T-RACKs: A Faster Recovery Mechanism for TCP in Data Center Networks

Ahmed M. Abdelmoniem, *Member, IEEE,* Brahim Bensaou, *Senior Member, IEEE*

Abstract—Cloud interactive data-driven applications generate swarms of small TCP flows that compete for the small switch buffer space in data-center. Such applications require a small flow completion time (FCT) to be effective. Unfortunately, TCP is myopic with respect to the composite nature of application data. In addition it tends to artificially inflate the FCT of individual flows by several orders of magnitude, because of its Internet-centric design, that fixes the retransmission timeout (RTO) to be at least hundreds of milliseconds. To better understand this problem, in this paper, we use empirical measurements in a small data center testbed to study, at a microscopic level, the effects of various types of packet losses on TCP's performance. In particular, we single out packet losses that impact the tail end of small flows, as well as bursty losses that span a significant fraction of small TCP congestion windows, and show a non-negligible effect of such losses on the FCT. Based on this, we propose the so-called, timely-retransmitted ACKs (or T-RACKs), a simple loss recovery mechanism that conceals the drawbacks of the long RTO even in the presence of heavy packet losses. Interestingly enough, T-RACKS achieves this transparently to TCP itself as it does not require any change to TCP in the tenant's virtual machine (VM) or container. T-RACKs can be implemented as a software shim layer in the hypervisor between the VMs and the server's NIC or in hardware as a networking function in a SmartNIC. Simulation and real testbed results show remarkable performance improvements.

Index Terms—Data-center, Cross Layer, Fast Recovery, TCP-Incast, Timeouts, Testbed.

I. INTRODUCTION

The recent growth in data-center deployments worldwide is reshaping how the Internet and its applications operate. New, cloud-based, data-driven applications have emerged over the past decade to harness the cost-effectiveness and scalability afforded by cloud computing. For storing and processing large data sets, most such applications rely on distributed programming and data storage frameworks such as Hadoop, HDFS, or Spark [1]. In such frameworks, master and aggregation nodes often require data transfers from tens to hundreds of worker nodes to build a 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 effectiveness of the application is not only correlated with the average latency of data delivery but also with the latency at the tail of the distribution. For example, in practice, the $90^{th}\%$

Ahmed M. Abdelmoniem is currently with CS Department, Assiut University, Egypt and KAUST, Saudi Arabia. Most of his contribution to this work was done while he was with CSE Department, HKUST, Hong Kong SAR, PRC. His work was supported under Grant HKPFS PF12-16707. E-mail: amas@cse.ust.hk

Brahim Bensaou is currently with Department of Computer Science and Engineering, HKUST, Hong Kong SAR, PRC. E-mail: brahim@cse.ust.hk

of the flow completion times (FCT) can be anywhere between two to four orders of magnitude worse than the median or the average latency.

In small scale private data-centers, CPU resources are often the bottleneck, and solutions that rely on task placement and scheduling already exist (e.g., [2]). In contrast, public data-centers seldom overload their server CPUs and usually have abundant computing resources; yet, they often adopt high over-subscription ratios in the network. As a result, network latency becomes the main performance bottleneck [3]. This is typical for many Internet-scale applications deployed on public (IaaS) clouds such as Microsoft Azure or Amazon EC2.

Measurements in real production data-centers [4–8] have shown over the years that the applications that produce small traffic flows predominate and that incast congestion events and excessive packet losses are frequent. To circumvent such problems, large corporations such as Microsoft, Facebook, and Google dedicate well-structured data-centers to deploying their time-sensitive applications. Smaller-scale private data-centers address the problem by deploying homogeneous custom-designed TCP variants (e.g., DCTCP [6]) on all the VMs in the data-center. In stark contrast, multi-tenant and public data-centers, where many-to-one (or many-to-many) communication patterns predominate, are populated with a large variety of versions of TCP with different behaviors in the face of congestion [5, 9, 10]. As a direct consequence unfairness in congestion resolution is inevitable and often leads to repeated packet losses and a long-tailed latency distribution for small flows. In particular, since commodity Ethernet switches are the backbone of all intra-data-center communications, their small buffer space can quickly be fully occupied by a few (large) TCP flows.

This raises two issues: *i)* small flows do not last long enough to be able to grab their fair share of the buffer from ongoing large flows, as their TCP sending window cannot grow large enough before a packet loss is experienced; *ii)* when a sudden swarm of such small flows (usually co-flows) surges, while the buffer is occupied by other flows, incast congestion loss events become inevitable. In this case, a burst of correlated packet losses from many such flows takes place. Bursty losses with small congestion windows often leave an insufficient TCP flight size to trigger TCP's fast-retransmit and recovery mechanism. As a consequence, small flows often experience timeouts. The retransmission timeout (RTO), which is orders of magnitude larger than the actual round-trip time (RTT), contributes thus the lion's share to the long FCT and number of missed deadlines experienced by small flows in data-centers.

In this paper, we study the impact of RTO on the performance of TCP applications in data-centers and propose a

simple mechanism to shield small TCP flows from the negative effects of the RTO, without changing TCP. Our methodology adopts a two-phased approach:

i) First, to fully understand the impact of the RTO on the FCT of small flows, we conduct an empirical study of the loss events in a small data-center, by examining the nature of the recovery mechanism invoked by TCP for each segment loss. To this end, we trace TCP traffic flows microscopically at the socket-level in the Linux Kernel. Then by analyzing the collected traces, we study the frequency of occurrence of the two TCP loss recovery mechanisms (viz., RTO[1] and Fast Retransmit and Recovery (or FRR)) in relation to the TCP window size. We show that tail-end losses and bursty losses primarily cause RTOs, and while they have a less dramatic effect on the latency of large flows, their impact on the performance of small flows is tremendous. *ii)* Second, to prevent RTOs from artificially inflating the actual loss recovery delays of small flows intra-data-centers, and without modifying TCP[2], we propose, implement and study the performance of a new mechanism to conceal the long retransmission timeout. This mechanism forces TCP in the VM to go into the FRR mode whenever a segment is estimated to be likely to experience a timeout, long before it actually does. We implement the resulting so-called T-RACKS in a real testbed and study its performance with realistic traces[3].

In the remainder, supported by an empirical study, we show in Section II the dramatic impact of the RTO on the performance of small flows. In Section III, we present the proposed methodology and system design. 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

Before we start discussing our empirical study of TCP and presenting our solution, let us first shed some light on the motives that led us to adopt such a non-traditional approach by contrasting it against alternative methods. In particular, while our approach is straightforward, it turns out to be very effective because it relies first on fully understanding the large number of incremental mechanisms that have been added over the years to TCP.

Many alternative schemes proposed in the literature deal with TCP congestion in data-centers in a classic Internet-centric approach by invoking mechanisms such as RED. This approach is flawed because of three major reasons: *i)* RED is a mechanism that was designed for the Internet. Its goal is to reduce the average queuing delay experienced by packets in the huge routers' buffer, which contributes a large proportion

of the end-to-end delay and delay-jitter. In contrast, data-centers use high-speed switches with small buffers; therefore, the contribution of queuing delay to the total FCT is not as dramatic as in the case of the Internet, regardless of the buffer occupancy. And so, maintaining a small average queue does not help the FCT. *ii)* With increasing link speeds in modern data-centers, the interplay between propagation delay and transmission delay is transformed, rendering the control mechanisms valid for one no longer valid for the other. For example, in data-centers with 1Gbps network interfaces, the transmission time of a single IP packet of 1500 bytes is about 12 microseconds; the round trip time over a 600m path at the speed of light is about 6 microseconds. So, there can be at most one packet spread over a link between any two adjacent interfaces in the network (e.g., server NIC, to ToR port, ToR port, to Aggregation Port, ...). In contrast, with 40 Gbps network interfaces, it takes only 0.3 microseconds to complete the transmission of an IP packet, yielding a possible flight of up to 33 IP packets per hop. So early detection and notification via buffer thresholds with the small buffers that exist in the switches are ineffective, and excessive packet losses are inevitable. *iii)* Packet losses in TCP per-se are not the reason for these problems; excessive congestion is. Packet losses are merely symptoms of congestion, so there is no reason to try to curtail them completely as long as we can control their frequency by recovering from losses fast. In fact, curtailing packet losses completely results in a non-competitive TCP behavior that yields poor performance in heterogeneous TCP environments. For example, pitting TCP Vegas against TCP New Reno, Cubic or DCTCP results in poor performance for the former.

As a consequence, eliminating packet losses in data-center networks while maintaining a high link utilization is not helpful in reducing the FCT. Moreover, several measurement studies [5, 9, 10] have been conducted on data-centers and have shown that latency in such environments varies greatly. Instead, we propose to pinpoint the true reason for increased delays in data-centers and to tackle such reasons directly [11]. To this end, we aim to further understand the reasons behind this behavior and so we deep-dive into the packet level analysis of the flows and the TCP socket state variables at a microscopic level to understand TCP behavior and its loss recovery mechanisms.

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 RTO_{min} 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, and as such it is inappropriate for public data-centers where multiple tenants can upload their own version of the OS; and, *ii)* there is no "magical" value of RTO_{min} that fits all possible environments. For instance, a RTO_{min} that works inside the data-center (e.g., between a web server and the back-end database server) will lead to spurious timeout events for Internet-facing connections (e.g., the connection between

---

[1]The minimum RTO is 200ms in Linux and 300ms in Windows

[2]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.

[3]An earlier version has been published in IEEE INFOCOM 2018 [11]. The implementation, simulation and experimental code and scripts are publicly available at http://ahmedcs.github.io/T-RACKs.

Table I: *TCP API Calls Intercepted by LossProbe Module*

Function Call	Description
tcp_set_state	Handles and update TCP connection state
tcp_v4_do_rcv	Handles the arrival of all types of TCP segments
tcp_retransmit_skb	Retransmits one socket buffer, where policy decisions and retransmit queue state updates are done by the caller
tcp_v4_send_check	Computes a TCP checksum

the web administrator workstation and the server in the data-center).

A. Impact of RTO on The FCT

In data-centers, partition/aggregate applications that generate small flows are challenged by the presence of small buffers, large initial sending windows, inadequate RTO_{min}, or slow-start exponential increase. This combination of hardware and TCP configuration frequently leads to timeout events for such applications. In particular, when the number of flows they generate is large and roughly synchronized, incast-TCP synchronized losses occur. As the loss probability increases linearly with the number of flows [13], the flow synchronization and the excessive losses lead to throughput-collapse for small-flows.

To illustrate this, consider a simplified fluid-flow model with N flows sharing a link of capacity C equally. Let B be the flow size in bits and n be the number of RTTs it takes to complete the transfer of one flow. The optimal throughput ρ^* can be simply approximated as the fraction of the flow size to its average transfer time: $\rho* = \frac{B}{n\tau + \frac{BN}{C}}$. That is, it takes BN/C to transmit the B bits plus an additional queuing and propagation delay of τ seconds for each of the n RTTs. In practice, when TCP incast congestion involving N flows results in throughput-collapse, the flow experiences one or more timeouts and recovers after waiting for RTO. Then, the actual throughput writes: $\rho = \frac{B}{RTO + n'\tau' + \frac{BN}{C}}$, Typically $n' \geq n$ and $\tau' \geq \tau$. In addition, in data-centers, the typical RTT is around 100μs, while existing TCP implementations impose a minimum RTO (i.e., RTO_{min}) of about 100 to 200ms[4]. As a consequence, in general large flows yield values of n' such that $n'\tau' \gtrsim RTO$. In contrast, small flows only last a few RTTs, therefore $n'\tau' \ll RTO$. And so, when a small flow experiences a loss that cannot be recovered by 3-duplicate ACKs, it systematically incurs a FCT that is orders of magnitude larger than it should.

B. Analyzing TCP congestion recovery

To investigate why packet losses seem to affect large flows only marginally, yet degrade the performance of small flows dramatically, we collected and examined socket-level TCP state information from a Websearch workload [6], in a small-scale data-center testbed. First, we implemented a socket-level monitoring module, named hereafter "LossProbe", based on KProbes/JProbes [14] in the Linux Kernel. Probes are dynamic debugging tools which in our case, allow us to intercept different TCP event handlers and API calls as listed in Table I

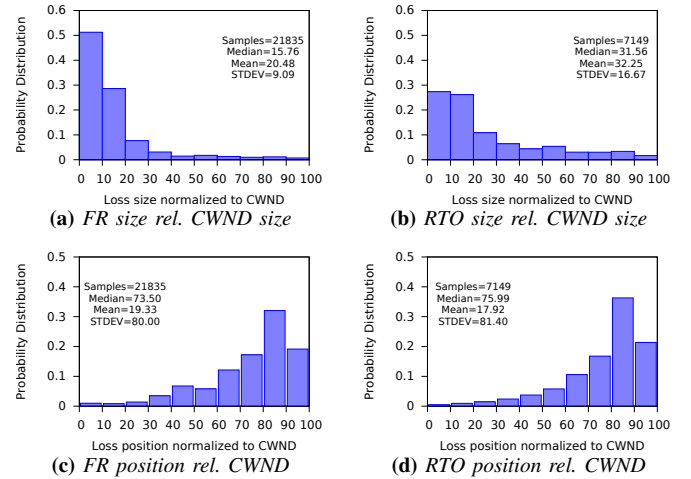[4]Linux uses 200ms and Windows uses 300ms

(a) *FR size rel. CWND size*

(b) *RTO size rel. CWND size*

(c) *FR position rel. CWND*

(d) *RTO position rel. CWND*

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

Table II: *TCP Socket-level State Info. logged by LossProbe Module*

Data Type	Variable	Description
uint	lost_out	Count of lost packets
uint	prr_out	Count of pkts. sent during recovery
uint	prr_delivered	Count of pkts. delivered during recovery
uint	prior_cwnd	Congestion window at start of recovery
uint	prior_ssthresh	SSThresh saved at recovery start
uint	total_retrans	Count of retrans. for entire connection
uint	retransmit_high	Highest sequence # of retransmitted data
uint	lost_retrans_low	Lowest sequence # of retransmitted data
uint	packets_out	Count of segments currently in flight
uint	retrans_out	Count of packets retransmitted
int	undo_retrans	Count of undo-able retransmissions
int	rcv_tstamp	Timestamp of last received ACK
int	lsndtime	Timestamp of last sent data packet
int	retrans_stamp	Timestamp of the last retransmit
int	recovperiod	Duration of total recovery period
int	maxrecovperiod	Maximal recovery time experienced

where we log the target TCP socket-level state information. The module works as follows:

1) Jprobe requires the address of the kernel function to trace; hence the target TCP handlers of the events of interest have to be identified from the Linux kernel source code base [15]. For example, **tcp_retransmit_skb** is the function called in the kernel to retransmit a TCP segment.

2) Then, a handler function is defined that will perform certain actions upon entry of the traced function (e.g., print the debugging message when the target kernel function is invoked). In the probe module, that function is defined for convenience with the same name as the original probed function (e.g., **jtcp_retransmit_skb**) and jprobe calls it upon entering the original function.

3) The monitoring module is dynamically installed into the kernel, and the probed workload (or experiment) is invoked. Upon entry of the target functions, the jprobe function defined in the module, is called to collect the state information of interest and write them into an RAM-buffer which in turn is flushed periodically and asynchronously onto the file system to avoid stalling the datapath artificially.

Since we do not have actual real world workload, we custom-built a traffic generator, that replicate a Websearch work-

load [6] consisting of thousands of flows and collect measurements on the data listed in Table II from all the servers in our testbed[5].

We summarize our findings in several figures to reflect TCP's behavior with respect to the mechanism invoked to recover from packet losses (e.g. Fast Retransmit and Recovery or Re-transmission Timeouts[6]). Fig. 1a shows the distribution (on the ordinate) of the size of each retransmission (on the abscissa) for the FRR-based recovery events. The size of a retransmission is calculated by subtracting the sequence number of the first segment from that of the last segment in a single recovery round which is then normalized by the size of the congestion window $Cwnd$[7]. Similarly, Fig. 1b shows the same metric for RTO-based recovery events. Fig. 1c and Fig. 1d show the distribution (on the ordinate) of the loss index or position (on the abscissa) in case of FRR and RTO-based recovery, respectively. The index points to the first retransmitted segment (for several consecutive segments losses), relative to $Cwnd$, when the segment was first transmitted (i.e., before a loss is detected)[8].

Analyzing these results, we can draw the following conclusions: Fig. 1a suggests that FRR loss size is distributed over the range of packets in the window with a positive skewness towards the first few fractions of the window (i.e., probability of losing more than 30% of the window is insignificant). However, Fig. 1b shows that this is not the case for RTO, which seems well distributed with positive skewness towards the tail end of the window (i.e., the probability of losing more than 30% of the window is significant). Also, we can see that there are only a few RTOs far away from the tail; these represent lost packets within the same congestion window. Fig. 1c points out that losses at the tail of the window occur with higher frequency for RTO events, however in the case of FRR, the $Cwnd$ is relatively large enough for TCP to receive a sufficient number of duplicate Acks, which allows for Fast-Recovery. Similarly, Fig. 1d clearly shows a similar trend 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, and eventually, RTO recovery occurs. To elaborate, we see in Fig. 2 that typically $Cwnd$ for the flows that experience RTO is smaller than its counterpart for those that recover via FRR.

In data-centers, the size of the pipeline is small: typically, with an RTT of $100\mu s$, 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 phenomenon highlights how TCP's performance can be degraded
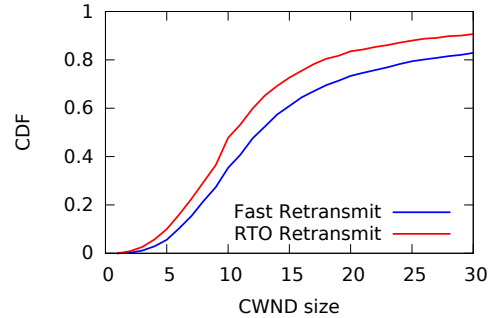

Figure 2: *CDF of CWND at the transmission of the lost packet*

when operating in small windows regime in a small buffer with high-bandwidth low-delay environments like data-centers. The effect on the FCT is more severe for small, time-sensitive flows, that generally last only a few RTTs, but that are in practice compelled to wait for 2 to 4 orders of magnitude extra time due to the RTO_{min} rule.

III. System Design and Implementation

T-RACKs design is based on the following observation: *packet losses are inevitable in TCP. So the key to reducing the long latency and jitter is not to try and avoid losses completely but instead try to avoid long waiting after the losses occur.* In achieving this, T-RACKS must also: (R1) improve the FCT of latency-sensitive applications by expediting the transmissions of small flows' packets. (R2) be friendly to throughput-sensitive large flows (i.e., it must not sensibly degrade the throughput of large flows to satisfy the delay requirements of latency-sensitive flows). (R3) be compatible with all existing TCP versions (i.e., it must not impose modifications inside the virtual machines, and if any are needed, they shall be in the hypervisors, which are fully under the control of the data center operator. It also must not require any extra special hardware). (R4) Finally, the mechanism must be simple enough to be easily deployable in real data-centers.

In this perspective, T-RACKs actively infers packet losses by monitoring (in the hypervisor) per-flow TCP ACK numbers and proactively triggers the FRR mechanism of TCP to take action whenever and RTO is deemed to be likely to take place in the near future. The goal is to help small TCP flows, that would otherwise experience a timeout, recover fast via the FRR instead of waiting for TCP's RTO_{min}. The proposed mechanisms intervene only when the loss is almost certain, leading to a significant improvement of recovery times, and hence the FCT. T-RACKS design relies on the following arguments: *i)* all TCP versions adopt the FRR mechanism as a way to detect and recover from losses fast. So, if the FRR mechanism can be forced into action by the hypervisor, regardless of the nature of the loss, the resulting system would be transparent to the TCP protocol in the VM and would require no changes to TCP in the VM; *ii)* TCP relies on a small number of duplicate ACKs to activate FRR; however, often (especially for small-flows), there aren't enough packets in flight to trigger duplicate ACKs. To achieve this, we propose to use "spoofed" TCP ACK signaling from the hypervisor to the VM. In this perspective, the hypervisor maintains a per-

---

[5]Code for the LossProbe module is publicly available at https://github.com/ahmedcs/TCP_loss_monitor/

[6]Each figure shows the aggregate of all servers in the data-center.

[7]The bar for 0-10, refers to the probability that 0-10% of $Cwnd$ is lost while a bar for 90-100 refers to the probability that 90-100% of $Cwnd$ is lost

[8]A bar for 0-10, refers to the probability that a loss occurs for the first 0-10% of the segments in $Cwnd$; whereas, a bar for 90-100 refers to the probability that the loss occurred in the last 90-100% of the segments of $Cwnd$

flow timer $\beta = \alpha * RTT + rand(RTT)$ to wait for the ACKs before it triggers FRR with spoofed duplicate ACKs.

Note that, while finalizing our work, we realized the similarity of our idea is similar in spirit to the so-called TCP SNOOP protocol [16], which retransmits lost segments on behalf of the communicating end-points to filter out bit-errors in low-speed wireless networks. As such, TCP SNOOP also could 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 or hypervisor), which requires an ample buffer space in data-centers. T-RACKs, in contrast, triggers the retransmissions from the actual TCP protocol in the VM instead of buffering and retransmitting the packets itself. It requires no packet buffering at all; it only relies on memorizing a few state variables from the last segment, and the final ACK received of each flow.

A. T-RACKs Algorithm

Algorithm 1: T-RACKs Packet Processing

---
1   /* Initialization   */
2   Create an in-memory flow cache pool;
3   Create a flow table and reset flow information;
4   Initialize and insert NetFilter hooks (for the NetFilter implementation);
   Input: α # of RTTs to wait before retransmitting ACKs
   Input: γ a threshold in bytes to stop tracking a flow as small
   Input: ϕ the dupACK threshold used by TCP flows
   Input: t: the current local time counted in jiffies
5   Define x: the exponential backoff counter
6   **Function** *Outgoing Packet Event Handler (Packet P)*
7   f=Hash(P);
8   **if** *SYN(P) or !f.active* **then**
9   Reset Flow (f);
10   Extract TCP options (i.e, TStamp, SACK, etc);
11   Update the flow information and set f.active;
12   **if** *DATA(P)* **then**
13   Update flow info (i.e., last seq#, etc);
14   f.active_time = now();

15   **Function** *Incoming Packet Event Handler (Packet P)*
16   /* For ACKs: extract and update flow information from incoming header   */
17   f=Hash(P);
18   **if** *f.long_lived* **then** return ;
19   **if** *ACK_bit_set(P)* **then**
20   Extract required values (e.g., seq#, ack#, etc);
21   **if** *New ACK* **then**
22   Update flow entry and state information (e.g., RTT);
23   Update last seen ACK number from receiver;
24   Reset f.dupAck_Nr = 0;
25   Reset f.ACK_time = now();
26   **if** *f.lastAckNo $\geq \gamma$* **then** f.long_lived = true ;
27   **else**
28   **if** *Duplicate ACK* **then**
29   f.dupAck_Nr ++;
30   /* Drop extra dup-ACKs   */
31   **if** *f.resent > 0* **then** Drop Dup ACK ;
32   Update TCP headers (i.e., TStamps, SACK, etc);

Algorithm 2: T-RACKs Timeout Handler

---
1   Create and initialize a timer triggere every 1 ms;
2   **Function** *Timer Expiry Event Handler*
3   **for** *Flow (f) $\in$ FlowTable* **do**
4       $\beta = \alpha * f.RTT + rand(f.RTT)$);
5   **if** *!f.active or f.long_lived* **then** Continue ;
6   T = MAX(f.ACK_time, f.active_time);
7   **if** *now() - T $\geq \beta$* **then**
8   Resend last ACK #($\phi - f.dupAck\_Nr$) times;
9   Set f.resent_time = now();
10   Set x = 2;
11   Continue;
12   **if** *now()-f.resent_time $\geq (\beta \ll x)$* **then**
13   resend ACK one more time;
14   x = x + 1;
15   Continue;
16   **if** *(now()-f.ACK_time) $\geq RTO_{min}$* **then**
17   stop T-RACKs recovery;
18   soft reset flow (f) recovery state;
19   Continue;
20   **if** *(now()-f.active_time)$\geq$1* **then** deactivate_flow(f) ;

The T-RACKs algorithm consists broadly of three major functions: the first two are in charge of maintaining per-flow state information on the server (hypervisor) on arrival and departure of packets, shown in Algorithm 1 and the third is a timer event handler described in Algorithm 2. In the initialization in (lines $2-5$) of Algorithm 1, an in-memory flow cache pool is created to track new flow arrivals. This approach speeds up flow objects creation. A hash-based flow table is created and manipulated via the Read-Copy-Update (RCU) synchronization mechanism to efficiently identify flow entries. Other parameters and variables are set in this step, as well. Before each TCP segment departure, T-RACKs performs the following actions: *i)* in line 7, the packet is hashed using its 4-tuple (source and destination IP addresses and port numbers), and the corresponding flow is identified; *ii)* in lines $9-11$, if this is an SYN packet 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 new flow record; and *iii)* in lines $13-14$, if this is a Data packet, then the $last_sent$ sequence number and time for this flow are updated.

Next, upon each TCP ACK arrival, the algorithm performs the following actions: *i)* in lines $17-18$, the flow entry is identified using its 4-tuple; if the flow is large, it will subsequently be ignored by T-RACKs. By doing so, the complexity of the scheme is reduced; *ii)* in lines $22-26$ if the ACK sequence number acknowledges a new packet arrival, the last seen ACK sequence number and time is updated. The dupACK counter is reset. If the accumulated flow size exceeds a threshold γ it is marked as a large flow (to be able to stop tracking it); *iii)* in lines $29-31$, if ACK number acknowledges an old packet (i.e., if this is a duplicate ACK), then we drop dupACKs if the flow is in recovery mode, otherwise increment the number of dupACKs seen so far; *iv)* in line 32, 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 periodic global timer expiry events and performs the following actions for all **active non-large**
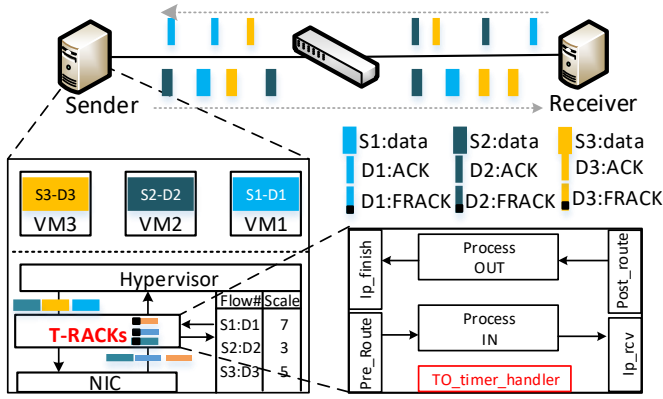
**Figure 3:** *T-RACKs System: consists of an end-host module that tracks TCP flows incoming ACKs and generates FAKE retransmitted ACKs upon T-RACKS timeout*

Figure 4: *RTT variation between transmission time and time of retransmission(i.e $\Delta RTT = RTT_{rt} - RTT_t$)*

flows in the table. In a typical implementation this timer lasts 1 ms and is processed regularly with the OS clock timer interrupt (i.e., does not require the special high-resolution timers): *i)* in lines $4 - 11$, if no new ACK acknowledging new data has arrived for β seconds since the last new ACK arrival, the flow is deemed to be likely to experience a timeout in the future. T-RACKs enters into action, spoofs an ACK using the last successfully received ACK sequence number, and sends it to the sending process or VM residing on the same end-host. An exponential backoff mechanism is activated to account for various dupACK thresholds set by the sender's TCP or OS. *ii)* In lines $13 - 15$, if using timer β backed-off by the number of retransmissions x of the spoofed ACK, does not yield a recovery, i.e., the flow still did not receive a new ACK, another spoofed ACK is created and sent to the corresponding sender. To ensure T-RACKS is not sending spurious spoofed dupAcks, the algorithm backs-off exponentially; i.e., after each transmission of a spoofed Ack, timer β is doubled. *iii)* In lines $17 - 19$, if the backoff time approaches the RTO_{min} (i.e., 200ms), we stop triggering Fast-Retransmit (by resetting the soft state) and letting the sender's TCP RTO timer handle the recovery of this segment. *iv)* In line 20, if the inactivity period exceeds 1 sec, flow (f) entry is hard reset.

B. T-RACKs System Implementation

Algorithm 1 relies on TCP header information of ACK packets to maintain per-flow TCP state information. In this paper, we only consider a lightweight end-host (hypervisor) shim-layer implementation to achieve this[9]. This approach is perfectly feasible even for production data-centers, because the number of flows in a server in production data-centers has been reported to be small in general not exceeding 30-40 [6]. In addition, the number of flows tracked by T-RACKs is further reduced on average by only tracking small-sized flows, abandoning large flows whenever they grow to reach a certain size threshold. 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
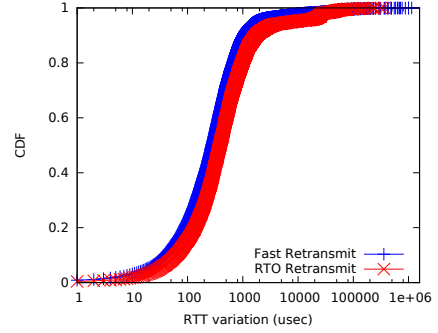
SYNs packets arrive or a flow sends data after a long silence period. For instance, referring to Figure 3, when VM1 on the sender end-host established a connection with its peer (or destination) VM on the receiving end-host, a new flow entry ($S2 : D2$) is created in the table as shown in Fig.3. Also, not shown in the Algorithm code, flow entries are cleared from the table whenever a connection is closed (following the TCP connection tear down FIN/FIN-ACK) or after a pre-set inactivity time threshold is exceeded. The flow table could track many relevant TCP-related per-flow state information, however, for T-RACKs to perform the fast recovery function, it needs to track a minimal set of TCP state variables (including the highest ACK sequence number seen so far and the arrival time of the most recent ACK).

The T-RACKs system uses a flow table to store and update TCP flow information, including the last ACK number, the last sent sequence number, the corresponding times, the RTT for the flow measured using the TCP timestamp option, as well as the optional TCP Sack information[10] for each outgoing TCP flow. T-RACKs intercepts the incoming ACKs and outgoing Data to update the current state of each tracked small flow. When packets are dropped by the network and the receiver receives enough out of sequence Data to generate sufficient dupACKs (real ones), the loss is recovered via FRR from the VM without the intervention of T-RACKs. However, when the receiver fails to receive enough Data segments to generate enough real dupACKs to trigger FRR, then T-RACKs intervenes after a timer (1ms) by sending spoofed dupACKs (or RACKs for retransmitted-ACKs) to the sender. Typically, the sender would receive enough dupACKs and RACKs to trigger FRR and retransmit the lost segment within a reasonable time (in this case 1ms), long before the TCP RTO (200ms) is reached.

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 timestamp of the last non-dupACK packet. The system in spirit is similar to recent works in [17, 18] that aim to enable virtualized congestion control in the

---

[9]We note that, in higher speed networks, T-RACKs could equally be implemented as a networking function on smartNICs

[10]Note that if TCP Sack is active, TCP's response to duplicate ACKs is different from the standard behavior, therefore we need to take this into account to elicit a proper reaction.

hypervisor without cooperation from the tenant VM, however these approaches require fully-fledged TCP state information tracking and implementing full TCP finite-state machines in the hypervisor including packet queuing. In contrast, T-RACKs tries to minimize the overhead by tracking the minimal amount of necessary information and implementing only a subset of the retransmission mechanism while letting the VM do the actual work of transmission and queuing.

Complexity: T-RACKs Complexity resides in its interception of ACKs to update the last seen ACK information. Since, it does not perform any computation on the ACK packets, it does not add much to the load on the hosting server nor to the latency[11]. 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 small** 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 can sustain rates of 60 Gbps of packet processing. Hence, the little processing required by the insertion of T-RACK state would not affect the TCP throughput.

Spurious retransmissions: T-RACKs may possibly introduce spurious retransmissions making in-network congestion worse. However, this boils down to answering similar question when choosing the correct RTO value in TCP. For this purpose, we refer to a previous study [19], that mostly showed that even when a relatively bad RTT estimator is used, setting a relatively high minimum RTO, can help avoid many of the spurious retransmissions in WAN transfers. This fact is supported by a subsequent study [20] that shows significant changes (or variance) in Internet delays. Recent works [21, 22] show similar behavior within current data-centers. In our testbed, we observed noticeable variation in the measured RTT. To quantify this, we measured the difference in RTT values collected at the time of the first transmission of a packet and then at the time of fast retransmission or RTO retransmission. From the collected data, a considerably large variation, ranging from a few hundred microseconds at the $\approx$20th percentile to a thousand microseconds at the $\approx$80th percentile, is observed in the smoothed RTT at the TCP sockets, as shown in Fig. 4. 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 the rapid retransmission and the risk of causing a 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 of the spurious retransmissions 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 ACK RTO (β) calculations until either the loss is recovered or TCP's default RTO (i.e., RTO_{min}) is close enough to be reached.

Synchronization of retransmissions: Since T-RACKs relies on a timer for ACK recovery, such timer may result in the synchronization of retransmissions from different VMs on different hosts leading to incast-like congestion. We studied this 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 ACK RTO, ultimately resulting in fewer flows experiencing repeated timeouts. We adopted this approach and added a random delay in the calculation of the RTO β, as shown above in the algorithms.

TCP Header manipulation: TCP does not accept any packet with an inconsistent timestamp, hence the timestamps are updated per ACK arrival with the local $jiffies$ variable to keep the consistency of timestamps whenever RACKs are sent. For SACK-enabled TCP, 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 [23] 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 a solution would require the 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 RACKs). Worth mentioning also is that RACKs are generated from the hypervisor layer, which is under the control of the trusted data-center operator.

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

IV. SIMULATION ANALYSIS

In this section, we study the performance of T-RACKs to verify if it can achieve its goals in small-scale and large-scale simulation scenarios. We first conduct a number of packet-level simulations using ns2 and compare T-RACKs performance to the state-of-the-art schemes. (For brevity, we refer to T-RACKs as RACK in the figures.)

A. Simulations in a Dumbell Topology

To study how TCP behaves in response to packet losses and how likely it may recover quickly with the help of T-RACKs,

---

[11]ACKs is updated in some instances (e.g., to insert fake SACK block to signify a small gap in the SACKed numbers. Otherwise RACK packets would be ignored by TCP).

we conducted several packet-level simulation experiments to cover a large variety of TCP and AQM combinations. We also conducted simulation experiments using the congestion control mechanisms code imported from the Linux kernel (i.e., New Reno and Cubic). We use ns2 version 2.35 [24], which we have extended with T-RACKs mechanism inserted as a connector between nodes and their link in the topology setup. Besides, we patched ns2 using the publicly available DCTCP patch. We use in our simulation experiments link speeds of 1 Gbps for sending stations, a bottleneck link of 1 Gb/s, low RTT of 100 μs, the default TCP RTO_{min} of 200 ms and TCP initial window of 10 MSS. We use a rooted tree topology with a single bottleneck at the destination and run the experiments for 15 sec. The buffer size of the bottleneck link is set to 100 packets, which is more than the bandwidth-delay product in all cases. We first designed two simulation scenarios:

1) CASE 1: Small flows.
2) CASE 2: Large flows coexisting with small flows.

In CASE 2, the ratio of small flows to large flows is set to 3, and large flows send data for the whole duration of the experiment. In both cases, a small flow sends a 14.6KB file (i.e., 10 MSS) before it completes its transmission. Small flows start with a random inter-arrival time that is drawn from an exponential distribution with a mean equal to the transmission time of one packet; this allows us to create clusters of small flows that start almost simultaneously, to emulate incast traffic. This process is repeated once every 3 sec, which gives five rounds of incast traffic, during the simulation. In each round, we randomize the order of the servers generating the flows according to a uniform distribution. We study packet losses, the likelihood of fast recovery, and recovery time. We first study TCP newReno with Droptail, RED-ECN, and Random Drop AQMs, and DCTCP covering the most common TCP and AQM settings.

First, we run the experiments without T-RACKs for 20 and 80 flow traffic scenarios generated according to CASE 1 (i.e., containing only small flows). Figures 5a and 6a show the average FCT for different schemes. It appears that the FCT is significantly high for all-schemes exceeding 200ms for most flows (more than $\approx$35% and $\approx$95% for 20 and 80 flows, respectively). This result indicates that most flows are experiencing RTOs. We repeat the same simulation, enabling T-RACKs on the end-hosts, to mitigate the RTOs. Figures 5b and 6b show the FCT for different schemes with T-RACKs for 20 and 80 flows scenario in CASE 1. The results show significant improvement in the average FCT for the two scenarios for all schemes. Specifically, T-RACKS could reduce the average FCT range significantly for both 20 and 80 flows scenarios. In the 20 flows scenario, at the 95th% (as highlighted by the horizontal black line), the reduction in average FCT is up to $\approx$15 times (i.e., from 200ms down to 13ms). In the 80 flows scenario, it reduces the FCT at the 95th% by $\approx$3 times for DropTail and DropRand, by $\approx$8 times for RED-ECN and DCTCP, respectively.

Now we conduct simulations for CASE 2, which introduces large flows (i.e., background traffic) to CASE 1. This case is used to study T-RACKs under stress. Figure 7a and Figure 7b
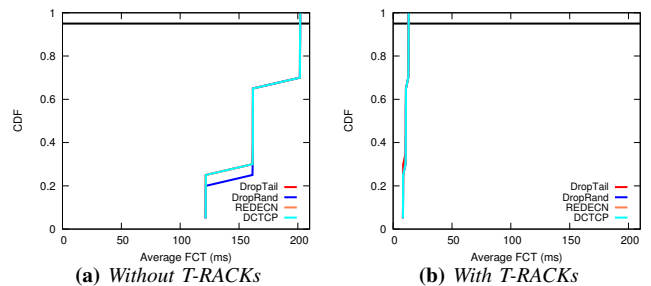

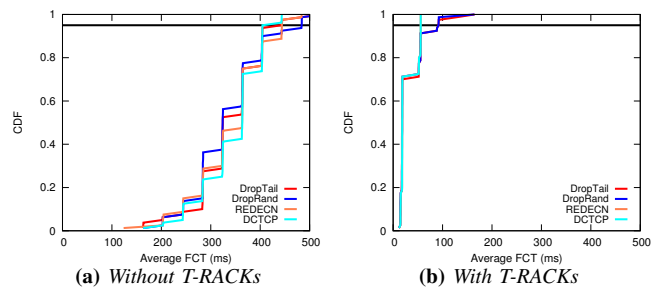**Figure 5:** *CDF of average FCT for CASE 1 with 20 flows*

Figure 6: *CDF of average FCT for CASE 1 with 80 flows*

show the results without and with T-RACKs for 20 flows scenario. The results show a significant reduction in the average FCT of small flows. In the 20 flows scenario, T-RACKs reduces the average FCT on the 95th% by $\approx$1.6 time, $\approx$1.9 times, $\approx$14 times and $\approx$14 times for DropTail, DropRand, RED-ECN, and DCTCP, respectively. Figure 8a and Figure 8b show the results without and with T-RACKs for the 80 flows scenario. In this scenario, T-RACKs can further improve the performance, and it manages to reduce the average FCT at the 95th% by $\approx$14 times, $\approx$1.5 times, $\approx$1.4 times and $\approx$37 times for DropTail, DropRand, RED-ECN, and DCTCP, respectively. These results show that T-RACKs improves the performance more in heavy load scenarios with background traffic.

B. Simulations in Data-center Topology

End-host based schemes are assumed by default to be scalable. To verify this, we simulate T-RACKs on a larger scale network with varying workloads and flow size distributions.

For this purpose, we conduct another packet-level simulation using a spine-leaf topology with nine leaf switches and four spine switches using links of 10 Gbps for end-hosts and an over-subscription ratio of 5^{12}. We again examine scenarios with TCP-NewReno, TCP-ECN, and DCTCP operating with Droptail, RED, and DCTCP AQMs, respectively. We use a per-hop link delay of 50 μs, TCP is configured with the default TCP RTO_{min} of 200 ms and an initial window of 10 MSS. Persistent connections are used for successive requests. Finally, buffer sizes on all the links are set to be equal to the bandwidth-delay product between end-points within one physical rack.

The flow size distribution for workload 1 (which represents Websearch flow sizes distribution [6]) and workload 2 (which represents datamining flow sizes distribution [9]) are shown in

---

[12]the typical over-subscription ratio in current production data-centers is in range of 3 to over 20
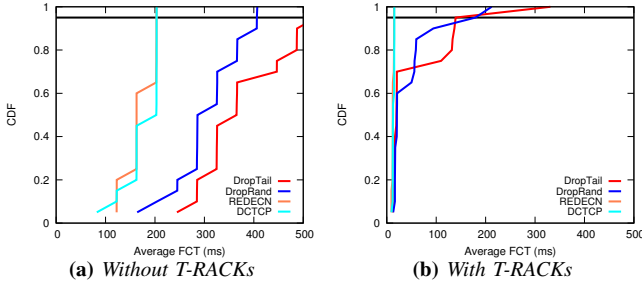
(a) *Without T-RACKs*   **(b)** *With T-RACKs*

Figure 7: *CDF of average FCT for CASE 2 with 20 flows*

(a) *Without T-RACKs*   **(b)** *With T-RACKs*
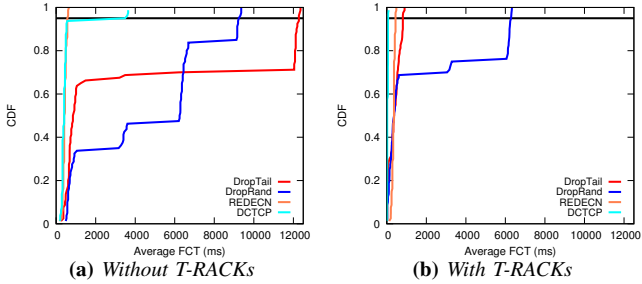
Figure 8: *CDF of average FCT for CASE 2 with 80 flows*

(a) *CDF of flow size*   **(b)** *CDF of inter-arrival time*

Figure 9: *Flow characteristics: (a) Actual Flow size distribution (b) Inter-arrival times for various network load*

Fig 9a which capture a wide range of flow sizes. The flows are generated randomly from any source host to any other destination host with the arrivals following a Poisson process with various average flow arrival rates to simulate different network loads. Fig 9b shows the inter-arrival times distribution for various traffic loads ranging from 30% to 90%.

We report the average FCT for small flows the average FCT for all flows[13] as well as the total number of timeout events in each case. In the simulation, the T-RACKs threshold γ is set to infinity (i.e., all flows are tracked including large flows). The T-RACKs RTO, the timeout to trigger spoofed dupACKs, is set to 10 times the measured RTT in this experiment. That means if an ACK with the expected sequence number is not delivered within 10 RTTs then RACK packets are spoofed to trigger the FRR.

The average FCT for small and all flows, as well as the total number of timeouts experienced by all flows for workloads 1 and workload 2 are shown in Fig. 10. Note that for both workloads, the FCT of small flows is dramatically degraded when they experience a timeout regardless of the TCP congestion control version or AQM mechanism in operation. In contrast, when T-RACKs is activated, it helps small flows the most by improving their FCT as a by-product of reducing the number of timeouts they experience. We also note that the overall FCT decreases for all flows for two reasons: 1) the threshold γ enables all flows to benefit from T-RACKs and 2) small flows finish quicker leaving network resources for larger ones. We notice that for workload 2, with almost 80% of the flows being less than 10KB and hence experiencing lesser timeout events overall, DCTCP can improve the FCT. This improvement is because DCTCP's ability to regulate the persistent queue length (i.e., there are few large flows to fill

---

[13]Flow sizes in the range [0-100KB] are considered to be small, flow sizes in the range [100KB - 10MB] are considered to be medium and flows sizes from 10MB and upwards are considered to be large flows.
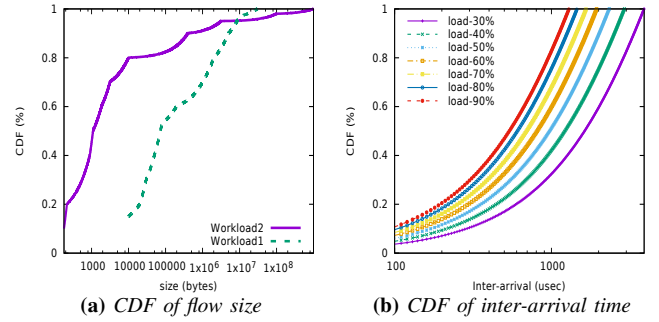
the buffer, unlike in workload 1).

C. Sensitivity to Choice of T-RACKS RTO

In this experiment, we study the sensitivity of T-RACKs to the value of the preset RACK RTO. For this purpose, we repeat the last simulation by varying the value of the RTT multiplicative factor α in the set [1, 5, 10, 50, 100]. We report the average FCT of small flows and all flows in each case for DropTail, RED, and DCTCP in Figure 11 for various loads. From the figure, we can see that the FCT is greatly affected by the choice of parameter α. Small values of α (i.e., 1 and 5) show a relatively large FCT compared to the RTT, which indicates that they tend to cause too many spurious retransmissions that 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 in the three figures for all loads a minimum FCT is achieved at or near a RACK RTO of 10 RTTs.

V. LINUX KERNEL IMPLEMENTATION

In this section, we study the performance of T-RACKs implementation as a loadable Linux kernel module using synthetic workloads reproduced from the statistics of workloads found in production data-centers [6, 9]. T-RACKs is a shim-layer between the VMs (or TCP/IP stack) and the hypervisor (or link-layer). We use the NetFilter framework [25], 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 the guest OS. 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 calculated via Jenkins hash (JHash) [26]. Then, TCP headers are examined, and the proper course of action is taken based on the flag bits (i.e., SYN-ACK, FIN, or ACK) following the logic in Algorithm 1. Unlike SNOOP [16], 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 need the fine-grained high-resolution timers in the microsecond time-scale, therefore the native OS *Jiffies*-based timer is used. T-RACKs uses a single timer for all flows to handle per-flow RTO events. These
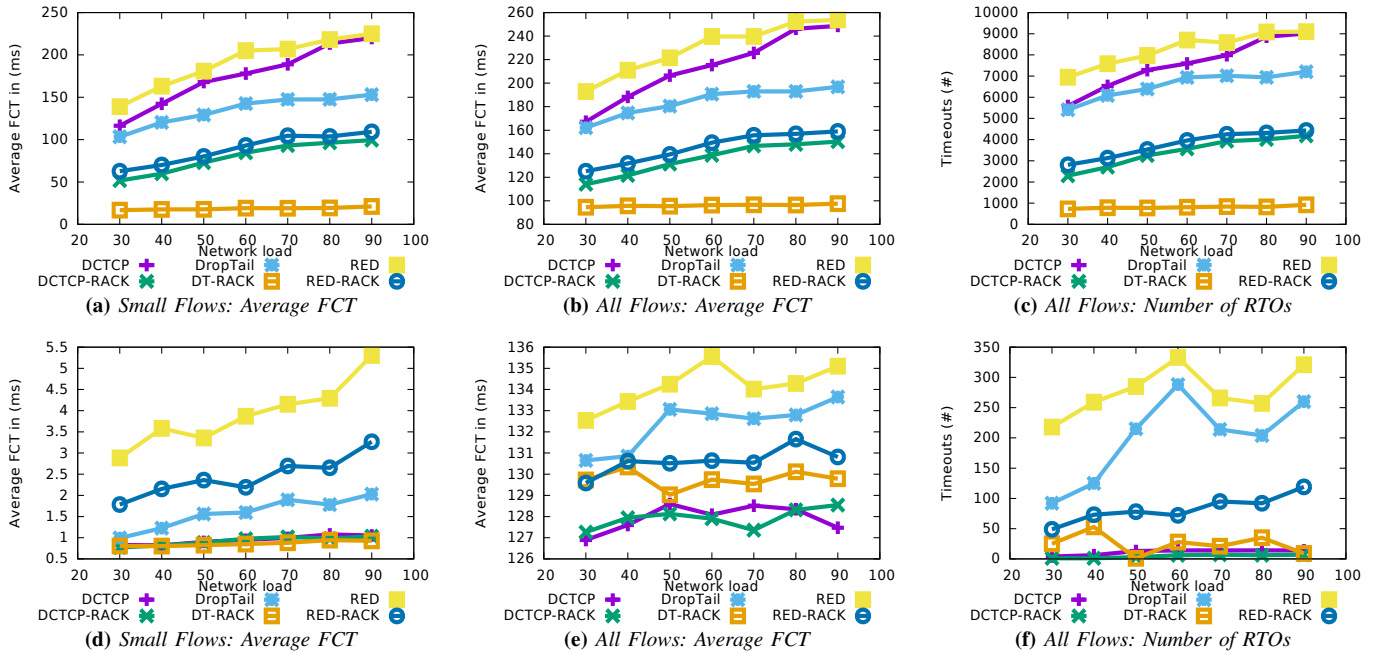
(a) *Small Flows: Average FCT*

(b) *All Flows: Average FCT*

(c) *All Flows: Number of RTOs*

(d) *Small Flows: Average FCT*

(e) *All Flows: Average FCT*

(f) *All Flows: Number of RTOs*

Figure 10: *Performance with network load in (30%, 90%) for workload 1 (top) and workload 2 (bottom)*

(a) *Small Flows: DropTail*

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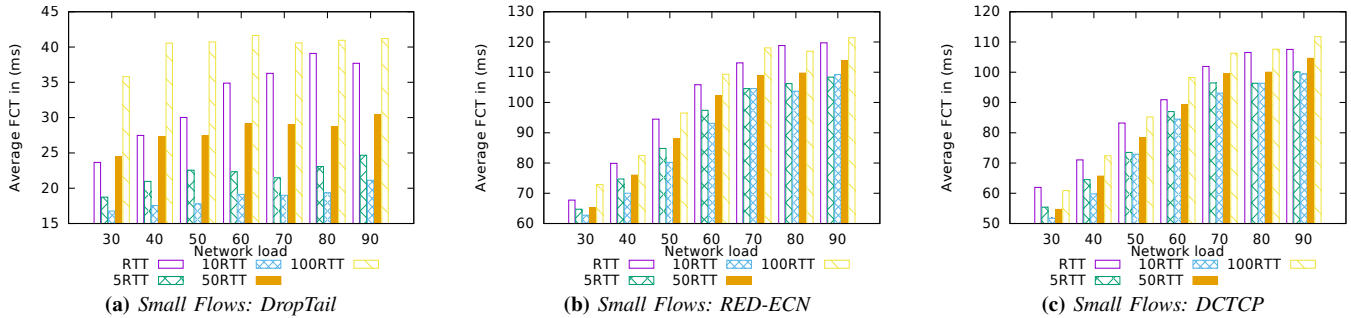
(c) *Small Flows: DCTCP*

Figure 11: *Sensitivity to alpha variation in the range 1 RTT to 100 RTTs for Websearch scenario with DropTail AQM.*

design choices make T-RACKs lightweight and help reduce the server overhead.

Using 14 data-center servers equipped each with 6 NICs, we built a small-scale testbed consisting of 84 virtual servers, each virtual server is assigned a dedicated physical NIC. The servers are interconnected via four non-blocking leaf switches and one spine switch. The testbed is organized into four 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 stated, T-RACKs runs with the default settings (i.e., The RTO and threshold γ of T-RACKs is set to 4 ms and 100 KB, respectively). The RTO of 4 ms is a reasonable 16 times (i.e., ≥ 10) the average RTT of $\approx 250\mu s$ without queuing. We use our custom-built traffic generator to run the experiments with realistic traffic workloads. The traffic generator generates common workloads described in the literature (e.g., industrial-like Websearch [6], Datamining [9] or institutional-like University and Private DC [27]). In addition, we have deployed the iperf program [28] to emulate large background traffic (e.g., VM migrations, backups) in some scenarios. We use different scenarios to reproduce one-to-all

and all-to-all flows with or without background traffic. In the one-to-all scenarios, randomly chosen clients in one rack send random requests to any of all the servers in the data-center. While in the all-to-all scenario, all clients in the data-center send requests to randomly picked servers out of all the servers in the data-center. If background traffic is introduced, we run large iperf flows from all clients to all servers to evaluate T-RACKs under sudden and persistent network load spikes. Like before, we classify flows of size $\leq 100KB$ as small, $[100KB$-$10MB]$ as medium, and $\geq 10MB$ as large.

A. Experimental Results and Discussion

One-to-all Scenario without Background Traffic: we report here the average FCT for small and all flows as well as the number of small flows that missed a deadline of 200ms.

The traffic generator is deployed on every client running on an end-host in the data-center and is set to randomly initiate 1000 requests to randomly chosen servers on any of the other racks. In the Websearch workload, Figures 12a, 12b and 12c show the average FCT and missed deadlines for small flows and the average FCT for all flows, respectively. While, Figures 12d, 12e, and 12f, show the average FCT for small flows in the Datamining, Educational, and private DC workloads,
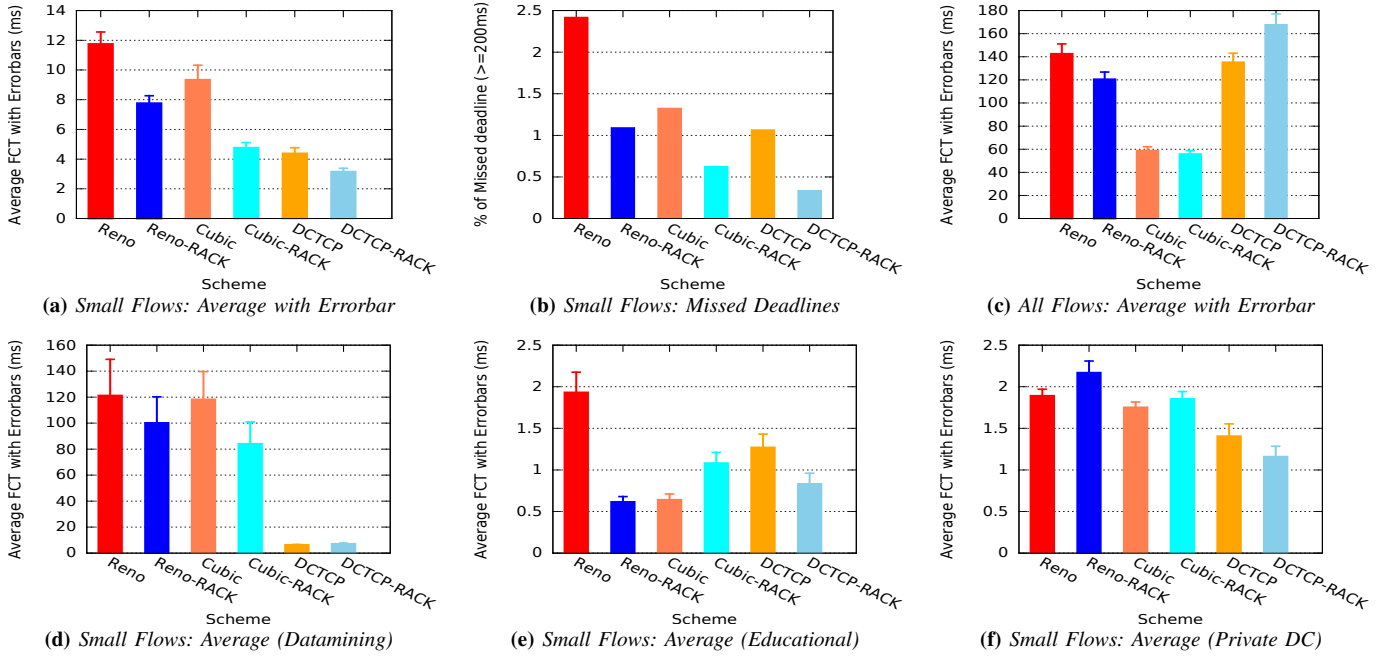
(a) *Small Flows: Average with Errorbar*

(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 12: *Performance metrics of one-to-all scenario without background traffic for (a-c) Websearch, (d) Datamining, (e) Educational and (f) Private DC workloads, respectively.*
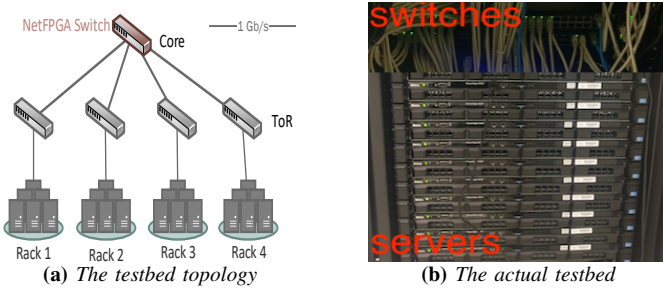

(a) *The testbed topology*

(b) *The actual testbed*

Figure 13: *Testbed setup of T-RACKS in small-scale cluster*

respectively. From these figures, we make the following observations: *i)* for all workloads, T-RACKS helps small flows regardless of the TCP version, on both the average FCT and its variation, as indicated by the error bars. 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 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 missed deadlines in Websearch. The average FCT, in certain cases of Educational and Private DC workloads, shows a negligible increase of FCT with T-RACKS. In these workloads, the network load is very light (as shown by the small FCT without T-RACKS), and hence the added overhead of deploying T-RACKS surpasses its performance gains for these light workloads. *ii)* for Websearch workload, T-RACKS reduces the missed deadlines for small flows by $\approx (55\%, 53\%, 35\%)$ for Reno, Cubic, and DCTCP, respectively. *iii)* T-RACKS slightly improves the overall average FCT. This can be attributed to the fact that small flows are finishing their transmission quicker, leaving some additional bandwidth

for medium and large flows. The improvement shown equals $\approx (16\%, 5\%)$ for Reno and Cubic, respectively. In Figure 12c, DCTCP with T-RACKS, shows a slight increase in average FCT of all flow types for Websearch workload which has many flows of medium size compared to other workload (Figure 9). While DCTCP is designed for improving FCT of small flows in DC environments, T-RACKS is designed to curb RTO events to improve the FCT of small flows whose transmitted volumes do not exceed the threshold γ. Hence, T-RACKS might be adding a slight overhead due to the need to maintain the flow table information for the medium and/or large flows. This overhead may be mitigated by simply skipping state maintenance for flows that exceed the threshold and become long-lived. Moreover, the overhead for TCP variants, which are designed for Internet (e.g., Reno and Cubic), is nearly negligible relative to the large FCT of their long-lived flows. **One-to-all Scenario with Background Traffic:** to put T-RACKS under true stress, we run the same one-to-all scenario with all-to-all background traffic. Figure 14a, Figure 14b and Figure 14c show the average FCT and missed deadlines for small flows as well as the average FCT for all flows for Websearch and Figure 14d, Figure 14e and Figure 14f show the average FCT for short flows for data mining, educational, and private DC workloads, respectively. We observe the following: *i)* T-RACKS can improve the average FCT of small flows for all workloads regardless of the TCP congestion control in use. As shown in the figures, 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 is better 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,
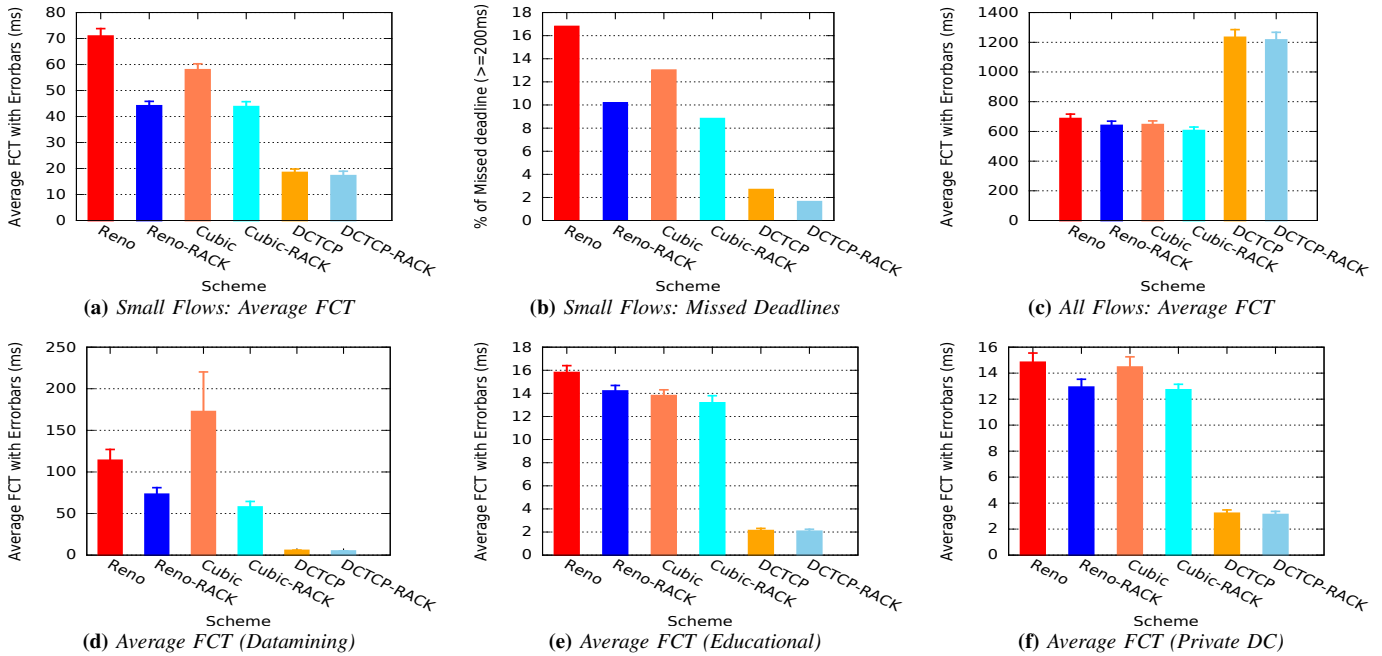
(a) *Small Flows: Average FCT*

(b) *Small Flows: Missed Deadlines*

(c) *All Flows: Average FCT*

(d) *Average FCT (Datamining)*

(e) *Average FCT (Educational)*

(f) *Average FCT (Private DC)*

Figure 14: *Performance metrics of one-to-all scenario with background traffic for (a-c) Websearch, (d) Datamining, (e) Educational and (f) Private DC workloads, respectively.*

and DCTCP, respectively. *iii)* T-RACKs still improves the overall average FCT by $\approx (7\%, 5\%, 2\%)$ 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 each to any of all the servers in the data-center. Figure 15a, Figure 15b, Figure 15c and Figure 15d show the average FCT for short flows in Websearch, Datamining, Educational, Private workloads, respectively. The network load is considerably higher than the previous cases, given the more complex nature of this all-to-all traffic. We can still see here that T-RACKs can deliver significant improvements of up to 71% in the FCT for all workloads.

In summary, the experimental results show the performance gains achieved by T-RACKs, especially for small flows, that constitute the lion's share in data-centers traffic, with minimal to no impact on the performance of larger flows. In particular:

- T-RACKs reduces the average and variance of FCT and missed deadlines of small flows.
- T-RACKs can maintain its gains even if bandwidth-greedy large flows hog the network.
- T-RACKs efficiently handles various workloads, regardless of the variant of TCP congestion controller.
- T-RACKs fulfilled its requirements with no assumptions about nor any modifications to in-network hardware nor the TCP/IP stack of the guest VMs.

VI. RELATED WORK

Several 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, 29–33]. Other works [34–44] analyzed the nature of incast events and
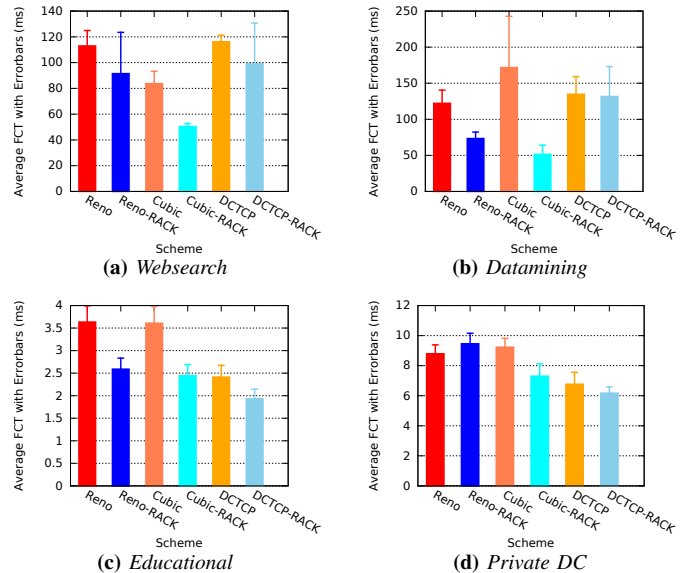

(a) *Websearch*

(b) *Datamining*

(c) *Educational*

(d) *Private DC*

Figure 15: *The average FCT of small flows in the all-to-all scenario for (a) Websearch, (b) Datamining, (c) Educational and (d) Private DC workloads, respectively.*

packet drops in data-centers. They also found that severe incast occurrences could lead to throughput collapse and longer FCT. They show in particular that throughput collapse and increased FCT are to be attributed to the data-center ill-suited timeout mechanism. For example, [12] showed that frequent timeouts could harm the performance of latency-sensitive applications. Numerous solutions have been proposed. These fall into one of four fundamental categories. The first mitigate the consequence of the long waiting times due to RTO, by reducing the default RTO_{min} to the 100 μs - 2 ms [12]. While

very useful, this approach affects the sending rates of TCP by forcing it to cut CWND to 1; it relies on a static RTO_{min} value, which can be ineffective in heterogeneous networks; and it imposes modifications to TCP stack on tenant's VM. Our approach is fundamentally different in its enforcement of RTO_{min} which allows for customized handling of Internet and data-centers flows. Therefore, T-RACKs allows flows to have different RTO_{min} which is easily imposed by the flow tables.

The second approach aims at controlling queue build-up at the switches by relying on ECN marks to limit the sending rate of the servers [6, 8], or by controlling the congestion window [38] or receiver window [7, 47–49] of TCP flows. Similar approaches deployed global traffic scheduler [50–54] or tacked fine-grained sub-microsecond updates in RTT to detect congestion [22]. All of these works achieved their goals and have shown they could reduce the 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 parameters or sometimes require application-side information. They also increase CPU utilization of the end hosts. [22] is sensitive to traffic variations in the backward path. [55] is a new congestion control for inter-DC traffic which is based on characterizing the band-width and RTT of the bottleneck path. While effective for high bandwidth-delay product (BDP), its minutes-level measurements and the aggressive start can exacerbate the problems in low BDP networks of data-centers.

The third approach is to achieve efficient sharing of network resources or enforce flow admission control to reduce TimeOut probability [29, 56–58]. [58] has proposed ARS, a cross-layer system that can dynamically adjust the number of active TCP flows by batching application requests based on 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 utilizing fast retransmit rather than waiting for a long timeout. For instance, TCP-PLATO [29] 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 the timeout. Even though TCP-PLATO is effective in reducing timeouts, 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 the sender and receiver.

Similar to DCTCP, DCQCN [59, 60] and HPCC [61] was proposed as an end-to-end congestion control scheme implemented in custom NICs designed for RDMA over Converged Ethernet (RoCE). Both DCQCN, and HPCC applies adaptive rate control at the link-layer to throttle large flows relying on Priority-based Flow Control (PFC) and RED-ECN marking, and In-Network Telemetry (INT) information, respectively. DCQCN, not only relies on PFC, which adds to network overhead, it introduces the extra cost of the explicit ECN Notification Packets between the end-points. HPCC requires programmable NICs and relies on the timely availability of the INT information which not only increases the packet size by 42 bytes for each hop in the path but also is subject to congestion, and contention with other traffic in the network.

More recent approaches have also identified the importance of the timeout problem in data-center environments [62–64]. For instance, the authors in [62] proposed injects high-priority packet after each window worth of packets. They infer the network congestion by checking the sequences of the received ACKs of high/low priorities. Consequently, they adjust the sending window to reduce buffer occupancy and early detect losses. However, this not only requires setting-up priority-queues (if available) in the switches but also imposes extra processing and communication overhead (esp., the congestion window consists of typically few packets in data-centers).

VII. CONCLUSION AND FUTURE WORK

In this paper, we studied packet losses and the impact of various recovery methods on flow performance in data centers. We then proposed T-RACKs, an efficient cross-layer approach for timely recovery from losses. T-RACKs improves the flow completion time 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 the virtual machines and the network hardware. Simulation and experimental results show that the flows completion time is improved by up to an order of magnitude, missed deadlines are reduced considerably, and a high-link utilization is attained. T-RACKs is shown to be lightweight and practical due to its minimal footprint on end-hosts. Also, because it does not change TCP and work to any TCP flavor, T-RACKs is very appropriate for multi-tenant public data-centers. Finally, it would be interesting to investigate the effectiveness of T-RACKs at scale by deploying it in cloud environments such as AWS or Azure.

REFERENCES

[1] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," in *USENIX HotCloud*, 2010.

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

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

[4] A. Phanishayee, E. Krevat, V. Vasudevan, D. G. Andersen, G. R. Ganger, G. a. Gibson, and S. Seshan, "Measurement and analysis of TCP throughput collapse in cluster-based storage systems," in *USENIX FAST*, 2008.

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

[6] 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.

[7] 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.

[8] G. Judd, "Attaining the promise and avoiding the pitfalls of TCP in the datacenter," in *USENIX NSDI*, 2015.

[9] 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 *ACM SIGCOMM*, 2009.

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

[11] A. M. Abdelmoniem and B. Bensaou, "Curbing Timeouts for TCP-Incast in Data Centers via A Cross-Layer Faster Recovery Mechanism," in *Proceedings - IEEE INFOCOM*, 2018.

[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] M. Mathis, J. Semke, J. Mahdavi, and T. Ott, "The macroscopic behavior of the TCP congestion avoidance algorithm," *ACM SIGCOMM Computer Communication Review*, 1997.

[14] M. H. Jim Keniston, Prasanna S Panchamukhi, "Kernel probes (kprobe)," https://www.kernel.org/doc/Documentation/kprobes.txt.

[15] bootlin.com, "Elixir Cross Referencer," https://elixir.bootlin.com/linux/latest/source.

[16] 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.

[17] B. Cronkite-Ratcliff, A. Bergman, S. Vargaftik, M. Ravi, N. McKeown, I. Abraham, and I. Keslassy, "Virtualized Congestion Control," in *ACM SIGCOMM*, 2016.

[18] 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.

[19] M. Allman and V. Paxson, "On Estimating End-to-end Network Path Properties," *SIGCOMM CCR*, vol. 29, pp. 263–274, 1999.

[20] Y. Zhang and N. Duffield, "On the Constancy of Internet Path Properties," in *ACM IMC*, 2001.

[21] 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 *ACM SIGCOMM*, 2015.

[22] 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 *ACM SIGCOMM*, 2015.

[23] 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.

[24] NS2, "The network simulator ns-2 project," http://www.isi.edu/nsnam/ns.

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

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

[27] T. Benson, A. Akella, and D. A. Maltz, "Network Traffic Characteristics of Data Centers in the Wild," in *ACM IMC*, 2010.

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

[29] 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, 2014.

[30] A. M. Abdelmoniem and B. Bensaou, "Hysteresis-based Active Queue Management for TCP Traffic in Data Centers," in *IEEE INFOCOM*, 2019.

[31] A. M. Abdelmoniem, H. Susanto, and B. Bensaou, "Taming Latency in Data centers via Active Congestion-Probing," in *IEEE ICDCS*, 2019.

[32] A. M. Abdelmoniem, B. Bensaou, and A. J. Abu, "Mitigating incast-tcp congestion in data centers with sdn," *Annals of Telecommunications*, vol. 73, no. 3, 2018.

[33] A. M. Abdelmoniem, H. Susanto, and B. Bensaou, "Reducing latency in multi-tenant data centers via cautious congestion watch," in *International Conference on Parallel Processing (ICPP)*, 2020.

[34] Y. Chen, R. Griffith, J. Liu, R. H. Katz, and A. D. Joseph, "Understanding TCP incast throughput collapse in datacenter networks," in *1st ACM workshop on Research on enterprise networking - WREN '09*, 2009.

[35] J. Zhang, F. Ren, and C. Lin, "Modeling and understanding TCP incast in data center networks," in *IEEE INFOCOM*, 2011.

[36] A. S.-W. Tam, K. Xi, Y. Xu, and H. J. Chao, "Preventing TCP Incast Throughput Collapse at the Initiation, Continuation, and Termination," in *International Workshop on Quality of Service*, 2012.

[37] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz, "DeTail : Reducing the Flow Completion Time Tail in Datacenter Networks," in *ACM SIGCOMM*, 2012.

[38] Jiao Zhang, Fengyuan Ren, Li Tang, and Chuang Lin, "Taming TCP incast throughput collapse in data center networks," in *IEEE ICNP*, 2013.

[39] S. M. Irteza, A. Ahmed, S. Farrukh, B. N. Memon, and I. A. Qazi, "On the coexistence of transport protocols in data centers," in *IEEE ICC*, 2014.

[40] W. Chen, F. Ren, J. Xie, C. Lin, K. Yin, and F. Baker, "Comprehensive understanding of TCP Incast problem," in *IEEE Conference on Computer Communications (INFOCOM)*, 2015.

[41] J. Zhang, F. Ren, L. Tang, and C. Lin, "Modeling and Solving TCP Incast Problem in Data Center Networks," *IEEE TPDS*, vol. 26, no. 2, pp. 478–491, 2015.

[42] A. M. Abdelmoniem, B. Bensaou, and A. J. Abu, "HyGenICC: Hypervisor-based Generic IP Congestion Control for Virtualized Data Centers," in *IEEE ICC*, 2016.

[43] A. M. Abdelmoniem and B. Bensaou, "Enforcing Transport-Agnostic Congestion Control via SDN in Data Centers," in *IEEE Local Computer Networks*, 2017.

[44] A. M. Abdelmoniem, B. Bensaou, and V. Barsoum, "Incastguard: An efficient tcp-incast mitigation mechanism for cloud networks," in *IEEE GLOBECOM*, 2018.

[45] A. J. Abu, B. Bensaou, and A. M. Abdelmoniem, "A Markov Model of CCN Pending Interest Table Occupancy with Interest Timeout and Retries," in *IEEE ICC*, 2016.

[46] A. J. Abu, B. Bensaou, and A. M. Abdelmoniem, "Leveraging the Pending Interest Table Occupancy for Congestion Control in CCN," in *IEEE Local Computer Networks (LCN)*, 2016.

[47] A. M. Abdelmoniem and B. Bensaou, "Reconciling Mice and Elephants in Data Center Networks," in *IEEE CLOUDNET*, 2015.

[48] A. M. Abdelmoniem and B. Bensaou, "Incast-Aware Switch-Assisted TCP Congestion Control for Data Centers," in *IEEE GLOBECOM*, 2015.

[49] A. M. Abdelmoniem, B. Bensaou, and A. J. Abu, "SICC: SDN-based Incast Congestion Control for Data Centers," in *IEEE ICC*, 2017.

[50] M. Alizadeh, A. Kabbani, T. Edsall, and B. Prabhakar, "Less is More : Trading a little Bandwidth for Ultra-Low Latency in the Data Center," in *USENIX NSDI*, 2012.

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

[52] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang, "Information-agnostic Flow Scheduling for Commodity Data Centers," in *USENIX NSDI*, 2015.

[53] H. Susanto, A. M. Abdelmoniem, H. Jin, and B. Bensaou, "Creek: Inter many-to-many coflows scheduling for datacenter networks," in *IEEE ICC*, 2019.

[54] H. Susanto, B. L. Ahmed M. Abdelmoniem, Honggang Zhang, and D. Towsley, "A Near Optimal Multi-Faced Job Scheduler for Datacenter Workloads," in *IEEE ICDCS*, 2019.

[55] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, "Bbr: Congestion-based congestion control," *Commun. ACM*, vol. 60, no. 2, 2017.

[56] T. Benson, A. Akella, A. Shaikh, and S. Sahu, "Cloudnaas: A cloud networking platform for enterprise applications," in *ACM Symposium on Cloud Computing (SoCC)*, 2011.

[57] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica, "FairCloud: Sharing the Network in Cloud Computing," *ACM SIGCOMM CCR*, vol. 42, pp. 187–198, 2012.

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

[59] M. Alizadeh, B. Atikoglu, A. Kabbani, A. Lakshmikantha, R. Pan, B. Prabhakar, and M. Seaman, "Data Center transport mechanisms: Congestion control theory and IEEE standardization," in *46th Annual Allerton Conference on Communication, Control, and Computing*, 2008.

[60] M. Alizadeh, A. Kabbani, B. Atikoglu, and B. Prabhakar, "Stability analysis of QCN," *ACM SIGMETRICS Performance Evaluation Review*, vol. 39, no. 1, p. 49, 2011.

[61] Y. Li, R. Miao, H. H. Liu, Y. Zhuang, F. Feng, L. Tang, Z. Cao, M. Zhang, F. Kelly, M. Alizadeh, and M. Yu, "HPCC: High Precision Congestion Control," in *ACM SIGCOMM*, 2019.

[62] C. Ruan, J. Wang, W. Jiang, G. Min, and Y. Pan, "PTCP: A priority-based transport control protocol for timeout mitigation in commodity data center," *Future Generation Computer Systems*, vol. 102, pp. 619 – 632, 2020.

[63] Y. Xu, S. Shukla, Z. Guo, S. Liu, A. S. . Tam, K. Xi, and H. J. Chao, "RAPID: Avoiding TCP Incast Throughput Collapse in Public Clouds With Intelligent Packet Discarding," *IEEE JSAC*, vol. 37, no. 8, pp. 1911–1923, 2019.

[64] J. Zhuang, X. Jiang, G. Jin, J. Zhu, and H. Chen, "PTCP: A Priority-Driven Congestion Control Algorithm to Tame TCP Incast in Data Centers," *IEEE Access*, vol. 7, 2019.

Ahmed M. Abdelmoniem (Member ACM, IEEE, USENIX) received his Ph.D. in Computer Science and Engineering from Hong Kong University of Science and Technology, Hong Kong in 2017 and his B.Sc. and M.Sc. degree in Computer Science from Assiut University, Assiut, Egypt, in 2007 and 2012 respectively. He is an Assistant Professor at Assuit University, Egypt and a Research Scientist at KAUST, Saudi Arabia. Formally, he held the position of a Senior Researcher with Huawei's Future Networks Lab, Hong Kong. He was awarded the prestigious Hong Kong PhD Fellowship from the Research Grant Council (RGC) of Hong Kong in 2013 to pursue his PhD. He has published numerous papers in top venues and journals in the areas of wireless networks, traffic engineering and congestion control. His current research interests are in the areas of optimizing systems supporting distributed machine learning and cloud/data-center networking with emphasis on performance, practicality, and scalability.

Brahim Bensaou (Senior Member, ACM, IEEE) received an Engineering Degree in Computer Science (with distinction) from the University of Science and Technology Houari Boumediene of Algiers, Algeria in 1982, and a DEA degree from University Paris XI in Computer Science in 1988. He earned his Ph.D. in Computer Science from the University Paris VI in 1993. He is now a tenured faculty member in the CSE department of the HKUST. Formerly, he held positions of research assistant at France Telecom Research labs, Research Associate at HKUST, and Senior Member of Technical Staff at the National R&D Centre for Wireless Communications in Singapore where he lead the strategic research group on wireless networking. His research is in general centered around the Internet communication and its performance (e.g., Congestion Control, Information-centric Networks, Energy efficiency, Performance evaluation). As well as, Wireless communications and mobile networks. He published more than 130 research papers in prominent conferences and journals, received numerous research grants, supervised and graduated more than 20 postgraduate research theses including both PhD and Master and holds 3 granted US patents, one of which is licensed.