Curbing Timeouts for TCP-Incast in Data Centers via A Cross-Layer Faster Recovery Mechanism

Ahmed M. Abdelmoniem*
CSE Dept., HKUST, Hong Kong
CS Dept., FCI, Assuit University, Egypt
Future Networks Theory Lab, Huawei, HK
amas@cse.ust.hk / ahmedcs@aun.edu.eg

Brahim Bensaou
Department of Computer Science and Engineering
The Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
brahim@cse.ust.hk

***Abstract*—We first study, at a microscopic level, the effects of various types of packet losses on TCP performance in a small data center. Then based on the findings we propose a simple recovery mechanism to combat the drawbacks of the long retransmission timeout. We emphasize through our empirical study that packet losses that occur at the tail of short-lived flows and/or bursty losses that span a large fraction of the congestion window are frequent in data center networks; and, in most cases, especially for short-lived flows, they result in a loss recovery that incurs waiting for a long retransmission timeout (RTO). The negative effect of frequent RTOs on the FCT is dramatic, yet recovery via RTO is merely a symptom of the pathological design of TCP's minimum RTO mechanism (set by default to the Internet scale). We propose the so-called Timely Retransmitted ACKs (T-RACKs), a very simple recovery mechanism for data centers, implemented as a shim layer between the virtual machines layer and the end-host NIC, to bridge the gap between TCP's huge RTO and the actual round trip times experienced in the data center. Compared to alternative solutions such as DCTCP, our T-RACKS has the virtue of not requiring any modification to TCP, which makes it readily deployable in virtualized multi-tenant public data centers. Experimental results show considerable improvements in the FCT distribution.**

***Index Terms*—Data Center, Cross Layer, Fast Recovery, Kernel Module, TCP-Incast, Timeouts.**

I. Introduction

The recent growth in data center deployments worldwide is reshaping the way the Internet and its application operate. New cloud-driven data intensive applications have emerged over the past few years to harness the cost-effectiveness and scalability afforded by cloud computing. Most such applications rely on distributed programming frameworks such as Hadoop, HDFS or Spark [1] for storage and processing of very large data sets. In such frameworks, master nodes often require data transfers from hundreds of worker nodes to build a complete (or partial) result. Due to the stringent timing requirements of interactive applications, a data transfer that misses a hard deadline, because of excessive waiting for packet loss recovery, returns a **partial** result (of lower quality). Hence, the quality of the application results is correlated not only with the average latency of traffic coming from a worker but also with the latency of the tail result (e.g., the 90th percentile of the flow completion times) which can be from 2 to 4 orders of magnitude worse than the median or even the average. In small scale private data centers, CPU resources are often the bottleneck and solutions that rely on task admission control and scheduling already exist (e.g., [2]). In contrast, public data centers are usually equipped with abundant computing resources but often adopt high over-subscription ratios in the network, making network latency the main bottleneck [3]. This is typical for many Internet-scale applications deployed on public (IaaS) clouds such as Microsoft Azure or Amazon EC2.

To circumvent such problem, large corporations such as Microsoft, Facebook and Google use dedicated well-structured data centers to deploy their time-sensitive applications. Nevertheless, in multi-tenant public data centers due to the predominance of many-to-one (or many-to-many) communication patterns and the variety of congestion control protocols in use, network congestion is inevitable and still results into long-tail waiting behavior. In addition, virtualization and the frequent context switching by the hypervisors to arbitrate resources among competing VMs contribute greatly to the inaccurate estimation of in-network delays by TCP in the VM, resulting in RTT estimates being bloated from the microsecond time-scale to the millisecond time-scale. Recent measurements [4–7] show excessive in-network packet losses during various congestion events.

TCP is by far the major transport protocol used by data center applications, however its design is still Internet-centric and is ill-suited for high-bandwidth low-latency environments like data centers. Proposed approaches for such environments [5, 8, 9] try to achieve in-network low delay via traditional methods by achieving a low queue occupancy; however these approaches fail to address the unfortunate cases when packet losses lead to timeouts, for instance, due to TCP-Incast. To understand fully the impact of timeouts, especially on short-lived flows, we first complement past works by conducting our own empirical study of the loss events in a small data center. To this end, we capture and trace microscopically TCP flows at the socket-level. Then by analyzing the collected data we study the frequency of occurrence of the two TCP loss recovery mechanisms (viz., RTO and 3-DUPACK) with respect to the flows size. We show notably that RTOs are often caused by tail-end losses or bursty losses and while they have

*This is work is done while Ahmed was Ph.D. student at HKUST, HK.

Manuscript is accepted for publication in proceedings of INFOCOM

negligible effects on the delay of long-lived flows, their effects are devastating on the performance of short-lived flows. To address this issue without modifying TCP[1], we propose a very simple mechanism to trigger a faster recovery for seemingly lost data long before the RTO expires. The contributions we make are three-fold:

1) We empirically study packet losses in depth as seen from the TCP socket level. The recovery mechanisms and their impact on the performance of TCP is highlighted.
2) We propose a light-weight cross-layer approach for a timely loss recovery before the RTO fire, without interfering with TCP in the guest VMs.
3) We evaluate the proposed scheme via large-scale ns2 simulations and a small-scale testbed implementation and experiments, showing up to 1 order of magnitude reduction in completion time of short-lived flows[2].

In the remainder, supported by an empirical study, we show the dramatic impact the RTO cause to the performance of time-sensitive flows in Section II. The proposed methodology and system design are presented in Section III. In Section IV, we discuss the packet-level simulation results in detail. Then, in Section V, we present the experimental results from the testbed deployment. We discuss important related work in Section VI. Finally, we conclude the paper in Section VII.

II. Problem and Motivation

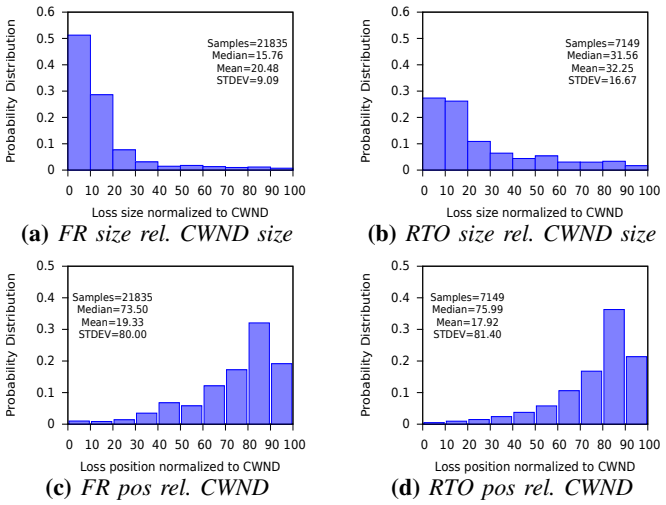
As the reader will see later, our proposed solution is simple and effective, however, it is not simplistic, as it derives from a deep understanding of TCP congestion and the mechanisms devised to address it. Therefore we will first motivate the approach and contrast it against existing alternatives before we move on to discuss the empirical study of TCP at the socket-level. Many schemes proposed in the literature deal with TCP congestion in data centers in a classic Internet-centric approach by invoking mechanism such as RED. The short-cut taken being "long delays and losses in the network can be curtailed by keeping the queue in the port buffer short via a RED-like mechanism". This short-cut unfortunately is fallacious for two major reasons: *i)* Data centers use high speed switches with small shallow buffers instead routers, therefore, the contribution of queueing delays to the total FCT is negligible regardless of the buffer occupancy. *ii)* Packet losses in TCP are not the pathology, they are merely the symptoms of congestion. The original designers of TCP, cleverly made packet losses an inevitable consequence of additive increase. They are meant to trigger a reaction with a conservative multiplicative decrease of the sending rate. For these reasons, many popular schemes designed specifically for data centers fail to actually address the problem of TCP incast congestion.

A number of measurement studies [4, 10, 11] have been conducted and shown that latencies in data centers environments vary greatly. To further understand the reasons of such issue, we here dive deep into the packet level analysis of the flows and the TCP socket variables at a microscopic level to understand TCP behavior and its loss recovery mechanisms.

Non traditional solutions were also proposed. An early work [12], based on data center measurements, found that the timeout mechanism is to blame for the long waiting times and proposed the very simple yet effective solution of reducing the minRTO value for TCP in data center environments, while using high resolution timers to keep track of delays at the microsecond-level. This approach actually solves the problem, reduces the FCT and mitigates TCP-incast congestion effects. However, *i)* it requires the modification of TCP, which makes it less appropriate for public multi-tenant data centers; and, *ii)* there is no magical minRTO value that fits all environments: for instance a minRTO that works inside the data center (e.g., between a web server and the backend database server) will definitely lead to spurious timeouts for Internet-facing connections (e.g., the connection between the web administrator workstation and the server in the data center).

A recent RFC [13] proposed the so-called tail loss probe (TLP) mechanism, which recommends sending TCP probe segments whenever ACKs do not arrive within a short Probe TimeOut (PTO)[3]. In addition to requiring changes to TCP, this approach suffers from two additional problems: *i)* probe packets also may be lost; and, *ii)* probe packets may worsen the in-network congestion, especially during TCP-incast.

A. Analyzing the Pathology and Symptoms



(a) *FR size rel. CWND size* **(b)** *RTO size rel. CWND size*

(c) *FR pos rel. CWND* **(d)** *RTO pos rel. CWND*

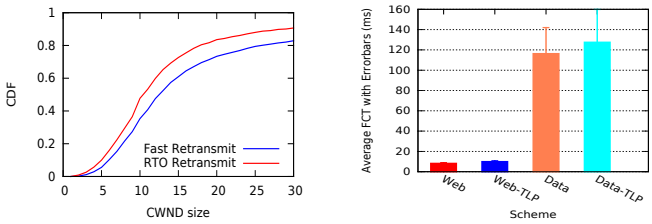
Figure 1: *(a-b) shows retransmission size while (c-d) position of the segment relative to CWND.*

To find out why packet losses do not seem to affect much elephant flows while degrading dramatically the performance of short-lived flows, we implemented a socket-level monitor module to collect TCP socket information. Then, we used our custom-built traffic generator to replicate Websearch workload of thousands of flows. Figs. 1a and 1b show the size of each retransmission (i.e., the seq# of the first and last segment

[1]Notice that in public data centers, under the IaaS model, the operating system and thus the protocol stack in the VM is under the full control of the tenant and cannot be modified by the cloud service provider.

[2]The implementation, simulation and experimental code and scripts are available at http://ahmedcs.github.io/T-RACKs.

[3]PTO is set to min(2*srtt, 10ms) if inflight>1 and to 1.5*srtt + worst case delayed ACK (i.e, 200ms) if inflight==1



(a) *CDF of CWND size* **(b)** *FCT of small flows*

Figure 2: *(a) shows the CDF of CWND at the time of the transmission of the lost packet. (b) TLP and NO-TLP FCT for Websearch and Datamining workloads*

in a single recovery) with respect to the congestion window $Cwnd$ for the fast retransmit recovery (FRR) and RTO-based recovery, respectively. While, Figs. 1c and 1d show the index[4] of the first retransmitted segment relative to $Cwnd$ when it was first transmitted (i.e., before loss is detected). Fig. 1b clearly shows that even though the RTO distribution is heavily tailed toward smaller fractions of the window, the distribution covers the whole range of fractions of the window. As, we can see that there are noticable RTO events covering large portions of the window (sometimes, the whole window). Fig. 1a suggests that FRR is also heavy tailed with much greater positive skewness towards small fractions of the window which allows for 3-dupACKs recovery. Fig. 1c points out that losses at the tail of the window occur with higher frequency, however in the case of FRR, the $Cwnd$ is relatively large enough to allow for FRR. Similarly, Fig. 1d clearly shows similar trends with higher frequency at the tail, however, in this case, $Cwnd$ is relatively small and hence, contains less in flight packets to allow for FRR. To put this into context, we see in Fig. 2a that typically $Cwnd$ for the flows with segments that experience RTO is smaller than that of those that recover via FRR.

Finally, we also show the ineffectiveness of the TLP mechanism [13] in recovering tail losses. Fig. 2b, shows that the TLP mechanism is not effective and due to its overhead it may even increase the FCT of short flows.

In data centers the pipeline size (i.e., Bandwidth-Delay Product) is small: typically with 100us RTT, a link of 1Gbps (respect. 10Gbps) can accommodate 8.3 packets (respect. 83 packets). In conjunction with shallow buffered switches, the nominal TCP fair share during TCP-incast barely exceeds one packet per flow and hence the occurrence of RTO is highly likely. This clearly highlights how TCP's performance can be degraded when operating in small windows regime in a small buffer with high-bandwidth low-delay switching environments like data centers. The effect on the flow completion time is more severe for short, time-sensitive flows, that normally last only a few RTTs, but that are compelled to wait for 2 to 4 orders of magnitude extra time due to the minRTO rule.

III. System Design

In this section, we will introduce T-RACKs whose design is based on the following observation: *packet losses are inevitable for the proper operation of TCP, the key to reducing the long latency and jitter is not to try to avoid losses but rather try to avoid long waiting after losses occur*. Its design is subject to the following requirements: (R1) To improve the FCT of latency-sensitive applications (mice flows). (R2) To be friendly to throughput-sensitive long-lived (elephant) flows (i.e., does not sensibly degrade their throughput). (R3) To be compatible with all existing TCP flavors (i.e., modifications must be in the hypervisors, which are fully under the control of the DCN operator, not in TCP itself); (R4) Last but not least, the mechanism must be simple enough to to be easily deployable in real data centers.

In this perspective, we design T-RACKs to actively infer packet losses by monitoring per-flow TCP ACK numbers and to proactively trigger the FRR mechanism of TCP to take action whenever and RTO is likely to take place. The goal is to help certain TCP flows recover from losses instead of waiting for TCP minRTO. The proposed intervention happens only when the loss is almost certain, leading to a significant improvement of recovery times, and hence the FCT of TCP. T-RACKs design derives from the following arguments: *i)* all TCP flavors adopt the FRR mechanism as a way to detect and recover from losses without incurring the expensive timeouts. If the one can force the mechanism into action, regardless of the nature of the loss, the resulting system would be transparent to the TCP protocol in the VM; *ii)* TCP relies on a small number of dupACKs to activate FRR, however in the majority of cases (especially for short-flows) there aren't enough packets in flight to trigger dupACKs. To achieve this, we propose to use "spoofed" TCP ACK signaling from the hypervisor to the VM. The hypervisor maintains a per-flow timer $\beta = \alpha * RTT + rand(RTT)$ to wait for the ACKs before it triggers FRR with spoofed dupACKs.

Note: After designing our mechanism we found out that our idea is reminiscent of the well known TCP SNOOP protocol [14] which retransmit lost segments on the behalf of the communicating end-points to filter bit-errors in low speed wireless networks. SNOOP also can can be applied in data centers, however, it is expensive to implement as it requires buffering all sent segments at the lower layers (e.g., link-layer) which requires a large buffer space in data centers.

A. T-RACKs Algorithm

T-RACKs algorithm 1 consists of three main functions: per-flow state maintenance on arrival and on departure and a timeout event handler. In the initialization (lines $1-6$), an in-memory flow cache pool is created to be used for new flow arrivals. This approach speeds up flow objects creation. To efficiently identify flow entries, a hash-based flow table is created and manipulated via the Read-Copy-Update (RCU) mechanism. Other parameters and variables are set in this step as well. Before each TCP segment departure, the program (lines $7-15$) performs the following actions: *i)* in (line 8), the packet is hashed using its 4-tuple and its corresponding flow is identified; *ii)* in (lines $9-15$), if a SYN arrives or the flow entry is inactive (i.e., a new flow), the flow entry is reset then TCP header info and options are extracted to activate a

[4]The index here points to the first retransmitted packet if a range of consecutive packets were lost. Each figure shows the aggregate of all servers

Algorithm 1: T-RACKs Packet Processing

```
1  Create flow cache pool;
2  Create flow table and reset flow information;
3  Initialize and insert NetFilter hooks;
   Input: α # of RTTs to wait before retransmitting ACKs
   Input: γ the threshold in bytes to stop tracking flows
   Input: φ the dupACK threshold used by TCP flows
   Input: t: the current local time counted in jiffies
4  Define x: the exponential backoff counter
5  β = α * RTT + rand(RTT);
6  Function Outgoing Packet Event Handler (Packet P)
7      f=Hash(P);
8      if SYN(P) or f.inactive then
9          Reset Flow (f);
10         Extract TCP options (i.e., TStamps, SACK, .., etc);
11         Update the flow information and flag entry as active;
12     if DATA(P) then
13         Update flow info (i.e., last seq#, .., etc);
14         f.active(t)=now();
15 Function Incoming Packet Event Handler (Packet P)
       /* For ACKs: extract and update flow
          information from incoming headers   */
16     if ACK_bit_set(P) then
17         f=Hash(P);
18         if f.elephant then return ;
19         Extract required values (e.g., seq#, ack#, .., etc);
20         if New ACK then
21             Update flow entry and state information;
22             Update the last seen new ACK from receiver;
23             Reset f.dupACK = 0;
24             Reset f.ACK(t) = now();
25             if f.lastackno ≥ γ then f.elephant = true ;
26         else
27             if Duplicate ACK then
28                 f.dupACK = f.dupACK + 1;
                   /* Drop extra dup-ACKs          */
29                 if f.resent > 0 then Drop Dup ACK ;
30         Update the TCP headers (e.g., TStamps & SACK);
```

Algorithm 2: T-RACKs Timeout Handler

```
1  Create and initialize a timer to trigger every 1 ms;
2  Function Timer Expiry Event Handler
3      for Flow (f) ∈ FlowTable do
4          if !f.Active or f.elephant then Continue ;
5          T = MAX(f.ACK(t), f.active(t));
6          if (t − T) ≥ β then
7              Resend last ACK (φ − f.dupACK) times;
8              Set f.resent(t) = now();
9              Set x = 2;
10             Continue;
11         if (t − f.resent(t)) ≥ (β ≪ x) then
12             resend ACK one more time;
13             x = x + 1;
14             Continue;
15         if (t − f.ACK(t)) ≥ TCPMinRTO then
16             stop RACK recovery;
17             soft reset flow (f) recovery state;
18             Continue;
19         if (t − f.active(t)) ≥ 1 then deactivate_flow(f) ;
```

new flow record; and *iii)* in (lines $13-14$), if a Data packet arrives, the $last\_sent$ seq# and time of the flow are updated.

Next, on each TCP ACK arrival, the code performs the following actions (lines $16-31$): *i)* in (line 17), the flow entry is identified using its 4-tuple; *ii)* in (lines $18-30$) if ACK sequence number acknowledges a new packet arrival, the $last\_seen$ ACK sequence and time is updated. The dupACK counter is reset. The flow is set as elephant if it exceeds a threshold γ; *iii)* in (lines $28-30$), if ACK number acknowledges an old packet (i.e., a duplicate ACK), drop dupACKs if the flow is in recovery mode, or increment the number of dupACKs seen otherwise; *iv)* in (lines 31), we update the TCP headers information of the ACK if necessary. We discuss this part in more detail later in sec III-C.

Algorithm 2 handles the global (per 1 ms) timer expiry events and performs the following actions for all **active non-elephant** flows in the table: *i)* in (lines $1-10$), if no new ACK acknowledging new data has arrived for β seconds since the last new ACK arrival, the flow times-out and then a spoofed ACK using the last successfully received ACK sequence is crafted. Then, T-RACKs sends it out to the sending process and/or VM residing on the same end-host. An exponential backoff mechanism is activated to account for various dupACK thresholds set by the sender TCP stack or OS. *ii)* In (lines $11-14$), if with the backed off timer the flow times out again and yet no new ACK has been received, another ACK is created and sent out to the corresponding sender. To ensure T-RACKs is not sending spurious spoofed dupAcks, the algorithm backs-off exponentially (as shown in line 11), after each retransmission of dupAck. *iii)* In (lines $15-18$), if the backoff time approaches the minRTO (i.e., 200ms), we stop triggering Fast-Retransmit (by resetting the soft state) and letting the sender TCP RTO handle the recovery of this segment. *iv)* In (line 19), if the inactivity period exceeds 1 sec, flow (f) entries are hard reset.

B. T-RACKs System Implementation

T-RACKs (i.e., Algorithm 1) relies on per-flow TCP header information of ACK packets to maintain per-flow TCP state information. We propose a light-weight end-host (hypervisor) shim-layer to implement T-RACKs[5]. The deployment of T-RACKs in data centers involves hashing the flows into a hash-based flow-table using the 4-tuples (i.e, SIP, DIP, Sport and Dport) whenever SYNs are signaled or a flow sends after a long silence period. The T-RACKs module uses the FlowTable to store and update TCP flow information (i.e., the last ack#, seq#, time, and so on) for each ongoing TCP flow. The module intercepts the outgoing ACKs and incoming Data to update the current state of each tracked (non-elephant) flow. Whenever packets are dropped and the receiver gets enough DATA to send enough dupACKs (real ones), the loss is recovered by FRR. In this case, the module does not intervene and the long

[5]T-RACKs can equally be implemented in the host NIC or in the switching chip of the ToR switches. This approach is feasible due to the relatively small number of flows at the end-host NIC or at the ToR level. This hardware extension is part of our future work.

RTO timeout is avoided. However, when the receiver fails to receive enough DATA to send dupACKs necessary to trigger FRR, then T-RACKs intervenes by sending spoofed dupACKs (or FRACKs) to the sender. Typically, the sender will trigger FRR to retransmit the lost segment within a reasonable time before the long TCP RTO is triggered.

C. Practical Aspects of T-RACKs System

T-RACKs System: is built upon a light-weight module at the hypervisor layer tracking a limited per-flow state, in the simplest case, it tracks TCP's identification 4-tuple, per-flow last ACK number and the time-stamp of the last non-dupACK. The system in spirit is similar to recent works in [15, 16] that aim to enable virtualized congestion control in the hypervisor or enforcing it via the vswitch without cooperation from the tenant VM. These approaches require fully-fledged TCP state information tracking and typically implement full TCP finite-state machines in the hypervisor. On the other hand, T-RACKs tries to minimize such overhead by tracking the minimal amount of necessary information and implementing only the retransmission mechanism.

T-RACKs Complexity: it resides in its interception of ACKs to update the last seen ACK information. However, since it does not perform any computation on the ACK packets[6], it does not add much to the load on the hosting server nor to the latency. This claim is supported by our observation in our experiments on our data center. A hash-based table is used to track flow entries of **active non-elephant** flows. In the worst case, when hashes collide, a linear search is necessary within the linked-list. However, this worst case is rare due to the small number of flows originating from a given end-host. Typically, end-host CPUs internally can sustain rates of 60+ Gbps of packet processing. Hence, the few processing required by the program would not affect the achieved TCP throughput.

Spurious retransmissions: T-RACKs may raise concerns related to the possibility of introducing spurious retransmissions and even making in-network congestion worse. This boils down to answering similar question when choosing the correct RTO value in TCP. For this purpose, we refer to a previous study [17], that essentially showed that even when a relatively bad RTT estimator is used, setting a relatively high minimum RTO can help avoid many spurious retransmission in WAN transfers. This fact is supported by a subsequent study [18] that shows significant changes (or variance) in internet delays. Recent works [19, 20] show similar behavior within current data centers. In our testbed, we observed noticeable variation in the measured RTT. The empirical data also shows a considerably large variation in smoothed RTT seen at the TCP sockets when measured at the time of first transmission of the packet and the time of fast retransmission or RTO retransmission, respectively. These variations can be mainly attributed to the beginning of some heavy background traffic, imbalance introduced by load balancing, or VM migrations, and so on. We note and agree with the aforementioned works that observed packet delays may not be mathematically nor stochastically steady. Hence T-RACKs ACK RTO (β) calculation shown in Algorithm 1 strikes a balance between rapid retransmission and the risk of causing spurious retransmission.

T-RACKs RTO β: in most of our experiments and simulations, we choose a value for ACK RTO (β) to be ($\geq$ 10) times the dominant measured RTT in the data center. We believe, and the results show, that this value achieves a good tradeoff between not having many spurious retransmission and at the same time not being too late in recovering from losses. We further adopt the well-know exponential back-off mechanism for subsequent RTO (β) calculations until either the loss is recovered or TCP's default RTO (i.e., minRTO) is close enough to timeout.

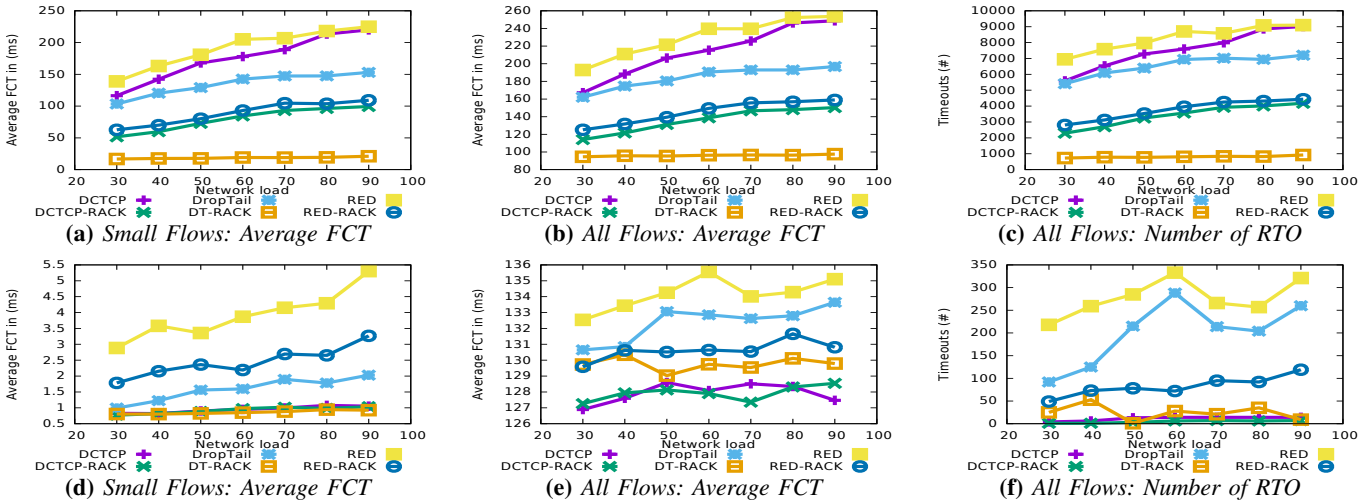
Synchronization of retransmissions: Since T-RACKs relies on a timer for ACK recovery, such timer may result in synchronization of retransmissions from different VMs on different hosts resulting into incast-like congestion. We studied the behavior in a simulation and the results show repeated losses due to possible synchronized retransmissions. A viable solution to de-synchronize such flows is to introduce some randomness in the RTO ultimately resulting into fewer flows experiencing repeated timeouts. We adopted this approach and added a random delay in the calculation of the RTO β.

TCP Header manipulation: TCP does not accept any packet with inconsistent timestamp, hence the timestamps are updated per ACK arrival with the local $jiffies$ variable to keep the consistency of timestamps whenever FRACKs are sent. For SACK enabled TCPs, fake SACK block information needs to be inserted for incoming ACKs (with no SACK blocks in TCP header) to indicate a small gap equal to the minimum segment size (i.e, 40 Bytes) after the last successfully acknowledged data.

Security Concerns: during FRR to be able to maintain its flight size and avoid timeout, TCP inflates the window artificially by 1MSS for each received dupAck. This can be exploited to launch ACK spoofing attack [21] on the senders. RFC 5681 released in 2009 addressed this particular attack and proposed implementing Nonce and Nonce-Reply as a way of verifying the source of dupACKs. However, such solution would require introduction of extra TCP headers prohibiting its deployment in real TCP implementations. In T-RACKs, we address such attack, by dropping dupACKS whenever the ACK timer expires when entering a recovery state. This approach is adopted to disable $Cwnd$ artificial inflation during recovery and at the same time prevents external ACK spoofing (other than FRACKs). Worth-mentioning also is that FRACKs are generated from the hypervisor layer which is under the control of the trusted datacenter operator.

TCP semantics: is conceptually violated since dupACKs should reflect packets following the lost one being received successfully. However, according to RFC 5681 [22], the network could replicate packets and hence the FRACKs could be treated as replicated packets from within the network.

[6]ACKs is updated in certain cases (e.g., to insert fake SACK block to signify a small gap in the SACKed numbers, otherwise FRACK is ignored)



(a) *Small Flows: Average FCT* **(b)** *All Flows: Average FCT* **(c)** *All Flows: Number of RTO*

(d) *Small Flows: Average FCT* **(e)** *All Flows: Average FCT* **(f)** *All Flows: Number of RTO*

Figure 3: *Performance metrics with various network load in range (30%, 90%) of workload 1 and 2, respectively*

IV. Simulation Analysis

In this section, we study the performance of T-RACKs to verify if it can achieve its goals in a large-scale simulation (we will investigate the implementation and deployment in a small data center later). To this end, we conducted a number of packet-level simulations using ns2 and compared T-RACKs performance against the state-of-the-art schemes. (For brevity, we refer to T-RACKs as RACK in the figures.)

A. Simulations in Data Center Topologies

Since any end-host based scheme is assumed to be scalable, to verify this, we experiment with T-RACKs in a larger scale-setup with varying workloads and flow size distributions. We use a spine-leaf topology with 9 leaves and 4 spines using link capacities of 10Gbps for end-hosts and over-subscription ratio of 5 (the typical ratio in current production data centers is in range of 3-20+). We examine scenarios that cover TCP-NewReno with DropTail, TCP-ECN with RED and DCTCP. We use a per-hop link delay of 50 μs, TCP RTO_{min} is set to the default 200 ms and the initial window of 10 MSS (similar to Linux implementation). Persistent connections are used for successive requests. The flow size distributions of the two workloads capture a wide range of flow sizes. The flows are generated randomly from any host to any other host with the arrivals following a Poisson process with various arrival rates (λ) to simulate various network loads. The inter-arrival times distribution is varied to mimic various network loads ranging from (30% to 90%). Finally, buffer sizes on all links are set to be equal to the bandwidth-delay product between end-points within one physical rack. We report the average FCT for small flows ([0-100KB]) and for all flows (medium [100KB - 10MB] and large [10MB+]) as well as the number of total timeouts in each case. We do not use T-RACKs elephant threshold, and T-RACKs RTO is set to 10 times the measured RTT.

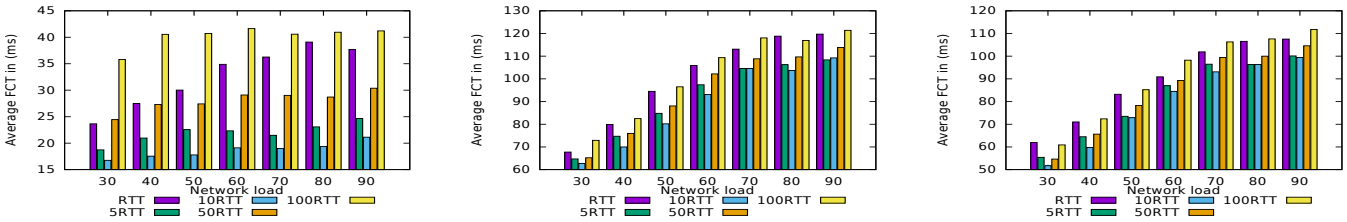
Figure 3 shows the average FCT for small and all flows as well as the total timeouts experienced by all flows in workloads 1 and 2, respectively. We note in both workloads that small flows take a great hit in FCT when they timeout regardless of congestion control and AQM in operation. This is where T-RACKs comes in help for small flows the most in improving their FCT by reducing the number of timeouts. We also note that overall FCT improves for all flows for two reasons: elephant threshold is disabled (i.e., all flows benefit from T-RACKs) and small flows finish quicker leaving network resources for larger ones. We notice that for workload 2 with almost 80% of flows less than 10KB, hence experiencing lesser timeouts overall, DCTCP can improve the FCT due to its ability to regulate the queue at small operating regime.

B. Sensitivity to Choice of T-RACKs RTO

We here repeat the last simulation by varying the value of the RTT multiplicative factor α in [1, 5, 10, 50, 100] to assess the sensitivity of our scheme to the RACK RTO. We report here the achieved FCT of small and all flows in each case for DropTail, RED and DCTCP. As shown in Fig. 4, the FCT is greatly affected by the choice of the parameter α. The lower values of α (i.e., 1 and 5) tend to cause spurious timeouts and exacerbate congestion in the network. On the other hand, excessively large values for α (i.e., 50 and 100) tend to be too conservative and result in TCP flows recovering later than they could. We can see a value of 10 achieves a good trade-off.

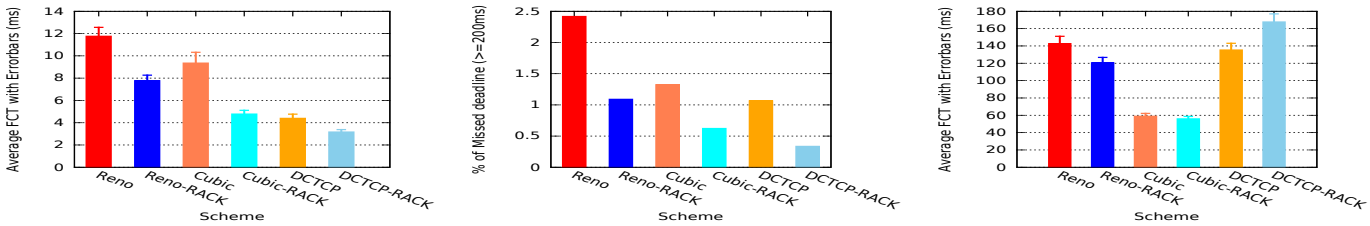
V. Linux Kernel Implementation

In this section, we discuss the implementation of T-RACKs as a loadable Linux kernel module then assess its performance using synthetic workloads found in production data centers [5, 10]. T-RACKs is a shim-layer residing between the VMs (or TCP/IP stack) and the hypervisor (or link-layer). We leverage the NetFilter framework [23] which is an integral part of Linux kernel. The NetFilter hooks attach to the data-path between the NIC driver and TCP/IP stack which imposes no modifications to the TCP/IP stack of the host OS nor guest OS, and being a loadable module, it allows for immediate deployment in production data centers. The module intercepts TCP packets incoming to the host or its guests before it is handed to the TCP/IP stack (i.e., at the post routing). First, the 4-tuples are hashed and the associated flow index is

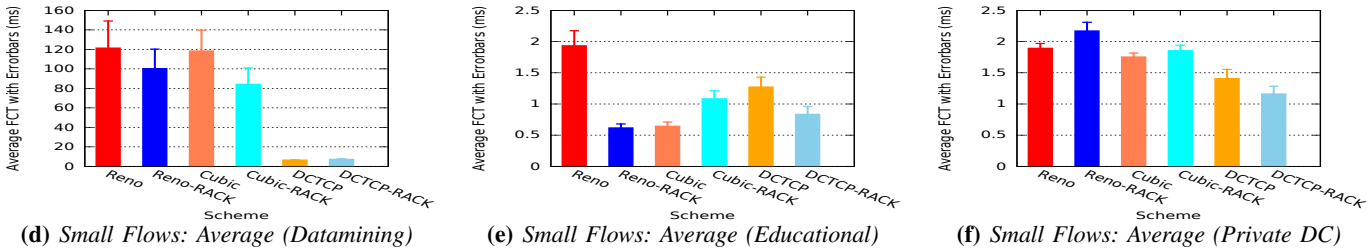


(a) *Small Flows: DropTail* **(b)** *Small Flows: RED-ECN* **(c)** *Small Flows: DCTCP*

Figure 4: *The same Websearch scenario as above but using DropTail, RED and DCTCP AQMs and α is varied from 1 - 100 RTTs.*



(a) *Small Flows: Average with Errorbar* **(b)** *Small Flows: Missed Deadlines* **(c)** *All Flows: Average with Errorbar*



(d) *Small Flows: Average (Datamining)* **(e)** *Small Flows: Average (Educational)* **(f)** *Small Flows: Average (Private DC)*

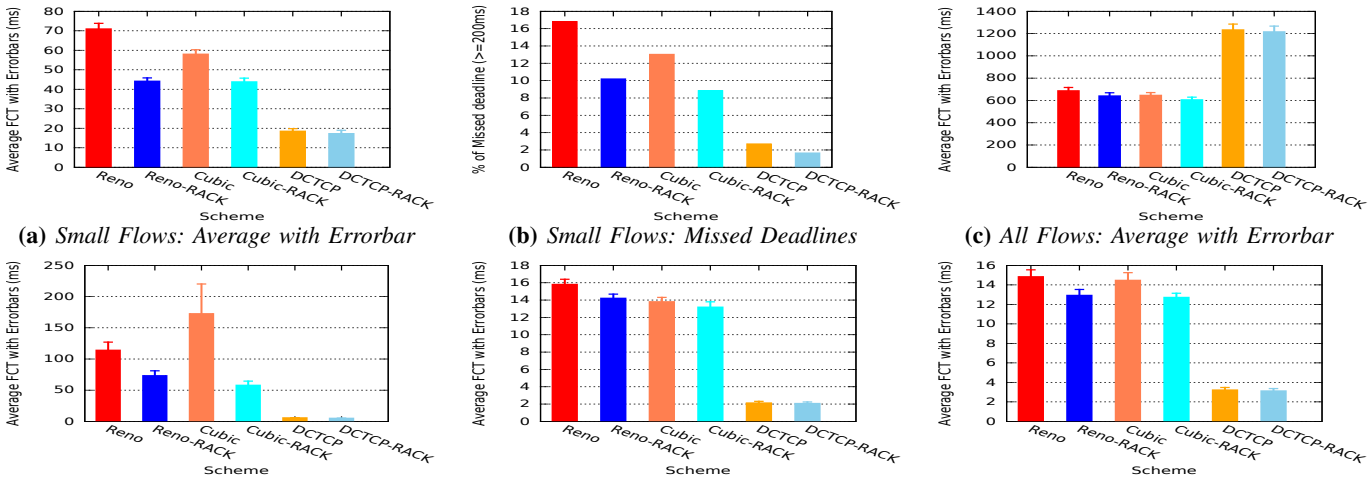
Figure 5: *Performance Metrics of one-to-all scenario without any background traffic.*

calculated via Jenkins hash (JHash) [24]. Then, TCP headers are examined and the right course of action is based on the flag bits (i.e., SYN-ACK, FIN or ACK) following the logic in Algorithm 1. Unlike SNOOP [14], the module does not employ any packet queues to store the incoming packets, it only stores and updates flow entry states (i.e., ACK No, arrival time and so on). Also, unlike [12] T-RACKs does not require fine-grained timers in microseconds time-scale, therefore the native $Jiffies$ timer is used (e.g., firing every 1ms). T-RACKs uses a single timer for all flows to handle per-flow RTO events. These design choices make T-RACKs lightweight and help reduce the server overhead.

We built a small-scale testbed consisting of 84 virtual servers each assigned a dedicated physical NIC (14 physical DC-grade servers equipped with 6 NICs each). The servers are interconnected via 4 non-blocking leaf switches and 1 spine switch. The testbed is organized into 4 racks (rack 1, 2, 3 and 4). The servers are connected to leaf switches and leaf switches are connected to the spine switch via 1 Gbps Ethernet links. The servers use Ubuntu Server 14.04 LTS with Linux kernel 3.18 which has integrated a full implementation of DCTCP. Unless otherwise mentioned, T-RACKs runs with the default settings (i.e., RTO of 4 ms and elephant threshold set to 100 KB). We use the traffic generator described in Section II, to run the experiments with realistic traffic workloads. In addition, we have installed the iperf program [25] to emulate long-lived background traffic (e.g., VM migrations, backups) in certain scenarios. We use different scenarios to reproduce one-to-all and all-to-all with/without background traffic. In one-to-all, randomly chosen clients in one rack sends random request to any of all the servers in the data center. While in all-to-all scenario, all clients in the data center send requests to randomly picked server out of all the servers in the data center. If background traffic is introduced, we run long-lived iperf flows in from all clients to all servers to evaluate T-RACKs under sudden and persistent network load spikes. We classify flows of size $<= 100KB$ small, $> 100KB \; and \; <= 10MB$ medium and $>= 10MB$ large.

A. Experimental Results and Discussion

One-to-all Scenario without Background Traffic: we report the performance of average FCT for small flows and all flows and the number of small flows that missed their deadlines. We set a hard deadline of 200ms for small flows however we do not terminate the flow even if it misses the deadline. The traffic generator is deployed on each single client running on an end-host in the data center and is set to randomly initiates 1000 requests to randomly picked servers on all other racks. Fig. 5a, Fig. 5b and Fig. 5c show the average FCT and missed deadlines for small flows and the average FCT for all flows, respectively, in the Websearch workload. While, Fig. 5d, Fig. 5e and Fig. 5f, show the average FCT for short flows in data mining, Educational, Private DC workloads, respectively. We make the following observations: *i)* for all workloads, T-RACKs helps small flows regardless of TCP flavor, on both the average and variation of FCTs. Compared to Reno, Cubic and DCTCP, T-RACKs reduces the average FCT of small flows by $\approx (34\%, 49\%, 19\%)$ for Websearch, $\approx (18\%, 29\%, -)$ for Datamining, $\approx (69\%, -, 35\%)$ for



(a) *Small Flows: Average with Errorbar* **(b)** *Small Flows: Missed Deadlines* **(c)** *All Flows: Average with Errorbar*

(d) *Small Flows: Average (Datamining)* **(e)** *Small Flows: Average (Educational)* **(f)** *Small Flows: Average (Private DC)*

Figure 6: *Performance metrics of one-to-all scenario with background traffic*

Educational and $\approx (-, -, 22\%)$ for Private DC workloads. We notice that DCTCP improves the FCT over its RENO and CUBIC counterparts and T-RACKs could improve their performance in terms of the missed deadlines in Websearch. The average FCT, in certain cases of Educational and Private DC workloads, has seen an increase of FCT with T-RACKs. In these workloads, the network load is quite light (as shown by the small FCT without T-RACKs) and hence the added overhead of deploying T-RACKs module surpasses its performance gains for these light workloads. *ii)* for Websearch workload, T-RACKs reduces the missed deadlines for short flows by $\approx (55\%, 53\%, 35\%)$ for RENO, Cubic, and DCTCP, respectively. *iii)* T-RACKs improves slightly the overall average FCT which is attributed to faster FCT of short flows who leave the network bandwidth for medium and large flows. The improvement was by $\approx (16\%, 5\%)$ for Reno and Cubic, respectively. However, for DCTCP case, the overall FCT slightly increases due to the setting of T-RACKs elephant threshold to only intervene for small flows.

One-to-all Scenario with Background Traffic: to put T-RACKs under a true stress, we run the same one-to-all scenario with an all-to-all background traffic. Fig. 6a, Fig. 6b and Fig. 6c show the average FCT and missed deadlines for small flows as well as average FCT for all flows in Websearch and Fig. 6d, Fig. 6e and Fig. 6f show the average FCT for short flows in data mining, Educational, Private DC workloads, respectively. We observe the following: *i)* T-RACKs can improves the average FCT of small flows for all workloads regardless of TCP congestion control in use. As shown compared to Reno, Cubic and DCTCP, T-RACKs reduces the average FCT of small flows by $\approx (38\%, 25\%, 7\%)$ for Websearch, $\approx (11\%, 5\%, 3\%)$ for Educational and $\approx (13\%, 13\%, 4\%)$ for Private DC workloads. The improvement increases for Datamining workload to $\approx (36\%, 67\%, 14\%)$ since it includes a wider range of short flows. *ii)* T-RACKs reduces the missed deadlines for short flows of Websearch by $\approx (40\%, 33\%, 39\%)$ for RENO, Cubic, and DCTCP, respectively. *iii)* T-RACKs still improves for the overall average FCT $\approx (7\%, 5\%, 2\%)$ for Reno and Cubic, and DCTCP respectively.

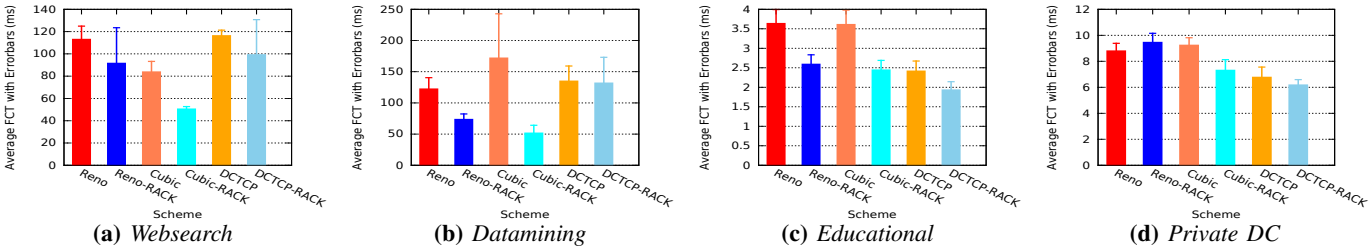
All-to-all Scenario without Background Traffic: we run the all-to-all scenario where all clients initiate 1000 requests to any of all the servers in the data center. Fig. 7a, Fig. 7b, Fig. 7c and Fig. 7d show the average FCT for short flows in Websearch, Datamining, Educational, Private workloads, respectively. The network load is higher given the more complex nature of this all-to-all traffic, yet, T-RACKs still can deliver significant improvements of up to 71% in the FCT for all workloads.

VI. Related Work

A number of research works have found, via measurements and analysis, that TCP timeouts are the root cause of most throughput and latency problems in data center networks [12, 26]. For example, [12] showed that frequent timeouts can harm the performance of latency-sensitive applications. Numerous solutions have been proposed. These fall into one of four key approaches. The first mitigates the consequence of long waiting times of RTO, by reducing the default MinRTO to the 100 μs - 2 ms [12]. While very effective, this approach affects the sending rates of TCP by forcing it to cut CWND to 1; it relies on a static MinRTO value which can be ineffective in heterogeneous networks; and it imposes modifications to TCP stack on tenant's VM.

The Second approach aims at controlling queue build up at the switches by either relying on ECN marks to limit the sending rate of the servers [5], or using receiver window based flow control [6] or deploying global traffic scheduling [8, 9]. These works achieved their goals and have shown they could improve FCT of short flows as well as achieving high link utilization. However, they require modifications of either the TCP stack, or introduce a completely new switch design, and are prone to fine tuning of various parameters or sometimes require application-side information.

The third approach is to enforce flow admission control to reduce TimeOut probability. [27] has proposed ARS, a cross-layer system that can dynamically adjust the number of active TCP flows by batching application requests based on



(a) *Websearch* **(b)** *Datamining* **(c)** *Educational* **(d)** *Private DC*

Figure 7: *The average FCT with errorbars of the small flows: Websearch, Datamining, Educational and Private DC in all-to-all scenario*

the sensed congestion state indicated by the transport layer. The last approach, which is adopted in this paper due to its simplicity, and feasibility, is to recover losses by means of fast retransmit rather than waiting for long timeout. TCP-PLATO [26] proposed changing TCP state-machine to tag specific packets using IP-DSCP bits which are preferentially queued at the switch to reduce their drop-probability enabling dupACKs to be received to trigger FRR instead of waiting for timeout. Even though TCP-PLATO is effective in reducing time-outs, its performance is degraded whenever tagged packets are lost, in addition, the tagging may interfere with the operations of middle-boxes or other schemes and most importantly it modifies the TCP state machine of sender and receiver.

VII. Conclusion and Future Work

In this paper, we studied packet losses and the impact of various recovery methods on the flow performance, and then proposed T-RACKs, an efficient cross-layer approach for timely recovery from losses. T-RACKs improves the FCT of time-sensitive flows and helps avoid throughput-collapse situations. T-RACKs is deployed either at the sender-side or the receiver-side as a shim-layer residing between TCP and the network. Simulation and experimental results show that the FCT is improved by up to an order of magnitude, missed deadlines are reduced and high-link utilization is attained. T-RACKs is shown to be lightweight and practical due to its minimal footprint on end-hosts. Finally, because it does not change TCP and adapts to any flavor, T-RACKS is very fit for multi-tenant public data centers.

References

[1] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of USENIX HotCloud*, 2010.

[2] M. Mattess, R. N. Calheiros, and R. Buyya, "Scaling MapReduce Applications Across Hybrid Clouds to Meet Soft Deadlines," in *Proceedings of IEEE AINA*, 2013.

[3] S. M. Rumble, D. Ongaro, R. Stutsman, M. Rosenblum, and J. K. Ousterhout, "It's time for low latency," in *Proceedings of USENIX HotOS*, 2011.

[4] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of data center traffic," in *Proceedings of ACM IMC*, 2009.

[5] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center TCP (DCTCP)," *ACM SIGCOMM CCR*, vol. 40, p. 63, 2010.

[6] H. Wu, Z. Feng, C. Guo, and Y. Zhang, "ICTCP: Incast congestion control for TCP in data-center networks," *IEEE/ACM Transactions on Networking*, vol. 21, 2013.

[7] G. Judd, "Attaining the promise and avoiding the pitfalls of TCP in the datacenter," in *Proceedings of 12th NSDI*, 2015.

[8] M. Alizadeh, S. Yang, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, "Deconstructing datacenter packet transport," *Proceedings of ACM HotNets*, 2012.

[9] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang, "Information-agnostic flow scheduling for commodity data centers," in *Proceedings of NSDI*, 2015.

[10] B. A. Greenberg, J. R. Hamilton, S. Kandula, C. Kim, P. Lahiri, A. Maltz, P. Patel, S. Sengupta, A. Greenberg, N. Jain, and D. A. Maltz, "VL2: a scalable and flexible data center network," in *Proceedings of ACM SIGCOMM*, 2009.

[11] T. Benson, A. Anand, A. Akella, and M. Zhang, "Understanding data center traffic characteristics," in *SIGCOMM*, 2010.

[12] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. G. Andersen, G. R. Ganger, G. A. Gibson, and B. Mueller, "Safe and effective fine-grained TCP retransmissions for datacenter communication," *SIGCOMM CCR*, vol. 39, p. 303, 2009.

[13] D. Nandita, C. Neal, C. Yuchung, and M. Matt, "An algorithm for fast recovery of tail losses," https://tools.ietf.org/html/draft-dukkipati-tcpm-tcp-loss-probe-01.

[14] H. Balakrishnan, S. Seshan, and R. H. Katz, "Improving reliable transport and handoff performance in cellular wireless networks," *Wireless Networks*, vol. 1, pp. 469–481, 1995.

[15] B. Cronkite-Ratcliff, A. Bergman, S. Vargaftik, M. Ravi, N. McKeown, I. Abraham, and I. Keslassy, "Virtualized congestion control," in *Proceedings of SIGCOMM*, 2016.

[16] K. He, E. Rozner, K. Agarwal, Y. J. Gu, W. Felter, J. Carter, and A. Akella, "AC/DC TCP: Virtual Congestion Control Enforcement for Datacenter Networks," in *SIGCOMM*, 2016.

[17] M. Allman and V. Paxson, "On estimating end-to-end network path properties," *SIGCOMM CCR*, vol. 29, pp. 263–274, 1999.

[18] Y. Zhang and N. Duffield, "On the constancy of internet path properties," in *Proceedings of ACM IMC*, 2001.

[19] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, Z.-W. Lin, and V. Kurien, "Pingmesh: A large-scale system for data center network latency measurement and analysis," in *SIGCOMM*, 2015.

[20] R. Mittal, V. T. Lam, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, and D. Zats, "Timely: Rtt-based congestion control for the datacenter," in *Proceedings of SIGCOMM*, 2015.

[21] S. Savage, N. Cardwell, D. Wetherall, and T. Anderson, "Tcp congestion control with a misbehaving receiver," *SIGCOMM Comput. Commun. Rev.*, vol. 29, no. 5, pp. 71–78, Oct. 1999.

[22] M. Allman, V. Paxson, and E. Blanton, "TCP Congestion Control," 2009, https://tools.ietf.org/html/rfc5681.

[23] NetFilter.org, "Packet Filtering Framework," http://netfilter.org/.

[24] B. Jenkins, "A hash function for hash table lookup," http://burtleburtle.net/bob/hash/doobs.html.

[25] iperf, "The Bandwidth Measurement Tool," https://iperf.fr/.

[26] S. Shukla, S. Chan, A. S.-W. Tam, A. Gupta, Y. Xu, and H. J. Chao, "TCP PLATO: Packet Labelling to Alleviate Time-Out," *IEEE JSAC*, vol. 32, no. 1, pp. 65–76, jan 2014.

[27] J. Huang, T. He, Y. Huang, and J. Wang, "ARS: Cross-layer adaptive request scheduling to mitigate TCP incast in data center networks," in *Proceedings of INFOCOM*, apr 2016.