and match type-1 and type-2 samples to their corresponding TDNs. To achieve this goal, TDTCP makes use of the `TD_DATA_ACK` option and its local per-segment state tagging. TCP already uses the Karn-Partridge algorithm [23] to filter out retransmitted segments in RTT measurement; TDTCP further ignores samples with different data and ACK TDNs. The remaining samples are matched to the appropriate TDN based on information in the option.

One special consideration is the timeout value used by the retransmission timer. TDTCP only knows which TDN a segment is sent on, but cannot predict in general upon which TDN the corresponding ACK will return. In order to avoid premature timeouts, our current TDTCP implementation pessimistically assumes ACKs will return on the TDN with largest RTT. The timeout value of a segment sent on TDN n is based on a synthesized RTT of $\frac{1}{2}RTT_n + \frac{1}{2}RTT_{slowest}$.

5 EVALUATION

We evaluate TDTCP using our in-kernel Linux implementation on a small-scale testbed using Etalon [32], a recently published RDCN emulator. First, we compare TDTCP performance to leading data center TCP variants in an RDCN setting: §5.2 shows that TDTCP achieves high throughput and low switch-buffer occupancy for long-lived flows despite both bandwidth and latency differences. Then, we revisit design decisions on reordering and TDN change notification in §5.3 and §5.4 through microbenchmarks.

5.1 Etalon testbed

Etalon is a software RDCN emulator based on the Click modular router [25] and DPDK [13]. We configure Etalon to emulate a simple two-rack demand-oblivious hybrid RDCN topology, as illustrated in Figure 6.

Hardware. We use two physical servers to emulate two racks of hosts; Etalon itself runs on a third server. We fix the

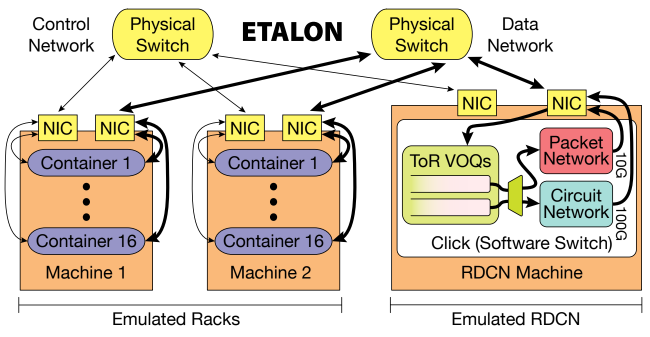


Figure 6: Our Etalon deployment. The emulated packet and circuit networks are 10 Gbps and 100 Gbps, respectively. Note Etalon only emulates the RCDN, the hosts are implemented as containers on physical machines.

source and destination of all connections in our experiments to use hosts attached to one of these two racks. We can emulate any scale of RDCN using this topology by pinning flows between this pair of racks. Each server runs 16 Docker containers [31] to emulate 16 distinct hosts residing in the rack. All servers have two 20-core Intel Xeon E5-2680 CPUs, 128 GB of memory, a 10 Gbps NIC for the control network, and a Mellanox ConnectX-3 40 Gbps NIC for the data network. We use `flowgrind` [42] to generate cross-rack flows over the data-plane network. Note that TDTCP is running in the actual Linux kernel networking stack, not through emulation or simulation; only the network itself is emulated.

RDCN configuration. In order to emulate a realistic target RDCN, we set the path-related parameters in Etalon according to §2.1. The number of TDNs is set to 2: TDN 0 corresponds to the EPS network and TDN 1 to the OCS network. The bottleneck bandwidth/average RTT of TDNs 0 and 1 are 10 Gbps/100 μs and 100 Gbps/40 μs, respectively. Etalon emulates two ToR virtual output queues (VOQs) for each rack—one for each traffic direction—which can be either static or reconfigurable at runtime. As a baseline configuration, we set the VOQ size to 16 packets (with jumbo frames [8] enabled). This is slightly larger than the packet network bandwidth-delay product, which has been shown to be sufficient for both TDNs [2]. One limitation of our testbed is the 40-Gbps hardware. In lieu of 100-Gbps network devices, we leverage time dilation [17] to "slow down" the end hosts. As a result, we are able to emulate a 100-Gbps network using only 5 Gbps of physical network capacity by dilating time by 20×.

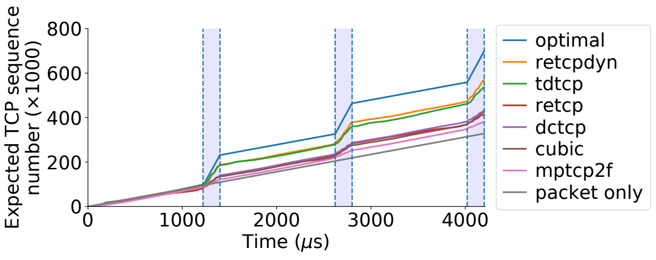
We measure long-lived cross-rack flow throughput for various TCP implementations as a function of TDN change frequency; all 16 hosts in one source rack transfer bulk data to their corresponding hosts in the destination rack. All flows are configured to start at the same time and last for 40 seconds. We deploy a pre-computed static schedule (in the fashion of RotorNet [30]) that consists of a variable number of 180-microsecond-long configurations. In every configuration, the racks are either connected directly to one another through the OCS (i.e., TDN 1) or to all other racks through the packet-switched network (TDN 0). Each configuration is separated out by a blackout (reconfiguration) period of 20 microseconds. Our experiments focus on a schedule with a 6:1 ratio of packet to optical TDNs, which naturally corresponds to an 8-rack hybrid RDCN where each rack is directly connected to the others in 1 out of every 7 OCS configurations. TDTCP has the most advantage over other TCP variants with ratios on this order. We leave it as future work to study TDTCP's performance when operating under extreme ratios.

We focus exclusively on long-lived flows because short-lived flows are unlikely to benefit from TDTCP. For example, RPC workloads that last a few RTTs likely only exist during one TDN. Even flows lasting tens of RTTs would terminate before exiting slow start in either TDN. In such cases, a larger initial cwnd would be more helpful than TDTCP. Overall, we do not expect TDTCP to impact on the completion time of short-lived flows but a full treatment is outside the scope of this paper.

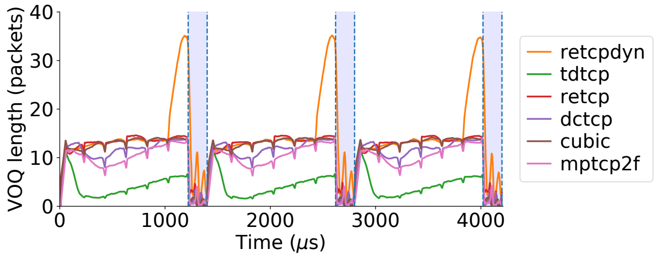
5.2 Bandwidth and latency difference

To understand the reasons behind TDTCP's superior performance we compare TDTCP with a wide variety of common TCP variants used in data centers: single-path TCP CUBIC and DCTCP [1], and MPTCP with two subflows, one for each network condition. In addition, we also compare to reTCP [32], a recently published approach to improving single-path TCP performance in RDCNs that requires active switch buffer management.

Figure 7 shows a side-by-side comparison of how each TCP variant performs when an RDCN oscillates between two conditions with differences in both bandwidth and latency. Specifically, the shaded area shows time periods when the source and destination racks are directly connected by the high-bandwidth/low-latency (optical) path; otherwise the connections are over the low-bandwidth/high-latency (electrical) path. Figure 7a shows the relative sequence number increase over time, which reflects the overall flow throughput. The "optimal" line plots the upper bound where an idealized TCP fully utilizes both conditions. The "packet only" line shows the same idealized TCP using the packet network only (i.e., it remains on the packet-switch fabric even with the optical path being available). The difference between these two lines represents the potential gain a TCP connection can capture in this RDCN by making use of the additional optical capacity. Note that the plots represent a ≈4-ms period during the experiment, not the absolute start (although both x and y axis values are normalized to the start of the plotted period). As a result, the performance of each variant is not necessarily equal

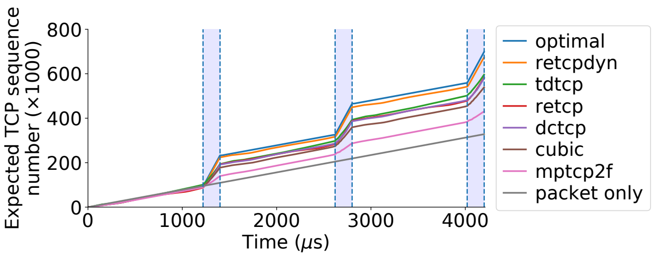


(a)

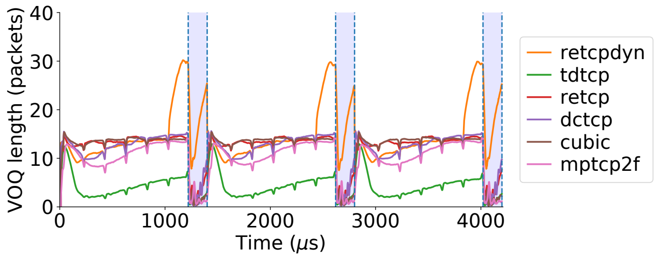


(b)

Figure 7: TCP throughput and switch buffer utilization under both bandwidth and latency difference: (a) sequence graph of each TCP variant, (b) ToR VOQ utilization over time (capped at 16 for all but "retcpdyn").



(a)



(b)

Figure 8: TCP throughput and switch buffer utilization with only bandwidth difference: (a) sequence graph of each TCP variant, (b) ToR VOQ utilization over time (capped at 16 for all but "retcpdyn").

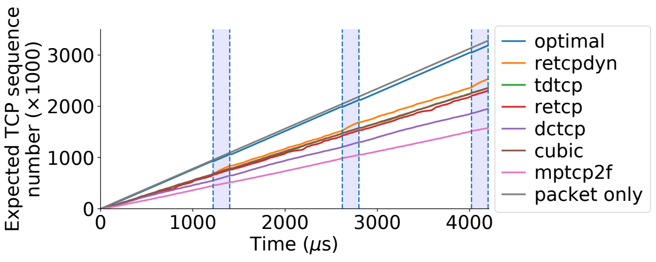


Figure 9: TCP throughput with only latency difference at 100 Gbps bandwidth. (TDTCP and CUBIC perform almost identically.)

even in the first TDN due to the state (e.g., send window and RTT estimates) accumulated earlier in the trace (c.f. Figure 9).

TDTCP dramatically out-performs CUBIC, DCTCP, and MPTCP, as none of the traditional variants are able to effectively make use of the substantial additional capacity made available by the optical link. The only competitive alternative is reTCP, but it requires sophisticated dynamic switch buffer resizing [32] in order to be effective. The "retcpdyn" line corresponds to a setting where the ToR enlarges its VOQ size to 50 packets at 150 microseconds ahead of the TDN change, and notifies reTCP to ramp up its congestion window. Thus, reTCP is able to pre-fill the VOQ and starts bursting at high bandwidth immediately after the TDN switch. In contrast, reTCP *without* dynamic buffer resizing ("retcp") sees a similar throughput to other single-path TCP variants. TDTCP, on the other hand, does not require any buffer resizing at runtime. In fact, Figure 7b shows that TDTCP's VOQ utilization is the lowest among all of the TCP variants we consider.

One pattern worth noting in TDTCP's VOQ curve is the spike when the RDCN transitions from high bandwidth/low latency to low bandwidth/high latency. We refer to the spike as the initial burst. The initial burst occurs because TDTCP is switching to a wide-open congestion window with near-zero inflight for the new active TDN. During the previous (optical) TDN, no new segments were arriving at the ToR's packet queue while virtually all in-flight packets have been drained. This process opens up the packet TDN's congestion window, hence the sender transmits a cwnd-sized burst upon receiving the TDN change notification. Techniques such as sender pacing can help prevent the potential switch buffer overflow.

To differentiate the impact of bandwidth difference versus latency difference on flow throughput, we repeat the same experiments but configure Etalon with only bandwidth difference or only latency difference at a time. Figure 8 shows the results for an RDCN with only bandwidth differences. Figure 8a suggests both CUBIC and DCTCP are able to adapt to bandwidth variation alone; they only slightly under-perform TDTCP in this setting (although MPTCP still struggles). Interestingly, the reTCP results suggest that dynamic buffer resizing, in the absence of latency variation,

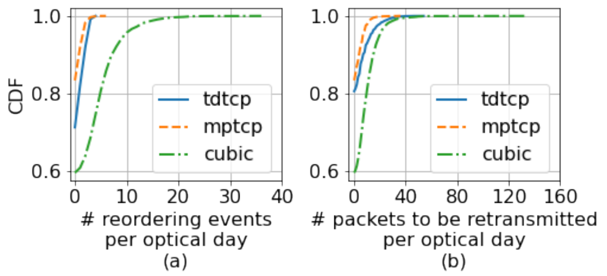


Figure 10: The (a) number of reordering events per day, (b) number of packets to be retransmitted per day. Note the y-axis does not start from 0.

can achieve near-optimal throughput. Without it, however, reTCP falls back in line with the other variants. Figure 8b shows that VOQ occupancy is largely unchanged in this scenario, with TDTCP still the clear winner.

Next, we consider a setting where the two TDNs have equal bandwidth but differing latency, such as in a multi-hop Opera-style [29] RCDN. Figure 9 illustrates the throughput[4] when link bandwidth is 100 Gbps. Because both TDNs have identical capacity, the optimal line and packet-only line almost overlap; the packet-only performance is slightly higher because flows in packet-only mode do not experience the reconfiguration blackout period. All of the buffer-filling TCP variants see similar throughput while DCTCP—a latency-sensitive variant—performs worse than its peers (with MPTCP again bringing up the rear). We hypothesize that the TDTCP approach could allow even latency-sensitive congestion control algorithms to perform well in such RDCN setting, but leave this topic to future work.

5.3 Reordering

At first glance, CUBIC's performance under latency variation is surprising, as TDTCP includes numerous heuristics (§3.4) to avoid packet reordering that one would expect to result from transitioning from a high-latency (packet) TDN to one with much lower latency (optical). To understand this behavior, we plot the number of reordering events in each optical day and the total number of packets impacted by this reordering. In one reordering event, multiple packets could be affected if their sequence numbers all fall into the gap between cumulative ACK sequence number and SACK sequence number. The total number of affected packets is the number of segments that will be resent as spurious retransmissions (if the congestion window permits). The measured reordering includes both cross-TDN reordering and intra-TDN reordering, which is an intrinsic property of the TDN itself.

[4]See Appendix A.4 for an analysis of in-network queuing.

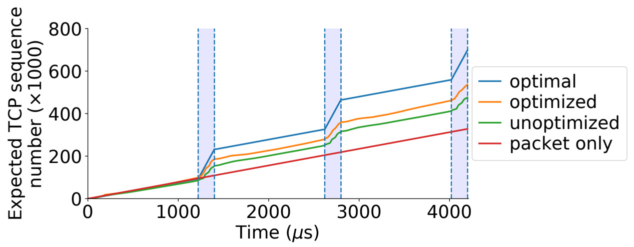


Figure 11: TDTCP performance with and without all optimizations on TDN change notification.

Figure 10 depicts the cumulative distributions of reordering events and retransmitted packets per optical day, for 3 representative TCP variants: CUBIC, MPTCP and TDTCP. CUBIC does, in fact experience significant reordering and spurious retransmission: 15 packets are retransmitted per TDN transition at the 90th percentile and 133 at maximum. TDTCP effectively cuts off the tail—only 7 packets are retransmitted at the 90th percentile with a maximum of 54. Moreover, 80% of the transitions to the optical TDN do not experience any reordering or retransmission at all. Hence, we conclude that CUBIC's impressive performance is a result of the sophisticated heuristics implemented in the Linux stack that are able to maintain an appropriate congestion window and high utilization despite significant reordering. Less finely-tuned stacks are unlikely to perform as well. We note that MPTCP maintains separate subflows for each of the two TDNs, so there should be no reordering due to TDN transitions. Hence, one can interpret the MPTCP line in these plots as a measure of the baseline intra-TDN reordering during the experiments.

5.4 TDN change notification

Following discussions in §3.2, we analyze some TDTCP traces and identify the component breakdown of end-to-end notification delivery latency. Barring the physical propagation delay, top contributing components are ToR-side packet generation and host-side notification processing. We list 3 concrete optimizations below that prove effective in reducing the component latency. These optimizations combined improve TDTCP's throughput by 12.7%, as demonstrated by Figure 11.

First, ToRs can use caching to minimize delay: by constructing an empty ICMP packet in advance and keeping it cached in memory, ToRs can just fill in the TDN ID and immediately send the packet out when needed. Compared to constructing a new packet from scratch every time, caching reduces the delay in sending out the notification in our software switch by 8× at the 50th percentile and 2.7× at the 99th.

Second, our initial implementation follows a "push" model to distribute new TDN IDs: the kernel loops over all established TDTCP flows and updates their current TDN IDs one by one. This scales poorly and the unlucky flows which see the TDN update after others get less time to send. Instead, we change to a "pull" model where each TDTCP flow checks a global TDN variable protected by a read-write lock when they need to send or process incoming packets. The time to update all TDTCP flows is reduced by 3 orders of magnitude.

Finally, using a separate dedicated network to send and receive the ICMP packets can eliminate the extra queueing delay on a busy data plane interface with a lot of incoming/outgoing data packets. This optimization lowers the one-way delay between sending out an ICMP packet and completion of receiver processing by 5× at both the 50th and 99th percentiles.

6 RELATED WORK

Existing TCP variants rely on in-band signals such as packet loss [18, 19], delay [4, 5], and explicit switch feedback [1] to estimate a fair share of the path bandwidth. TDN changes interfere with these signals, leading to estimates that tend to average across TDNs rather than track individual TDNs. TCP variants for high-speed networks ramp up more aggressively [10, 19] and can recover more quickly from estimation errors, but do not address the root of the problem. Wireless networks, with highly-variable bandwidth and frequent packet drops, have spurred research on modeling bandwidth fluctuations [40] and differentiating between congestion loss and random loss [14, 28]. RDCNs, on the other hand, can provide certainty regarding the current network conditions, which should in theory enable better performance.

TCP over multiple networks. MPTCP [41] is similar in spirit to TDTCP in that it divides a connection into multiple subflows that traverse (potentially) different paths, such as simultaneous WiFi and cellular connections on a mobile device. While theoretical analysis shows MPTCP can perform optimally by leveraging the least-congested path, our experiments show it is not well equipped to switch rapidly between subflows when the subflows experience periods of disconnectivity as they do in RDCN.

One recent proposal, reTCP [32], focuses specifically on the RDCN environment and advocates for explicit support from ToR switches, which entice senders to ramp up their transmission rate before additional bandwidth becomes available by dynamically increasing the size of switch buffers in advance of circuit establishment. Furthermore, reTCP requires ToR switches to mark packets to inform senders when their flows traverse an optical network; senders then react by multiplicatively increasing their congestion window. TDTCP builds on the idea of explicit notification regarding network conditions but eschews such elaborate switch support. Moreover, TDTCP is general, supporting an arbitrary number of distinct TDNs with various properties, not just the bimodal fabric reTCP presumes. Even still, our experiments show that TDTCP's performance is competitive with reTCP in precisely the environments for which reTCP was designed without requiring its extensive changes to ToR switches.

RDCN technologies. Our evaluation focuses on one particular class of RDCN, but there are many others in the literature. In hybrid RDCNs [9, 20, 27, 39], the OCS network is an accelerator for an existing packet network, and thus uses the packet network to hide the OCSes' limited connectivity. Alternatively, OCS-only RDCNs [3, 6, 15, 21, 29, 30] do not include a separate packet network; instead, ToRs with no direct connectivity send traffic through transit ToRs or hold traffic until direct connectivity is restored. In either case, the number and frequency of TDN changes can vary. In demand-aware RDCNs [6, 9, 15, 20, 21, 26, 33], a controller collects real-time traffic demand information and calculates a schedule that serves the current demand. More recent demand-oblivious RDCNs [29, 30] cycle through a set of configurations that together provide all-to-all connectivity, avoiding runtime calculations by using a fixed schedule. TDTCP is applicable in either case; all that is required is that ToRs notify the senders of the upcoming TDN when they change configurations.

7 CONCLUSION

TDTCP is a new TCP variant designed to make efficient use of the full spectrum of recent RDCN proposals that provide time-division multiplexed network paths between communicating end hosts. TDTCP monitors path characteristics of different time-division networks using independent copies of TCP state variables, leverages switch-generated path change notifications to rapidly converge to the optimal congestion state, and relaxes TCP's fast-recovery heuristics to avoid spurious retransmissions when network conditions change. We have implemented TDTCP in Linux kernel 5.8 and evaluated its performance on a small-scale (emulated) RDCN testbed. Results show that TDTCP dramatically decreases queue occupancy within the network and outperforms traditional single-path TCP variants such as CUBIC and DCTCP for long-lived flows. TDTCP is competitive with reTCP, a recent proposal targeting a particular class of RDCNs, but does not require switches to dynamically adjust buffer sizes as reTCP does.

ACKNOWLEDGEMENTS

We thank Hugo Sadok, Anup Agarwal, Daehyeok Kim, Stewart Grant, George Porter, George Papen, Alex Forencich and the anonymous SIGCOMM reviewers for their useful feedback. This material is based upon work supported by the National Science Foundation under grants CNS-1122444, CNS-1911104, CNS-1956095, CNS-2008624, SHF-2107244, ASCENT-2023468, and CAREER-2144766; the Department of Energy under the ARPA-E ENLITENED program's LEED and PINE projects; DARPA under the FastNICs program; the Sloan fellowship; and gifts from Meta and Google.

REFERENCES

[1] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data Center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 Conference (SIGCOMM ’10)*. Association for Computing Machinery, New York, NY, USA, 63–74. https://doi.org/10.1145/1851182.1851192

[2] Guido Appenzeller, Isaac Keslassy, and Nick McKeown. 2004. Sizing Router Buffers. *SIGCOMM Comput. Commun. Rev.* 34, 4 (aug 2004), 281–292. https://doi.org/10.1145/1030194.1015499

[3] Hitesh Ballani, Paolo Costa, Raphael Behrendt, Daniel Cletheroe, Istvan Haller, Krzysztof Jozwik, Fotini Karinou, Sophie Lange, Kai Shi, Benn Thomsen, and Hugh Williams. 2020. Sirius: A Flat Datacenter Network with Nanosecond Optical Switching. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM ’20)*. Association for Computing Machinery, New York, NY, USA, 782–797. https://doi.org/10.1145/3387514.3406221

[4] Lawrence S. Brakmo, Sean W. O’Malley, and Larry L. Peterson. 1994. TCP Vegas: New Techniques for Congestion Detection and Avoidance. In *Proceedings of the Conference on Communications Architectures, Protocols and Applications (SIGCOMM ’94)*. Association for Computing Machinery, New York, NY, USA, 24–35. https://doi.org/10.1145/190314.190317

[5] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2016. BBR: Congestion-Based Congestion Control. *ACM Queue* 14, September-October (2016), 20 – 53. http://queue.acm.org/detail.cfm?id=3022184

[6] Kai Chen, Ankit Singla, Atul Singh, Kishore Ramachandran, Lei Xu, Yueping Zhang, Xitao Wen, and Yan Chen. 2014. OSA: An Optical Switching Architecture for Data Center Networks With Unprecedented Flexibility. *IEEE/ACM Transactions on Networking* 22, 2 (2014), 498–511. https://doi.org/10.1109/TNET.2013.2253120

[7] Yuchung Cheng, Neal Cardwell, Nandita Dukkipati, and Priyaranjan Jha. 2021. The RACK-TLP Loss Detection Algorithm for TCP. RFC 8985. (Feb. 2021). https://doi.org/10.17487/RFC8985

[8] Mike Duckett, Jerome Moisand, Tom Anschutz, Diamantis Kourkouzelis, and Peter Arberg. 2006. Accommodating a Maximum Transit Unit/Maximum Receive Unit (MTU/MRU) Greater Than 1492 in the Point-to-Point Protocol over Ethernet (PPPoE). RFC 4638. (Sept. 2006). https://doi.org/10.17487/RFC4638

[9] Nathan Farrington, George Porter, Sivasankar Radhakrishnan, Hamid Hajabdolali Bazzaz, Vikram Subramanya, Yeshaiahu Fainman, George Papen, and Amin Vahdat. 2010. Helios: A Hybrid Electrical/Optical Switch Architecture for Modular Data Centers. *SIGCOMM Comput. Commun. Rev.* 40, 4 (aug 2010), 339–350. https://doi.org/10.1145/1851275.1851223

[10] Sally Floyd. 2003. HighSpeed TCP for Large Congestion Windows. RFC 3649. (Dec. 2003). https://doi.org/10.17487/RFC3649

[11] Sally Floyd, Jamshid Mahdavi, Matt Mathis, and Dr. Allyn Romanow. 1996. TCP Selective Acknowledgment Options. RFC 2018. (Oct. 1996). https://doi.org/10.17487/RFC2018

[12] Sally Floyd, Dr. K. K. Ramakrishnan, and David L. Black. 2001. The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168. (Sept. 2001). https://doi.org/10.17487/RFC3168

[13] Linux Foundation. 2022. Data Plane Development Kit (DPDK). (2022). http://www.dpdk.org

[14] Cheng Peng Fu and S.C. Liew. 2003. TCP Veno: TCP enhancement for transmission over wireless access networks. *IEEE Journal on Selected Areas in Communications* 21, 2 (2003), 216–228. https://doi.org/10.1109/JSAC.2002.807336

[15] Monia Ghobadi, Ratul Mahajan, Amar Phanishayee, Nikhil Devanur, Janardhan Kulkarni, Gireeja Ranade, Pierre-Alexandre Blanche, Houman Rastegarfar, Madeleine Glick, and Daniel Kilper. 2016. ProjecToR: Agile Reconfigurable Data Center Interconnect. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM ’16)*. Association for Computing Machinery, New York, NY, USA, 216–229. https://doi.org/10.1145/2934872.2934911

[16] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. 2009. VL2: A Scalable and Flexible Data Center Network. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication (SIGCOMM ’09)*. Association for Computing Machinery, New York, NY, USA, 51–62. https://doi.org/10.1145/1592568.1592576

[17] Diwaker Gupta, Kenneth Yocum, Marvin McNett, Alex C. Snoeren, Amin Vahdat, and Geoffrey M. Voelker. 2006. To Infinity and Beyond: Time-Warped Network Emulation. In *3rd Symposium on Networked Systems Design & Implementation (NSDI 06)*. USENIX Association, San Jose, CA. https://www.usenix.org/conference/nsdi-06/infinity-and-beyond-time-warped-network-emulation

[18] Andrei Gurtov, Tom Henderson, Sally Floyd, and Yoshifumi Nishida. 2012. The NewReno Modification to TCP’s Fast Recovery Algorithm. RFC 6582. (April 2012). https://doi.org/10.17487/RFC6582

[19] Sangtae Ha, Injong Rhee, and Lisong Xu. 2008. CUBIC: A New TCP-Friendly High-Speed TCP Variant. *SIGOPS Oper. Syst. Rev.* 42, 5 (jul 2008), 64–74. https://doi.org/10.1145/1400097.1400105

[20] Daniel Halperin, Srikanth Kandula, Jitendra Padhye, Paramvir Bahl, and David Wetherall. 2011. Augmenting Data Center Networks with Multi-Gigabit Wireless Links. In *Proceedings of the ACM SIGCOMM 2011 Conference (SIGCOMM ’11)*. Association for Computing Machinery, New York, NY, USA, 38–49. https://doi.org/10.1145/2018436.2018442

[21] Navid Hamedazimi, Zafar Qazi, Himanshu Gupta, Vyas Sekar, Samir R. Das, Jon P. Longtin, Himanshu Shah, and Ashish Tanwer. 2014. FireFly: A Reconfigurable Wireless Data Center Fabric Using Free-Space Optics. In *Proceedings of the 2014 ACM Conference on SIGCOMM (SIGCOMM ’14)*. Association for Computing Machinery, New York, NY, USA, 319–330. https://doi.org/10.1145/2619239.2626328

[22] Yannick Hauri, Debopam Bhattacherjee, Manuel Grossmann, and Ankit Singla. 2020. "Internet from Space" without Inter-Satellite Links. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks (HotNets ’20)*. Association for Computing Machinery, New York, NY, USA, 205–211. https://doi.org/10.1145/3422604.3425938

[23] P. Karn and C. Partridge. 1987. Improving Round-Trip Time Estimates in Reliable Transport Protocols. In *Proceedings of the ACM Workshop on Frontiers in Computer Communications Technology (SIGCOMM ’87)*. Association for Computing Machinery, New York, NY, USA, 2–7. https://doi.org/10.1145/55482.55484

[24] John Kim, Wiliam J. Dally, Steve Scott, and Dennis Abts. 2008. Technology-Driven, Highly-Scalable Dragonfly Topology. *SIGARCH Comput. Archit. News* 36, 3 (jun 2008), 77–88. https://doi.org/10.1145/1394608.1382129

[25] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. 2000. The Click Modular Router. *ACM Trans. Comput. Syst.* 18, 3 (aug 2000), 263–297. https://doi.org/10.1145/354871.354874

[26] He Liu, Feng Lu, Alex Forencich, Rishi Kapoor, Malveeka Tewari, Geoffrey M. Voelker, George Papen, Alex C. Snoeren, and George Porter. 2014. Circuit Switching under the Radar with REACToR. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI’14)*. USENIX Association, USA, 1–15.

[27] He Liu, Matthew K. Mukerjee, Conglong Li, Nicolas Feltman, George Papen, Stefan Savage, Srinivasan Seshan, Geoffrey M. Voelker, David G. Andersen, Michael Kaminsky, George Porter, and Alex C. Snoeren. 2015. Scheduling Techniques for Hybrid Circuit/Packet Networks. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT ’15)*. Association

for Computing Machinery, New York, NY, USA, Article 41, 13 pages. https://doi.org/10.1145/2716281.2836126

[28] Saverio Mascolo, Claudio Casetti, Mario Gerla, M. Y. Sanadidi, and Ren Wang. 2001. TCP Westwood: Bandwidth Estimation for Enhanced Transport over Wireless Links. In *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking (MobiCom '01)*. Association for Computing Machinery, New York, NY, USA, 287–297. https://doi.org/10.1145/381677.381704

[29] William M. Mellette, Rajdeep Das, Yibo Guo, Rob McGuinness, Alex C. Snoeren, and George Porter. 2020. Expanding across time to deliver bandwidth efficiency and low latency. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 1–18. https://www.usenix.org/conference/nsdi20/presentation/mellette

[30] William M. Mellette, Rob McGuinness, Arjun Roy, Alex Forencich, George Papen, Alex C. Snoeren, and George Porter. 2017. RotorNet: A Scalable, Low-Complexity, Optical Datacenter Network. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA, 267–280. https://doi.org/10.1145/3098822.3098838

[31] Dirk Merkel. 2014. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux J.* 2014, 239, Article 2 (mar 2014).

[32] Matthew K. Mukerjee, Christopher Canel, Weiyang Wang, Daehyeok Kim, Srinivasan Seshan, and Alex C. Snoeren. 2020. Adapting TCP for Reconfigurable Datacenter Networks. In *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation (NSDI'20)*. USENIX Association, USA, 651–666.

[33] George Porter, Richard Strong, Nathan Farrington, Alex Forencich, Pang Chen-Sun, Tajana Rosing, Yeshaiahu Fainman, George Papen, and Amin Vahdat. 2013. Integrating Microsecond Circuit Switching into the Data Center. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM (SIGCOMM '13)*. Association for Computing Machinery, New York, NY, USA, 447–458. https://doi.org/10.1145/2486001.2486007

[34] Jon Postel. 1981. Internet Control Message Protocol. RFC 792. (Sept. 1981). https://doi.org/10.17487/RFC0792

[35] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. 2011. Improving Datacenter Performance and Robustness with Multipath TCP. In *Proceedings of the ACM SIGCOMM 2011 Conference (SIGCOMM '11)*. Association for Computing Machinery, New York, NY, USA, 266–277. https://doi.org/10.1145/2018436.2018467

[36] Costin Raiciu, Christoph Paasch, Sebastien Barre, Alan Ford, Michio Honda, Fabien Duchene, Olivier Bonaventure, and Mark Handley. 2012. How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. USENIX Association, San Jose, CA, 399–412. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/raiciu

[37] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. 2015. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. *SIGCOMM Comput. Commun. Rev.* 45, 4 (aug 2015), 183–197. https://doi.org/10.1145/2829988.2787508

[38] Ankit Singla, Chi-Yao Hong, Lucian Popa, and P. Brighten Godfrey. 2012. Jellyfish: Networking Data Centers Randomly. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. USENIX Association, San Jose, CA, 225–238. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/singla

[39] Guohui Wang, David G. Andersen, Michael Kaminsky, Konstantina Papagiannaki, T.S. Eugene Ng, Michael Kozuch, and Michael Ryan. 2010. C-Through: Part-Time Optics in Data Centers. In *Proceedings of the ACM SIGCOMM 2010 Conference (SIGCOMM '10)*. Association for Computing Machinery, New York, NY, USA, 327–338. https://doi.org/10.1145/1851182.1851222

[40] Keith Winstein, Anirudh Sivaraman, and Hari Balakrishnan. 2013. Stochastic Forecasts Achieve High Throughput and Low Delay over Cellular Networks. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX Association, Lombard, IL, 459–471. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/winstein

[41] Damon Wischik, Costin Raiciu, Adam Greenhalgh, and Mark Handley. 2011. Design, Implementation and Evaluation of Congestion Control for Multipath TCP. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI'11)*. USENIX Association, USA, 99–112.

[42] Alexander Zimmermann, Arnd Hannemann, and Tim Kosse. 2010. Flowgrind - A New Performance Measurement Tool. In *2010 IEEE Global Telecommunications Conference GLOBECOM 2010*. 1–6. https://doi.org/10.1109/GLOCOM.2010.5684167

A APPENDIX

Appendices are supporting material that has not been peer-reviewed.

A.1 Cross-TDN reordering scenarios

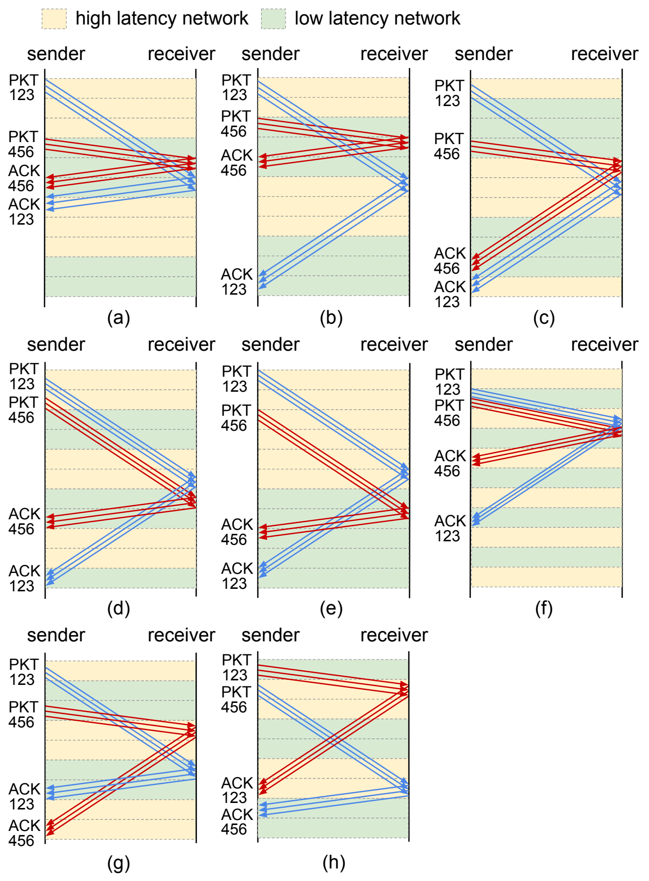


Figure 12: All types of cross-TDN reordering.

This section extends the cross-TDN packet reordering discussion in §3.4. It is worth pointing out that all types of cross-TDN reordering occur when the RDCN schedule transitions from a high latency network to a low latency network. There is no cross-TDN reordering in transitions from low latency to high latency because in that case packets will only arrive farther apart in the same order.

Figure 12 provides a comprehensive list of cross-TDN packet arriving scenarios. Scenarios (a)–(c) are data-crossing-only cases, where data packets arrive reordered at the receiver. Scenarios (d)–(f) are ACK-crossing-only cases, where ACK packets are out of order on the return path. And in scenarios (g)–(h), both data and ACKs are crossed. Within each category, scenarios differ by how far apart the packets are sent and the specific RDCN schedule. Scenarios (g) and (h) contain both data and ACK crossing, hence the ACKs return to the sender in the same order as the outgoing packets. This means that double crossing either cancels each other out or does not manifest as an issue from the sender's perspective. For instance, scenario (h) actually sees no reordering in either the outgoing or incoming paths.

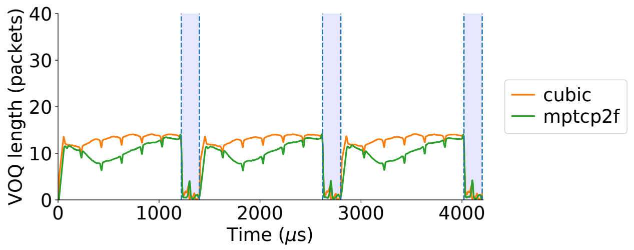
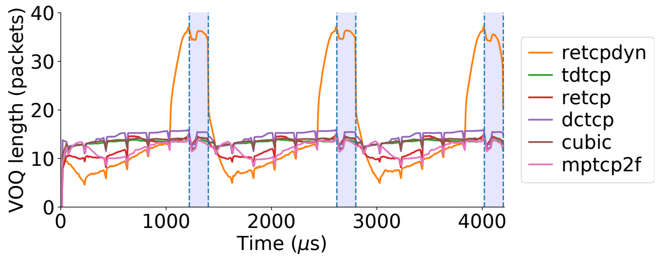
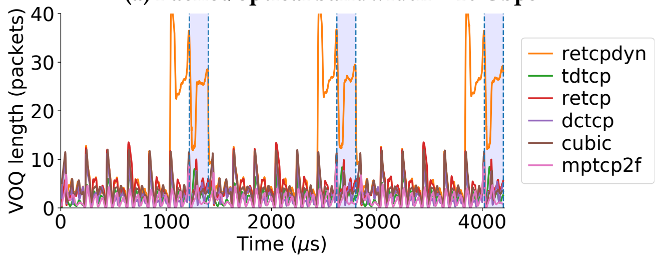


Figure 13: ToR VOQ utilization of single-path TCP CUBIC and MPTCP running in a hybrid RDCN for three optical weeks. All ToRs have a VOQ limit of 16 packets.



(a) Packet/optical bandwidth = 10 Gbps



(b) Packet/optical bandwidth = 100 Gbps

Figure 14: ToR VOQ utilization for an RDCN with only latency differences (max VOQ size is 16 packets for all cases except *retcpdyn*). Packet and optical RTTs are 20 μs and 10 μs, respectively. Bandwidth is fixed.

A.2 Special handling for SYN packet

The first SYN packet of a TCP connection is also counted as a segment. However, despite incrementing `packets_out` for this SYN, the Linux TCP stack does not keep a local copy of the segment like it does for other segments. Moreover, during the TCP handshake, there is not yet a notion of TDNs associated with the to-be-established connection, because the TDTCP negotiation has not completed. A problem hereby emerges: when the `SYN/ACK` packet is received by the sender, `packets_out` should be decremented to reflect the successful delivery of the `SYN` segment, but TDTCP does not know which `packets_out` was incremented or should be decremented. To mitigate this issue, TDTCP treats the first `SYN` packet as a special case and always tracks it under TDN `0`.

A.3 Queuing with bandwidth and latency changes

Continuing from §2.2, Figure 13 shows the virtual output queue (VOQ) utilization in the source ToR for the same configuration as Figure 2. The VOQ utilization of single-path TCP CUBIC is relatively stable during the first six days in a week (unshaded regions). When the optical network becomes active (shaded regions), packets in the VOQ are quickly drained, and the VOQ remains close to empty, showing that the service rate is much higher than the arrival rate. In other words, TCP has not ramped up to the new bottleneck rate.

The interesting dip observed in the *mptcp2f* curve is the result of *tdm_schd* switching subflows for transmission. When the 7th day ends, the VOQ drain rate drops from 100 Gbps to 10 Gbps while in-flight packets are still arriving at 100 Gbps. This results in a burst in the VOQ utilization. The sending host then switches to another subflow for the packet network, which transmits at a lower rate. Therefore, the VOQ is partially drained (causing the dip) before the sender ramps up again to fully occupy the VOQ.

A.4 Queuing with only latency changes

Continuing from §5.2, Figure 14 shows the ToR VOQ utilization for an RDCN configuration with bandwidth fixed to 100 Gbps or 10 Gbps, but with varying latency. In both cases, TDTCP's buffer utilization is in line with that of CUBIC, DCTCP, and MPTCP. reTCP, on the other hand, builds up large queues in advance of circuit start. This is to prepare enough packets so that it can start blasting at the circuit rate once the circuit TDN is activated. However, in these scenarios where bandwidth is fixed, the circuit BDP is *smaller* than the packet BDP because the circuit latency is lower, hence reTCP's queue-building technique is mismatched. TDTCP, on the other hand, achieves a high throughput without having to build large queues.

B ARTIFACT APPENDIX

Abstract

We provide an artifact package of Linux kernel source code, Etalon RDCN emulator source code, Wireshark source code, log parser and plotting scripts, as well as raw experiment dataset for this paper. The artifact package is fully open source and hosted online with public access. Specific hardware is required to set up a testbed for evaluating TDTCP. We welcome feedbacks and contributions to our artifacts.

Scope

The artifact package can be used to measure the performance of various TCP variants in an RDCN environment. Specifically, the kernel source code can be used to compile and install different versions of Linux kernels that implement multiple TCP variants. The Etalon source code can be used to emulate an RDCN given some specific hardware requirements. The Wireshark source code can be used to compile and install Wireshark with TDTCP protocol parser support. This is a debug tool when developing new features in TDTCP.

The artifact package allows users to reproduce the evaluation results and figures in the paper.

Contents

The artifact package contains the following artifacts:

- source code of Linux 4.19 MPTCP kernel
- source code of Linux 4.15 reTCP kernel
- source code of Linux 5.8 TDTCP kernel
- source code of Etalon RDCN emulator
- source code of Wireshark with TDTCP protocol parser support
- log parser and plotting scripts
- raw experiment dataset generated from evaluation

The source code of each kernel variant corresponds to what we used in the evaluation section. The source code of TDTCP kernel contains 2 branches: an optimized branch and an unoptimized branch. The source code of vanilla 4.15 kernel is not provided as it can be easily obtained from the Ubuntu 18.04 distribution. We also provide you our raw experiment dataset for cross reference.

Hosting

All source code in the artifact package is hosted on GitHub. You can find it at `https://github.com/shuoshuc/TDTCP`. Commit `1d84b62491e9db00847940029f24a2e43596e18c` of the main branch points to a stable snapshot.

The raw dataset in the artifact package is hosted on Zenodo. You can find it at `https://zenodo.org/record/6618182#.YsC54mDMJPY`. DOI `10.5281/zenodo.6618182` points to a stable snapshot.

Requirements

All experiments in the paper run on a testbed that emulates a reconfigurable data center network (RDCN) and hosts. 4 machines are required in total, each machine should have at least 32 CPU cores, 128GB RAM, 100GB disk space, 1x 10GbE NIC, 2x 40GbE NIC, and IPMI or other out-of-band means to connect to it. The 40GbE NICs must support DPDK. All four machines should reside in the same subnet, and the 10GbE and 40GbE interfaces should each connect to a separate common switch.

We recommend a spec that looks like:

- Intel(R) Xeon(R) CPU E5-2680 v2
- 128GB ECC RAM
- Mellanox ConnectX-3 dual port 40GbE NIC
- Intel 82599ES 10GbE NIC

What happens if using different hardware?

CPU. Since we run 16 containers on each machine to emulate a server rack, 2 CPU cores are pinned to each container for packet processing and other tasks to avoid process thrashing. With fewer than 32 cores, you might observe delay in packet processing and a reported performance different from the paper.

NIC. We require 2 DPDK-capable 40G NICs to emulate a SDN network with a data plane and a control plane. The Etalon RDCN emulator binds these 2 NICs to DPDK for userspace packet processing. The 10G NIC is used as a separate channel to send commands to and collect measurements from the hosts without interfering with the 40G network. With NICs of lower speed, you would not be able to emulate a fast data center network. Results may look different than expected. However, we expect that you should be able to at least verify the functionality of the artifacts and general trend of the results despite different hardware.

We base our experiment environment on Ubuntu 18.04 LTS and highly recommend that you do the same. This streamlines the setup process and avoids unexpected issues caused by incompatible software versions. Please make sure that you have Python installed and the binary points to Python 2.7. Also make sure that you have root or sudo permission on all machines.

Detailed instructions

Detailed instructions about how to configure and install kernels, start Etalon, run experiments and plot figures are documented in the README file at https://github.com/shuoshuc/TDTCP/blob/main/README.md.