End-host Timely TCP Loss Recovery via ACK Retransmission in Data Centres.

Ahmed M. Abdelmoniem and Brahim Bensaou
Department of Computer Science and Engineering
The Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
{amas, brahim}@cse.ust.hk

June 28, 2017

In this report, we have studies various switch-based schemes to solve certain issues that TCP applications face in public cloud networks. In this report, we study the problem of Retransmission Timeout (RTO) inadequacy for TCP in data centers. To this end, we empirically, in a small data center, analyze the effects of packet losses on various types of flows. Then, we highlight the non-uniform effects of packet losses on the Flow Completion Time (FCT) of short-lived flows. In particular, we show that packet losses which occur at the tail-end of short-lived flows and/or bursty losses that span a large fraction of a small congestion window are frequent in data center networks. These losses in most cases result in slow loss recovery after waiting for a long 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 that was set by default to hundreds of milliseconds to meet the Internet scale. Hence, we propose the so-called Timely Retransmitted ACKs (**T-RACKs** which is pronounced T-Rex), a recovery mechanism for data centers, implemented as a shim layer between the VMs layer and the hypervisor. T-RACKs aims to bridge the gap between TCP RTO and the actual Round-Trip Time (RTT) as experienced in the data center. Simulation and experimental results show considerable improvements in the FCT distribution and missed deadlines without any need to modify the TCP implementation or configurations in the guest VMs. T-RACKs requires no special hardware features which makes it appealing

for immediate deployment in production data centers.

For the purposes of reproducibility and openness, we make the code and scripts of our implementations, simulations, and experiments available online at **`http://github.com/ahmedcs/T-RACKs`**.

1 Introduction

In Part **??** of this thesis, we have shown that the quality of the application results is correlated not only with the average latency but also with the latency of the tail-end result (e.g., the 90^{th}-up percentile of the FCT) which can be from 2 to 4 orders of magnitude worse than the median or even the average. In a recent study [20], it was shown that CPU resources are often the bottleneck in private data centers and hence they adopt solutions that invoke task admission control and scheduling techniques. On the other hand, public data centers are equipped with abundant computing resources and usually the network is the bottleneck. They also tend to apply high over-subscription ratios in the network, greatly impacting on the network latency [29]. Hence, most Internet-scale applications deployed on public clouds (e.g., Microsoft Azure or Amazon EC2) suffer from latency issues.

In data centers, the transfer of data involves various delays that contribute to the latency. These delays include processing, transmission, propagation and queueing delays and in certain cases the delay of waiting for RTO. Processing, transmission and propagation delays tend to be small and they are either constant or variable with very small variations. Hence, they are almost negligible and are not blamed for increased network latencies. Therefore, the increased latency is typically a result of the inflated in-network queueing delays or the long waiting for RTO. Queueing delays typically increase when the network is congested and the buffers start to fill up. The waiting in the queue can add a significant amount of delay to the flow completion time. However, thanks to the fast link speeds employed in data centers, these delays range between 100s microseconds to a few milliseconds. On the other hand, waiting for the RTO can greatly impact on the performance of any short-lived flow and the FCT is bloated up to values that render the transfer of small flow useless. This is because RTOs are pre-fixed with lower bound values that fit the wide-area Internet communications which are in orders of 100s of milliseconds. It is evident that waiting for 100s of milliseconds for intra-datacenter communications is a devastating situation for small flows if they experience losses not recoverable by fast retransmit.

In Part **??**, we have explored techniques to control the queue occupancy at low levels, techniques to alleviate complex events such as incast and techniques to stretch out the short TCP cycles in data centers. Even though, these techniques were shown to be effective, they possess a serious drawbacks which is that they are switch-based systems and they may see slow adoption in production data centers. In addition and more importantly, they have not addressed the cases when flows experience Non-Recoverable Losses (NRL) (i.e., the flow have to wait for long RTO). In this report, we try to fill this gap and present a hypervisor-based system to handle NRL in a timely manner.

In the remainder of this report, we first give the background and discuss in Section 2 how waiting for RTO can dramatically affect the performance of time-sensitive TCP flows. We support our claims with results from an empirical study in a small testbed. Our proposed methodology and the T-RACKs system are presented and discussed in Section 3. In Section 4, we discuss our simulation results in detail. Then, in Section 5, we discuss the implementation details of T-RACKs and show experimental results from its deployment in the testbed. We discuss important work related to the report in Section 6. Finally, we conclude the report in Section 7.

2 Background and Problem Statement

Major application providers (e.g., Microsoft, Facebook and Google) use dedicated well-structured data centers to deploy their time-sensitive applications. Nevertheless, due to the predominance of many-to-one (or many-to-many) communication patterns in data centers, TCP-incast network congestion is inevitable and still results in many RTO events [28, 18, 2, 35, 17]. 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 guest VM. This bloats the perceived RTT from the microsecond time scale to the millisecond scale. In the following, we discuss more about TCP retransmission timeout [33] and the effect of virtualization on TCP's RTT measurements [34].

2.1 TCP Retransmission Timeout

TCP is a reliable transport protocol and to ensure reliability it employs an end-to-end acknowledgement system. Whenever the data is sent by the sender, the receiver must acknowledge the receipt of the data if the data packet has no errors. In an ideal case, data and acknowledgment packets are successfully delivered, however there are unfortunate cases where the packets get corrupted/lost. To fulfill the reliability requirement, the TCP sender sets a timer when the data is transmitted. If the acknowledgement of the data is not received before timeout expiry, then TCP assumes the segment is lost and retransmits the lost segment.

A critical part of any TCP implementation is the setting of RTO and the employed retransmission strategy. Hence, the important questions to be answered are: Q_1) how is the RTO period determined?; Q_2) how are consecutive timeouts handled?; Q_3) and which packets are retransmitted? The answers to these questions are as follows:

1. We find that the typical TCP Request For Comments (RFC) and implementations measure the RTT of TCP segments. Then, TCP employs a smoothed average of these measurements to estimate a reasonable RTO value for each segment to be retransmitted. However, most implementation also put a lower bound on the RTO value (i.e., Minimum RTO) which are in orders of 100s milliseconds (1 second in RFC [27]). This minimum RTO is typically hard-coded (e.g., Linux Kernel) and is set to match delays seen on the Internet.
2. To handle the case of the consecutive timeouts for the same data packet, TCP adopts an exponential backoff strategy. Exponential backoff gradually doubles the RTO value for each retransmission of the retransmitted data packet. For this purpose, TCP upper bounds the RTO (i.e., Maximum RTO) which are in orders of seconds. This maximum RTO is also hard-coded (e.g., 64 seconds in the Linux Kernel).
3. When TCP retransmits packets, it either adopts a Go-Back-n or selective repeat strategy. Typical implementations follow the Go-Back-n strategy in which all packets after and including the lost one are retransmitted. Selective repeat transmits only the lost packet; however this function requires to be negotiated by the end-points and a special TCP option are needed in the TCP header.

In Section 2, we will perform an empirical study on packet loss measurements from a small data center. Our findings show that the RTO is a frequent event in typical data center workloads. In addition, the performance of small and large flows are equally impacted by the extra waiting time for RTO.

2.2 RTT Inflation by Virtualization

Most CSPs use the virtualization mechanism to share the resources among users in a flexible and cost-effective manner. For instance, Amazon EC2 uses Xen virtualization [19] to run multiple VMs on the same physical machine. Typically, VMs share CPU, I/O devices (e.g., disk and NIC) with other VMs residing on the same machine. It is not surprising that the task of virtualizing the physical resources has a great impact on the performance of computation and communication in virtualized cloud environments. A few measurement studies have been conducted to observe the impact of virtualization technologies on the Quality of Service (QoS) of cloud applications.

The work in [34] was one of the first studies to report their findings. They found that VMs of small type typically get only a 40-50% share of the processor and they see very unstable TCP/UDP throughput. More surprisingly, the TCP/UDP throughput seen by many applications can rapidly fluctuate between 0-1 Gbps within 10s millisecond time granularity. They also observe that even though DCN is not heavily congested, abnormally large packet delay variations among different instance variants are reported. Moreover, the delay fluctuations can be up to 100x times the propagation delay between two end hosts. These large delay variations are a result of the long queuing delay the packets experience at the interface between VM driver domain and the hypervisor. The queue typically builds up when the packets from guest VM accumulate and wait for the hypervisor to move them to the host OS side. This queueing and data movement add up to the software overhead on the end systems and consequently on the overall delay as well.

The performance of applications is not only affected by the delays introduced during data movement from VM space to host OS but also the delays induced by the hypervisor's scheduling which can greatly affect the I/O performance of the VMs. This is the case because these scheduling delays increase the VM's interrupt processing time. To support this fact, we have collected various network measurements including RTT and interrupt-processing overhead (i.e., delay) from our experimental cluster running different types of Hypervisors (e.g., KVM/Qemu and Xen). We perform an empirical study on a small cluster testbed as well as on measurements from Amazon EC2. Our findings is that the context switching among guest VMs adds at least 30ms of delay in a Xen virtualized environment. The performance of small and large flows is equally impacted by the extra delay.

2.3 Empirical Analysis of RTO

To be able to study when and how the timeout happens in high-bandwidth low-delay environments, we have built a customized traffic generator written in C and python. The generator reproduces traffic similar to those reported from commercial and private data centers. The traffic is generated via socket based client/server applications running on the VMs (1 client and 1 server per VM) in the cluster. The client generates flow requests with flow sizes drawn from distributions extracted from [18, 9, 2, 6]. Similarly, the request inter-arrival times are generated from the supplied distribution files. The server is a simple TCP-listener that accepts requests and sends random data (of size that match the client requested size) to the client. In order to coordinate the experiments, we have used the remote XMLRPC server on each machine to start up clients and servers with the supplied experimental configurations.

The traffic generator was augmented with scripts that collect the uplink and downlink utilization of all active NICs on the end-host. To measure the system load and overhead introduced by our probing module as well as T-RACKs, we sampled the system load information from *sysstat* program on all participating end-hosts. Our cluster uses non-blocking Top-of-Rack (ToR) switches and a NetFPGA-card on a PC as our core switch. The traffic generator framework is instrumented to collect queue occupancy and packet drops for each of the 4-ports on the NetFPGA card by sampling the corresponding statistic registers at regular intervals.

To observe the events at the TCP socket level, we have invoked the Linux kernel built-in probing features (jprobe [16]) and have built a kernel module to trace TCP socket events shown in Table 1. The module installs a jprobe object per traced TCP function to call our designated probe function. Probe functions[1]. To log the events of interest, we leverage the */proc* file system. TCP stack Probing potentially adds extra overhead to the data-path processing pipeline. Hence, to lower such overhead, we create one PROC file for arrival events and another one for departure events. The module dumps for each event various TCP socket-level state variables. Table 2 shows a number of these variables, while other variables are omitted for brevity. These variables are used later in our analysis of TCP dynamics[2]

[1]Typically, probe functions is named with "j" prefix added to the name of its Linux counterpart. For example, the probe function of tcp_set_state would be jtcp_set_state

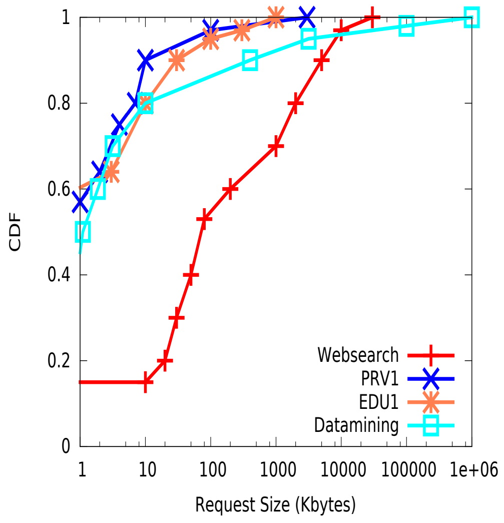
[2]In some case, we used additional information from packet-level traces collected at the link-layer using TCPdump [32].

Traced Event	Linux Kernel Function	Description
Connection establishment and teardown	tcp_set_state	tcp_set_state is invoked upon change of the current TCP socket state. The transitions follows the TCP state machine For instance, sockets are opened when state changes from TCP_SYN_SENT to TCP_ESTABLISHED. Similarly, sockets are closed if state changes from TCP_LAST_ACK or TCP_FIN_SENT to TCP_CLOSE
ACK & dupACK & FRACK	tcp_v4_do_rcv	Upon arrival of TCP segments wether Data or ACK, tcp_v4_do_rcv is invoked to process the segment. dupACKs are identifed if the seq# is equal to tp-¿snd_wnd. FRACKs are differentiated from the normal ACKs (for tracing purpose) by using a different TTL value for when FRACKs are transmitted.
Segment departures	tcp_v4_send_check	Before departure of TCP segments, function tcp_v4_send_check performs certain checks on the segment. We probe this function to detect timeout events based on the transmitted seq#.
Fast Retransmit & RTO	tcp_retransmit_skb	Whenever segments are transmitted by TCP, function tcp_retransmit_skb is called. The type of retransmission can be inferred from the sockets CA_STATE variable. If the value is TCP_CA_LOSS then this is RTO event, otherwise it is fast retransmit event.

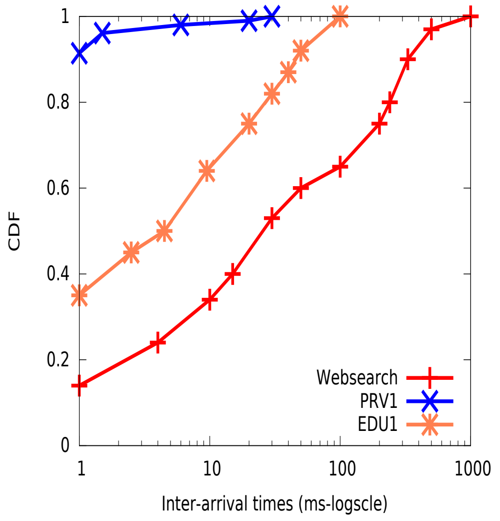
Table 1: *Traced events by our custom TCP socket-level Probe module. Each event is probed within a certain function of Linux TCP/IP stack.*

Socket/Pkt Variable	Description
Event Timestamp	The current timestamp of the event (in nano-second granularity).
Event Type	XMIT, RXMIT, FXMIT, OPEN, CLOSE, RCV, RCVACK, dupACK or FRACK.
4-tuples ID	End-points IP and port numbers for flow ID.
SEQ No	For received and/or (re)transmitted segments.
ACK No	For ACKs and/or dupACKs and/or FRACKs.
Cwnd	The current value of TCP congestion window
Rwnd	The recent value of TCP advertised window
Swnd	The current value of TCP sending window
Flight pkts	The number of packets currently on flight
RTT	Smoothed RTT and variance of measured RTT

Table 2: *TCP Socket level state variables traced by our module.*



(a) *CDF distribution of requested flow sizes*



(b) *CDF distribution of inter-arrival times*

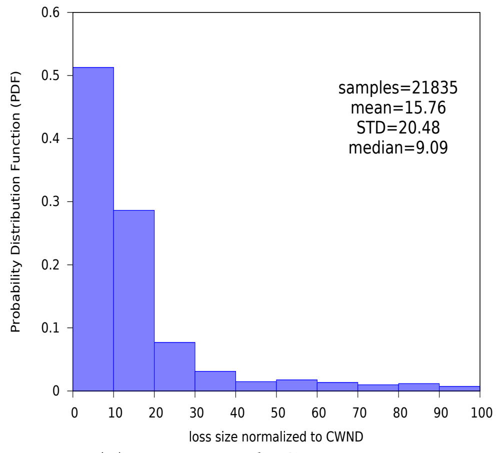
Figure 1: *Flow size and inter-arrival time distributions*

The Traffic Generator (TG) can be configured to produce one of four different realistic workloads including websearch [2], datamining [9], educational [7] and private cluster [7]. Fig 1a shows per server-application response size distribution and Fig 1b shows the inter-arrival times distribution of the aforementioned workloads. Notably, for a websearch workload, the average inter-arrival time is 117 ms and the average response size is 1.44 MBytes. For datamining workloads since the inter-arrival times distribution is not available we have used a Poisson process with mean inter-arrival of 810ms and the average of the response size distribution is 12.66 MBytes. Our testbed cluster uses 28 server-grade end-hosts organized into 4 racks. The racks are connected via 5 switches organized into 4 ToR and 1 Core (NetFPGA) switch. Even though, our cluster does not match the scale of commercial-sized clusters, we believe using realistic workloads allows us to draw meaningful qualitative conclusions from our results and findings that may benefit the design of larger-clusters.

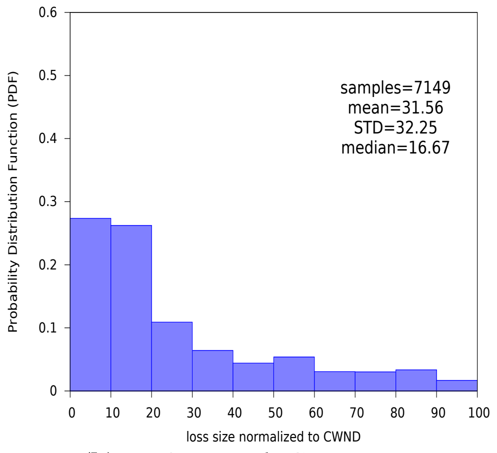
To find why packet losses do not really hurt much elephant flows yet dramatically degrade the performance of short-lived flows, we used our socket-level probe module to collect relevant information shown in Table 2. We used the traffic generator to replicate the websearch workload of thousands of flows. We display the size of each retransmission (i.e., the seq# of the first and last segment in single recovery) with respect to CWND and the position of the first retransmitted segment relative to CWND when it was transmitted (i.e., before loss is detected).

Here we show our results and try to pin-point the actual cause of increased latency in data centers. Fig 2c suggests that fast retransmission is well distributed over the range of packets with positive skewness towards packets at the end of the window. However, Fig 2b clearly shows that this is not the case for timeout retransmissions where the distribution is heavily tailed again with positive skewness towards the few packets at the end of the window. Also, we can see that there are very few RTOs far away from the tail, these reflect sequences of lost packets within the same congestion window. Fig 2d shows the distribution of the position of fast retransmitted packets relative to the size of the congestion window[3]. This points out that losses at the tail of the window occur with higher frequency, however in the case of Fast Retransmit and Recovery (FRR), the ***Cwnd***has to be relatively large to allow for recovery via FRR. Similarly, Fig 2a shows the distribution of the position of packets retransmitted through RTO which clearly shows 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 FR recovery. To put this into context, Fig 3a shows that the segments that have recovered via

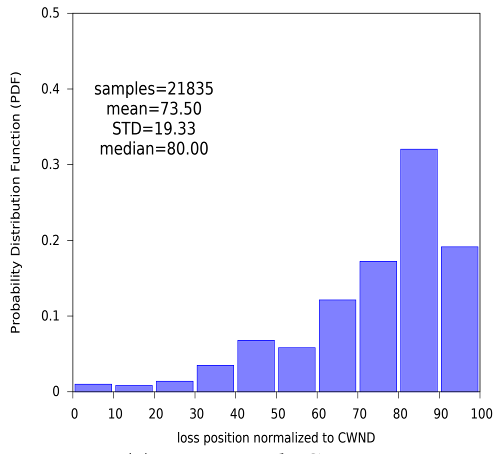
[3]The position here points to the first retransmitted packet if a range of consecutive packets were lost. Each figure shows the aggregate of all servers in the cluster



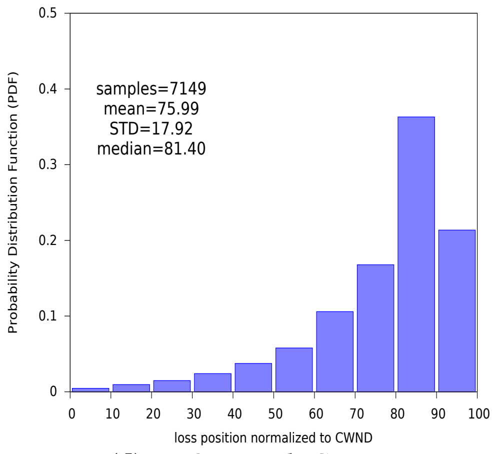
(a) *FR size rel. CWND size*



(b) *RTO size rel. CWND size*



(c) *FR pos rel. CWND*



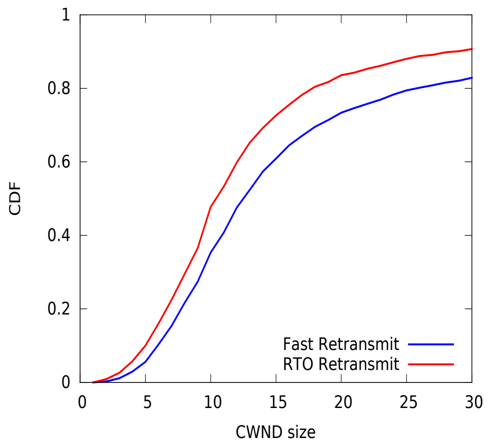
(d) *RTO pos rel. CWND*

Figure 2: *(a-b) shows retransmission size while (c-d) position of the segment relative to CWND*

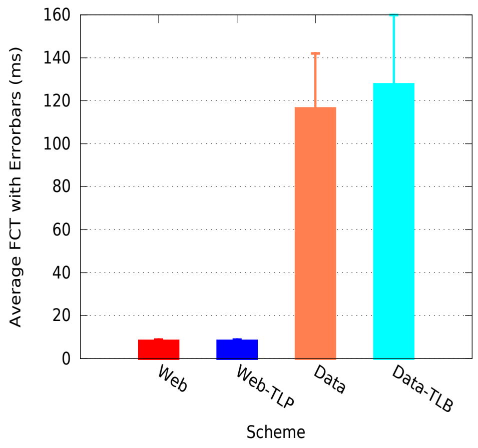
RTO have ***Cwnd*** values that are smaller than the ones that have recovered via FRR.

In Section 6, we present a tail-loss recovery mechanism namely TLP [23] which was proposed as an RFC and was adopted in Linux Kernel. Hence, we also quantify the ineffectiveness of TLP mechanism [23] in recovering tail losses. Fig 3b, shows that the TLP mechanism is not effective and due to its overhead it may even increase the FCT of short flows.

To further understand how RTO would degrade the performance of TCP flows,



(a) *CDF of CWND size*



(b) *FCT of small flows*

Figure 3: *(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*

assume a TCP flow is established, and its current congestion window has the value of w. TCP will start sending w full-sized segments (numbered 1,2,..,w). Assuming Drop-Tail AQM in place at the bottleneck link, and that a packet w is lost and subsequently L packets are lost. Then, TCP surely experiences timeout if the following inequality holds:

$$w-(x+L) \leq \phi, \tag{1}$$

where ϕ is the minimum number of TCP DUPACKs necessary to trigger the FRR mechanism. Equation(1) simply states that if a packet loss is followed by enough losses (not necessarily consecutive) leading to the depletion of the flight-size, there is a high chance of not having sufficient dupACKs to recover via FRR. In data centers the pipeline size is small: typically with 100us RTT, a link of 1 (10) Gbps can accommodate 8.3 (83) packets, respectively. In conjunction with shallow buffered switches, the nominal TCP fair share during TCP-incast barely exceeds one packet per flow, making the condition described by Equation.1 very frequent. This clearly highlights how TCP's performance can be degraded when operating in small windows regimes 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.

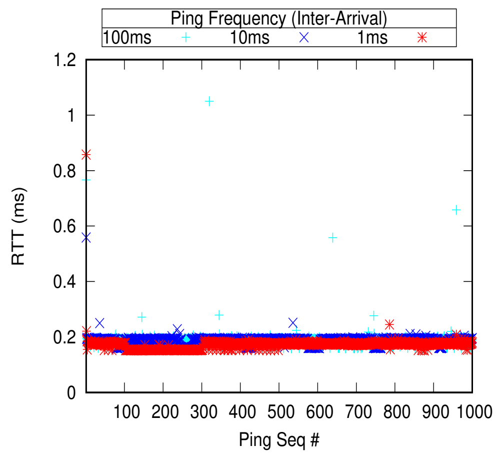
2.4 Empirical Analysis of Virtualization Delays

In this study, we install Xen hypervisor on a small scale testbed environment and collect measurements to quantize the effect of virtualization on applications. Xen hypervisor runs a driver domain (Dom0) which performs I/O forwarding for all guest domains and normally runs on dedicated CPU cores for guaranteed efficiency. I/O virtualization models adopted by Xen are: *i)* Full Virtualization (FV): which allows for an unmodified guest OS to run on processors with virtualization support (i.e, Intel VT and AMD SVM) while the Xen hypervisor leverages QEMU to emulate I/O devices for VMs; and *ii)* Para-Virtualization (PV): which is a modified guest OS that uses a front-back-driver model for the execution of privileged instructions.

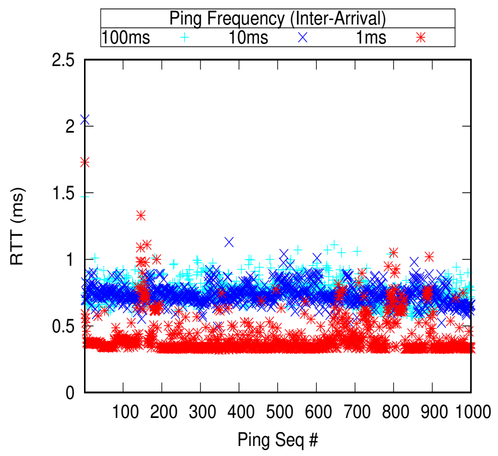
The testbed consists of 7 servers each associated with physical 4-port dedicated 1GB Server-grade Network Interface Card (NIC) and running on 7 high performance Dell PowerEdge R320 machines. The machines are equipped with Intel Xenon E5-2430 6-cores CPU, 32 GBytes of RAM and Intel I350 server-grade 1 Gb/s quad-port NIC. The servers are organized into 4 racks (each rack is a subnet) and connected via 4 non-blocking ToR switches with the NetFPGA switch serving as the core switch. The topology forms a leaf-spine tree where each 7 out of the 28 ports belonging to the same subnet are connected to one of the non-blocking ToR switches through 1 Gb/s links. The servers are installed with Ubuntu Server 14.04 LTS upgraded to kernel version 3.18 which has by default the implementation of Cubic, New Reno and DCTCP [1] congestion control mechanisms.

In the following experiments, we use ICMP-based ping program which can report consecutively the end-to-end RTT values seen by the VM. We collected the measurements for 1000 consecutive ICMP ECHO request-reply with various inter-request time interval of $100, 10, 1$ ms. First, we show the RTT of the physical network between 2 servers which are 4 hops away in our leaf-spine topology. Figure 4a shows that an average RTT of $0.191, 0.188, 0.172$ ms (which was consistent for other servers) for intervals of $100, 10, 1$ ms, respectively. Then the RTT of VM-to-VM (each VM allocated 1 VCPU pinned to 1 CPU core) is shown in Figure 4b where the average RTT jumps to $0.757, 0.733, 0.41$ ms for intervals of $100, 10, 1$ ms, respectively. The overhead added by the extra work of hypervisor moving data between the driver domain (Dom0) and the guest domain leads to increase in the RTT of $\sim 0.2 - 0.55$ ms.

Now we co-allocate the physical CPU core used by the VCPU of the VMs with another CPU-bound VM (CVM), this can be achieved by means of setting the

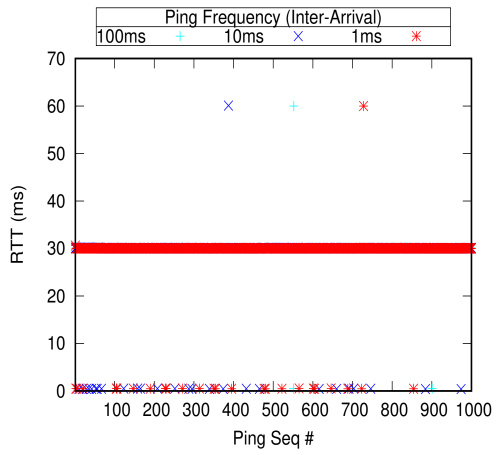


(a) *[PM→PM] no load*

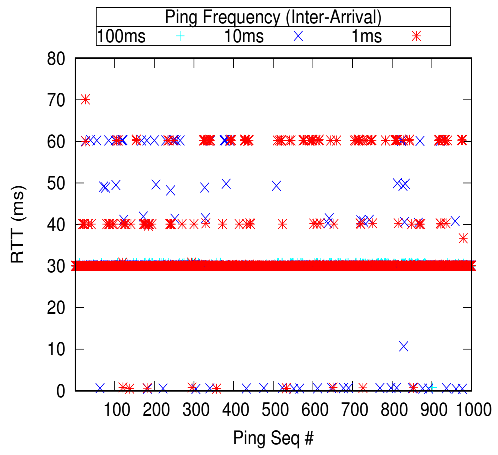


(b) *[VM1 → VM2] (no CVM)*

Figure 4: *Ping RTT between two different PMs and two VMs running on them. The VMs are of type Para-virtualization.*



(a) *VM1 with 1 CVM*



(b) *(VM1&VM2 with 1 CVM)*

Figure 5: *RTT between two VMs running on two PMs. The VMs are co-allocated CPU core with another CVM.*

affinity (i.e., pinning) of each VCPU to a set of CPU cores. This is a typical setting for current IAAS data centers where customers rent out VMs to deploy their own applications or perform certain tasks. We observe a large jump in the RTT of VM-to-VM when the sender is co-allocated CPU cores with only 1 CVM as shown in Figure 5a. The average RTT jumps drastically to $30.010, 29.357, 29.411$ ms for intervals of $100, 10, 1$ ms, respectively. The RTT jumps more frequently above these values when 1 CVM is pinned down to the same CPU core of the receiver VM. This clearly suggests that the extra delay is introduced by the hypervisor scheduling delays which slows down the processing of the interrupts of the guest VMs leading to an increase in the RTT up to values of $\sim$ 30ms. This value is not surprising since Xen hypervisor sets a scheduling time-slice of 30 ms for the guests which will be fully used by any CPU-bound VMs delaying IO-bound VMs. We repeat the experiment but co-allocate 1 CVM with the receiver VM2 as well. Figure 5b shows that the average RTT values jumps frequently to levels above the 30 ms while the average RTT becomes $30.091, 30.687, 33.011$ms for intervals of $100, 10, 1$ms, respectively. The values larger than the 30ms bar indicates that the scheduling delay at the receiver affects the time the VM has to wait to process incoming data and send replies.

3 The Proposed Methodology

Since, most public data centers adopt the so-called IaaS model, the operating system and thus the protocol stack in the VM are under the full control of the tenant and cannot be modified by the cloud service provider. Hence, in this report, we design a cross-layer recovery mechanism to address the shortcomings of TCP in data centers without the need for modifying TCP itself. The design requirements of our framework are:

1. Improve the FCT of latency-sensitive applications (mice).
2. Friendly to throughput-sensitive long-lived (elephant) flows (i.e., do not sensibly degrade their throughput).
3. Compatible with all existing TCP-flavors (i.e., modifications must be in the hypervisors, which are fully under the control of the DCN operator).
4. Simple enough to appeal to cloud service operators for deployment.

With these requirements in mind, we propose T-RACKs system that actively infers packet losses by tracking per-flow TCP ACK numbers and proactively invokes the FRR mechanism of TCP. 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 certain, leading to a significant improvement of recovery times, and hence the FCT of TCP.

To explain our idea and its effectiveness, we first need to learn to distinguish the symptoms (e.g., packet losses and recovery via RTO) from the pathology (e.g., the value of the RTO). More specifically, many schemes proposed in the literature to deal with TCP congestion in data centers mistakenly treat packet losses as the direct culprits behind the FCT degradation and devise complex mechanisms to curb them, treating them as the pathology. Actually by design of TCP, packet losses are inevitable; they are an integral part of the protocol and are used to convey symptoms of congestion to the source. To understand why packet losses result in such long delays in data center networks, we collect measurements from our testbed with realistic traffic-workloads to pinpoint the root causes of such increased latency. Even though, a number of measurement studies [18, 9, 6] have shown varying latencies in data centers environments, we here dive deep into the packet level analysis of the flows to understand TCP behavior and its loss recovery mechanisms. An earlier work [33], based on data-center measurements, found that the timeout mechanism is to blame for the long waiting times and proposed the simple solution of reducing the minRTO value for TCP in data center environments. This approach ultimately solves the problem, reduces FCT of flows and mitigates TCP-incast congestion effects in data centers. However, i) it requires the modification of TCP and ii) there is no single value of the minRTO that fits all environments: a minRTO that works in the data center (e.g., between a web server and the backend database server) will definitely lead to spurious timeouts for customer-facing connections (e.g., the Internet side). A recent RFC [23] proposed the so-called tail loss probe (TLP) which recommends sending TCP probe segments whenever ACKs do not arrive within a short Probe TimeOut (PTO)[4]. In addition to requiring changes to TCP, this approach suffers from two additional problems: i) probe segments may be lost and ii) probe packets may worsen the in-network congestion, especially during TCP-incast.

Next, we will introduce our proposed T-RACKs, discuss its design aspects and highlight how it achieves its goals. Following the previous discussion, the design is based on the following observation: As packet losses are inevitable for the proper operation of TCP, the key to reducing the long latency and jitter is not to try and avoid losses but rather try to avoid long waiting after losses occur.

We then introduce the rationale of our design choices: *1)* knowing that all TCP flavors adopt the fast retransmission mechanism as a way to detect and recover from losses without incurring the expensive timeouts, if one can force the mechanism into action regardless of the nature of loss, the resulting system would be transparent to the TCP protocol in the VM; *2)* TCP relies on a small amount of

[4]PTO is set to minimum of (2 $\times$ srtt, 10ms) if inflight $>$ 1 and to (1.5 $\times$ srtt + worst case delayed ACK (i.e., 200ms)) if inflight==1

dupACKs to activate FRR, however in the majority of cases (esp. for short-flows) there aren't enough packets in the pipe to trigger dupACKs. To achieve this, we propose to use "fake" TCP ACK signaling from the hypervisor to the VM. For this, the hypervisor maintains a per-flow timer $\beta = \alpha * RTT + rand(RTT)$ to wait for the ACKs before it triggers FRR with fake dupACKs.

The resulting T-RACKs system consists of two components mainly the in/out packet processing operation driven by Algorithm 1 and the ACK time-out handling shown in Algorithm 2.

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 invoked 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 new flow record. *iii)* In (lines $13-14$), If Data packet arrives, the last sent sequence number and time of the flow is updated accordingly;
Next, on each TCP ACK arrival, the program (lines $16-31$) performs the following actions: *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., 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, we discuss this part in more detail later.
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 a new data has arrived for β secs since the last new ACK arrival, the flow times-out and then an 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 senders TCP stack or OS. *ii)* In (lines $11-14$), if the backoff time and yet no new ACK has

been received, another ACK is created and sent out to the corresponding sender. Each time the algorithm backs-off exponentially per retransmission until the hard coded minRTO timeout is reached. *iii)* In (lines $15 - 18$), If backoff approaches the minRTO (i.e., 200ms), stop triggering Fast-Retransmit (by resetting the soft state) and let 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.

3.1 System Design and Algorithms

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]. Figure 6 depicts the deployment of T-RACKs on datacenter end-hosts and shows its role in our system. It shows that flows are hashed 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. For instance, when VM1 on the sender established a connection with the same VM on the receiving end-host, a new flow entry is created (i.e., flow entry $A2 : X2$). 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, the loss is recovered by FRR. In this case, the module does not intervene and the long 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 Fake dupACKs (or FRACKs) to the sender. Typically, the sender will trigger FRR to retransmit the lost segments within a reasonable time before the long TCP RTO is triggered. For example, in Figure 6, when Data segments of flow $A1 : X1$ are lost and the T-RACKs flow entry times-out, the module generates a FRACK and starts the exponential backoff to force the sender into FRR.

3.2 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 identifica-

[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.

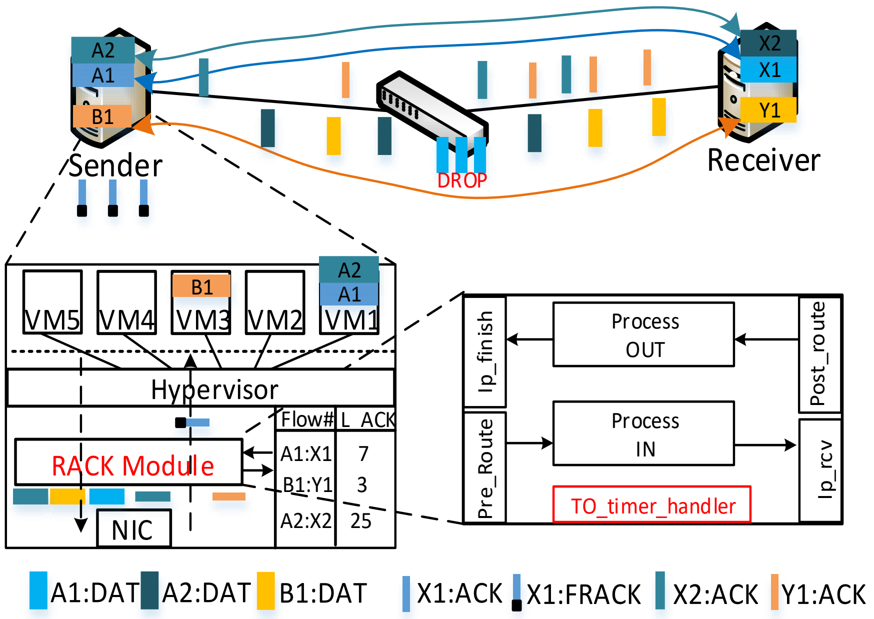


Figure 6: *T-RACKs System: It consists of an end-host module that track TCP flows incoming ACKs and generates FAKE ACKs whenever a flow timesout*

tion 4-tuple (IPs, Ports), per-flow last ACK number and the time-stamp of the last non-dupACK. The system in spirit is similar to recent works in [8, 11] that aim to enabling 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: emerges from 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 latency. This claim is supported by our observation and collected traces from our experiments on our cluster. A hash-based table is used to track flow entries of **active non-elephant** flows. In the worst case, when hashes collide, a linear

[6]ACKs may be updated whenever necessary in certain cases for example when SACK is in use to add SACK blocks if not present

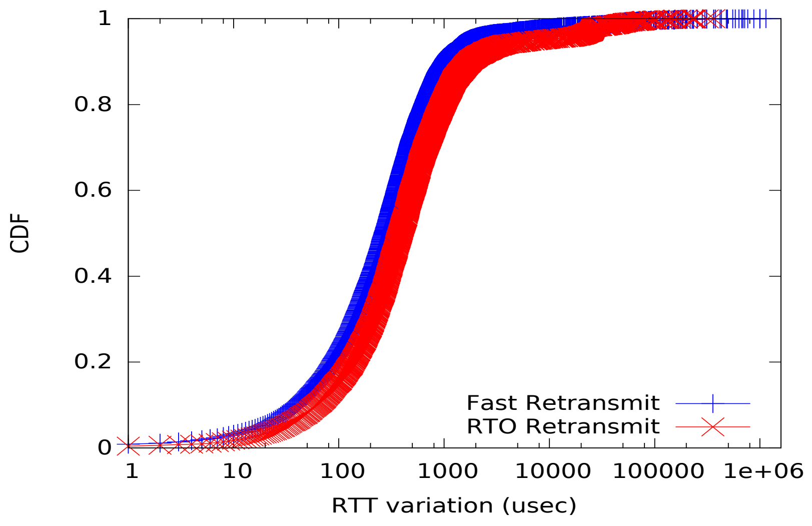


Figure 7: *RTT variation between transmission time and time of retransmission (i.e., $\Delta RTT = RTT_{rt} - RTT_t$)*

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 (i.e., all the operations including lookup in hash tables) would not affect the achieved TCP throughput[7].

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 right RTO value in TCP. For this purpose, we refer to a previous study [4], that essentially showed that even when a relatively bad RTT estimator is used, setting a relatively high minimum RTO (i.e., in hundreds of milliseconds) can help avoid many spurious retransmission in WAN transfers. This fact is supported by a subsequent study [37] that shows significant changes (or variance) in internet delays. Recent works [10, 21] show similar behavior within current datacenters. In our testbed setup, we observed noticeable variation in the measured RTT. Fig 7 shows the variation in smoothed RTT measurement of TCP sockets between the time of transmission of a packet and its fast retransmission or RTO retransmission, respectively. These variations can be mainly attributed to the beginning of

[7]TCP throughput is generally bound by the minimum its end-host NIC share and the in-network bottleneck share

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: Following up the previous discussion, 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 [27] 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 of such synchronization effect in a simulation, by setting the same ACK RTO. The results show repeated losses due to possible synchronized retransmissions. A viable solution for de-synchronizing such flows would be to introduce some randomness in the ACK RTO ultimately resulting into fewer flows experiencing repeated timeouts. We adopted this approach which justifies the 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 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 ACKed data.

Security Concerns: the receipt of each dupACK gives a signal for the sender that one of the transmitted segments has left the network. Based on this intuition, RFC 2581 suggested what is known as Fast Recovery Algorithm. In Fast Recovery, ***Cwnd***is set to $ssthresh + \phi \times MSS$ then for each additional dupACK, ***Cwnd***is artificially inflated by 1 MSS to account for the segment that left the network. This can be exploited to launch ACK spoofing attack [30] on the senders as the senders have no way to verify that incoming dupACKs are valid from the legitimate receiver or spoofed from some attacker. 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 ACK timer expires and entering a recovery state. This approach is adopted to disable ***Cwnd***artificial inflation during recovery and at the same time prevents ACK spoofing. A point worth-mentioning is that under T-RACKs dupACKs 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, the network could possibly replicate packets and hence the FRACK segments could be treated as replicated packets from within the network.

4 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[8]. 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.

4.1 Single-Rooted Tree Topology

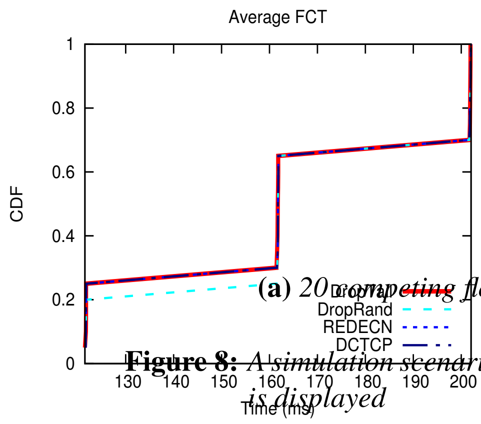
To study the behavior of TCP in response to packet losses and how likely it may recover quickly with the help of T-RACKs (Note, we refer to T-RACKs as RACK in the figures). We conducted a number of packet-level simulation experiments which covers a wide range of TCP and AQM settings. We also conducted simulations using congestion control mechanisms imported from Linux kernel (i.e., Cubic and New-Reno (abbreviated as Reno)). We use ns2 version 2.35 [26], which we have extended with T-RACKs mechanism inserted as a connector between nodes and their link in topology setup[9]. In addition, we patched ns2 using the publicly available DCTCP patch. we use in our simulation experiments speed links of 1 Gb/s 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 single bottleneck at the destination and run the experiments for a period of 15 sec. The buffer size of the bottleneck link is set to be more than the bandwidth-delay product in all cases (100 Packets), the IP data packet size is

[8]Note, for brevity, we refer to T-RACKs as RACK in the figures

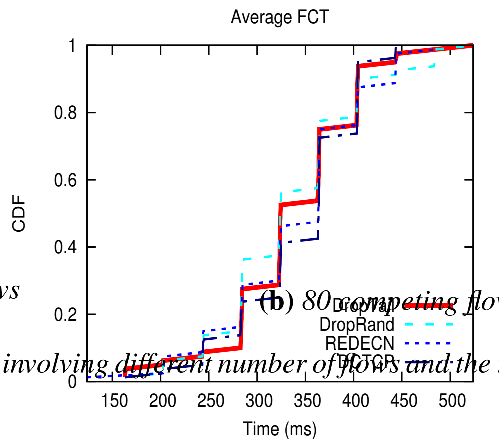
[9]Simulation code is available upon request from the authors.

1500 bytes. We designed one elephant-free and one elephant-coexisting simulation scenarios where N (mice) TCP flows send a 14.6KB file (i.e., 10 MSS) while $\frac{N}{4}$ (elephant) flows send non-stop during the simulation. Mice flows start with a short inter-start time that is drawn from an exponential distribution with mean equal to transmission time of one packet. The simulation has 5 epochs every 3 sec and the order of servers are varied in each epoch by drawing server number from a uniform distribution. We study packet loss, the likelihood of fast recovery and recovery time. We first study TCP newReno with DropTail, TCP with RED-ECN, TCP with Random Drop AQM and DCTCP covering the most common TCP and AQM settings.

First, Fig 8 shows the flow completion time for different schemes when T-RACKs is not enabled. They show significantly high FCT in DropTail and DropRand cases and schemes such as RED ECN marking and DCTCP can help regulate the queue at certain level which helps improve the FCT.



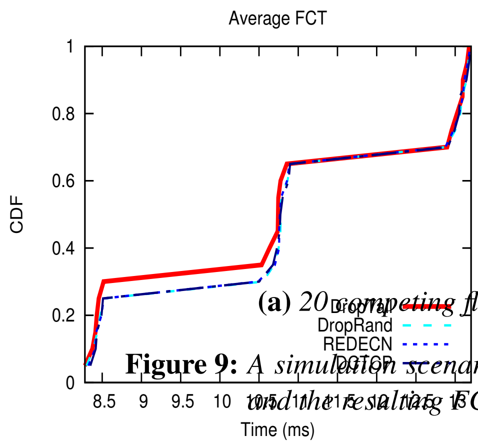
(a) *20 competing flows*



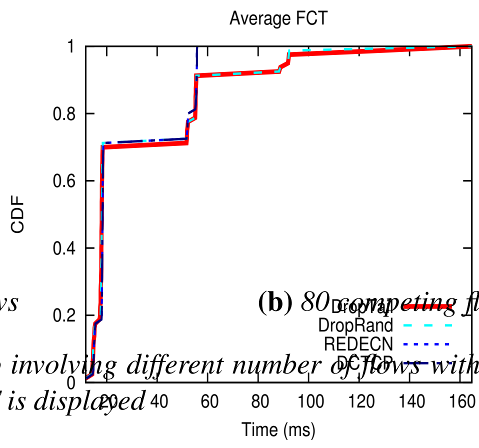
(b) *80 competing flows*

Figure 8: *A simulation scenario involving different number of flows and the resulting FCT is displayed*

We repeat the same simulation however with T-RACKs enabled on end-hosts to enforce dupACKs whenever the T-RACKs timer expires. Fig 8, shows significant improvement to the average FCT in 20 and 80 flow cases for all schemes and AQMs in use. We notice a significant improvement for FCT for both cases where the advantage of DCTCP and RED-ECN over DropTail and DropRand become more obvious. This is attributed to the low queue occupancy they can sustain when Timeouts are handled properly.



(a) *20 competing flows*



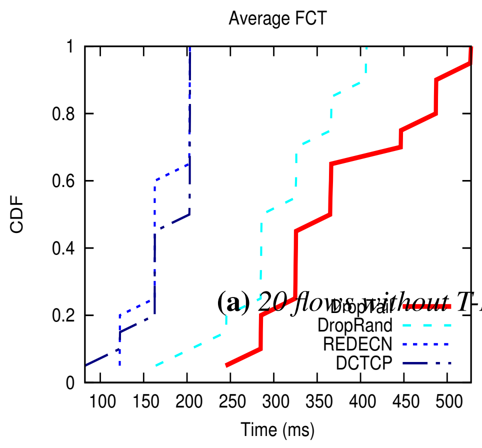
(b) *80 competing flows*

Figure 9: *A simulation scenario involving different number of flows with T-RACKs help and the resulting FCT is displayed*

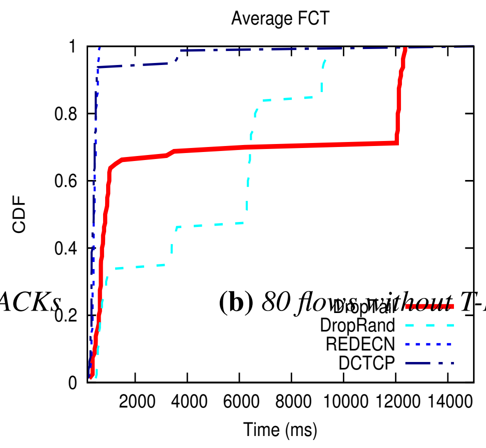
Now we repeat the same experiments however we introduce a long-lived background traffic to put all schemes under stress and see how T-RACKs can handle such scenario. Fig 10 shows that still T-RACKs can deliver almost the same performance improvements for schemes in 20 and 80 flows cases alike.

4.2 Large-Scale DataCenter Topology

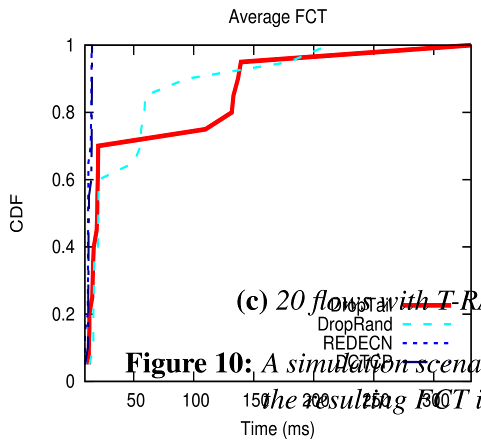
Since any end-host based scheme is assumed to be scalable, to verify this, we experiment with T-RACKs in a larger scale-setup with varying workloads and flow size distributions. For this purpose, we conduct another packet-level simulation using a spine-leaf topology with 9 leafs and 4 spines using link capacities of 10G for end-hosts and over-subscription ratio of 5 (the typical ratio in current production datacenters is in range of 3-20+). We again examine scenarios that covers a TCP-NewReno, TCP-ECN and DCTCP operating along with DropTail, RED and DCTCP AQM respectively. we use a per-hop link delays of 50 μs, TCP is set to the default TCP RTO_{min} of 200 ms and TCP is reset to an initial window of 10 MSS, and a persistent connection is used for successive requests. The flow size distribution for workload 1 and workload 2 are shown in Fig 11a which captures 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. Fig 11b shows the inter-arrival times



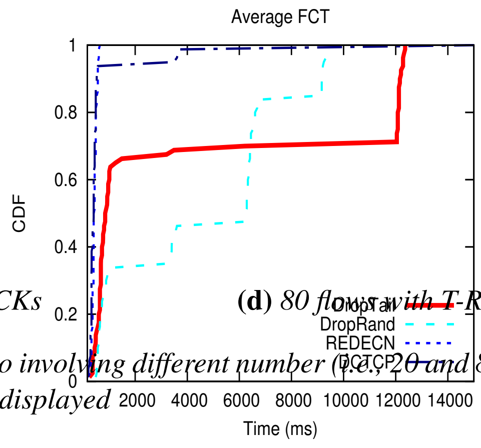
(a) *20 flows without T-RACKs*



(b) *80 flows without T-RACKs*



(c) *20 flows with T-RACKs*

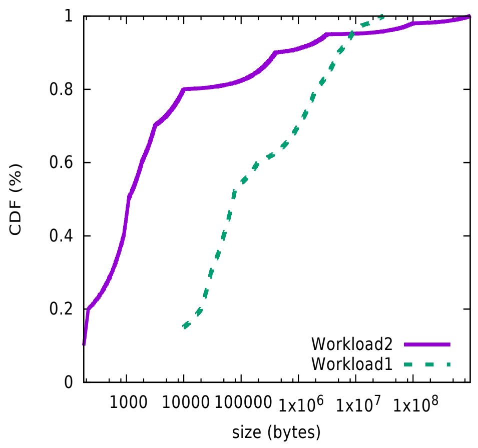


(d) *80 flows with T-RACKs*

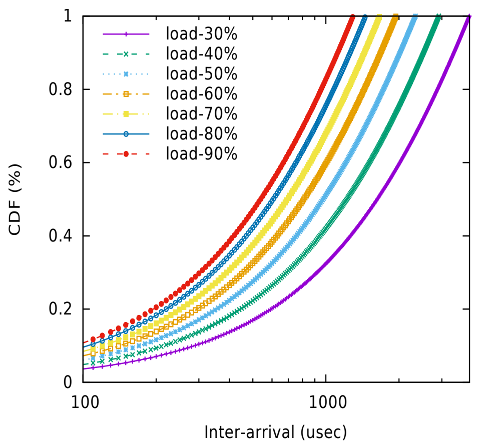
Figure 10: *A simulation scenario involving different number (i.e., 20 and 80) of flows and the resulting FCT is displayed*

distribution for various 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 and all flows (i.e., small [0-100KB], 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 in this experiment.

Figure 12 shows the average FCT for small, medium and all flows as well



(a) *CDF of flow size*



(b) *CDF of inter-arrival time*

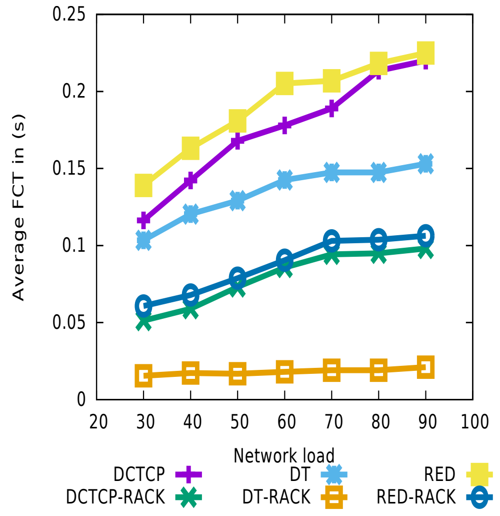
Figure 11: *Flow characteristics: (a) Actual Flow size distribution (b) Inter-arrival times for various network load*

as the total timeouts experienced by all flows in the websearch workloads. It is evident that in this workload that timeouts have a significant impact on the FCT of small TCP flows regardless of congestion control type and AQM in operation. This is where T-RACKs becomes of the utmost benefit to small flows in improving their FCT. In addition, Fig 12c shows performance gains for all flows, the reasoning behind the improvement of overall FCT for all flows is for two reasons:

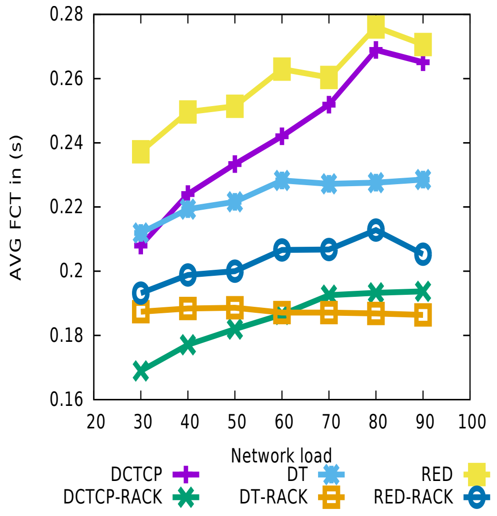
- In the experiments, elephant threshold is disabled which allows for all flows to benefit from the improvement provided by T-RACKs.
- When small flows finish faster, they leave the network and occupied resources becomes available for large flows.

Finally, Fig 12d shows the total number of RTO seen by TCP flows. The results strongly suggest that the improvement in FCT is mostly attributed to the lower number of RTOs when T-RACKs is taking care of fast recovery for TCP flows.

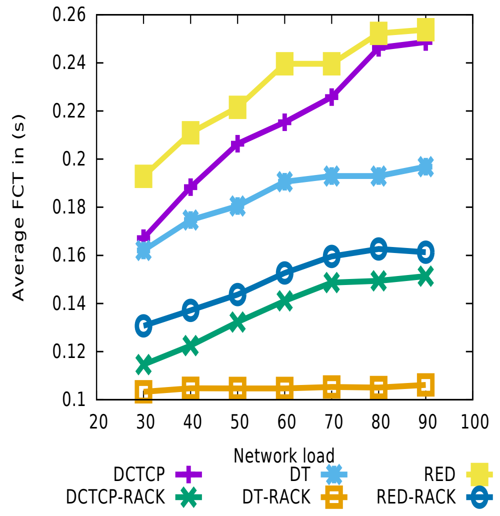
Similarly, Figure 13 shows the average FCT for small, medium and all flows as well as the total timeouts experienced by all flows in the datamining workloads. In this case, still all flows see noticeable improvement in FCT, however the performance gains are less than the websearch case. Furthermore, since almost 80% of flows are of size less than 10KB, hence overall they experience lesser timeouts. In such case, DCTCP can provide significant improvement in the FCT due to its



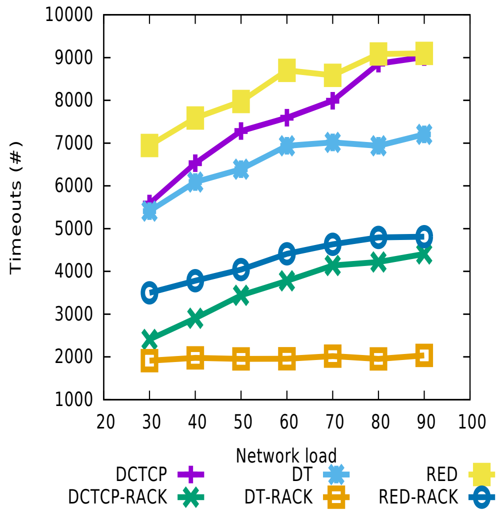
(a) *Small Flows: Average FCT*



(b) *Medium Flows: Average FCT*



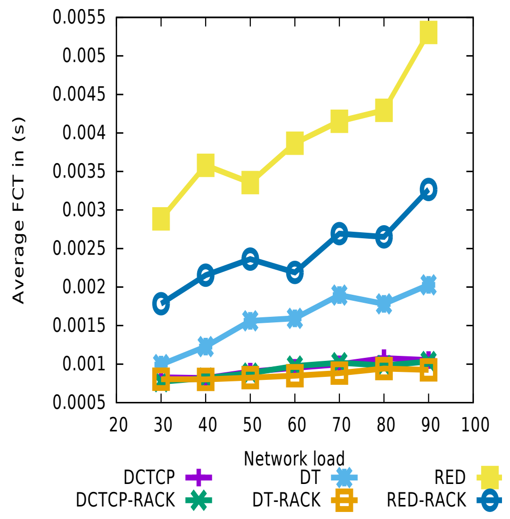
(c) *All Flows: Average FCT*



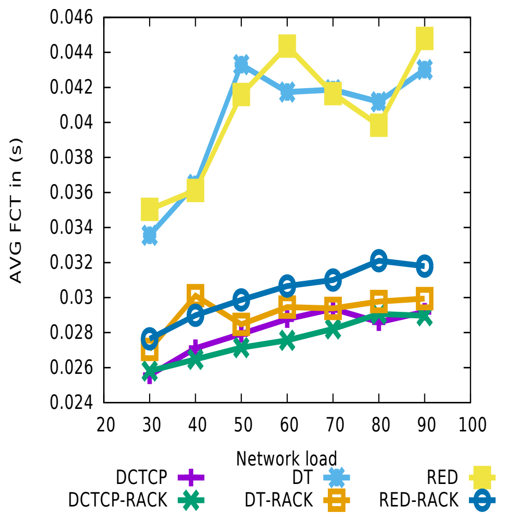
(d) *All Flows: Number of RTO*

Figure 12: *Performance metrics with various network load in range (30%, 90%) of web-search workload (i.e., workload 1)*

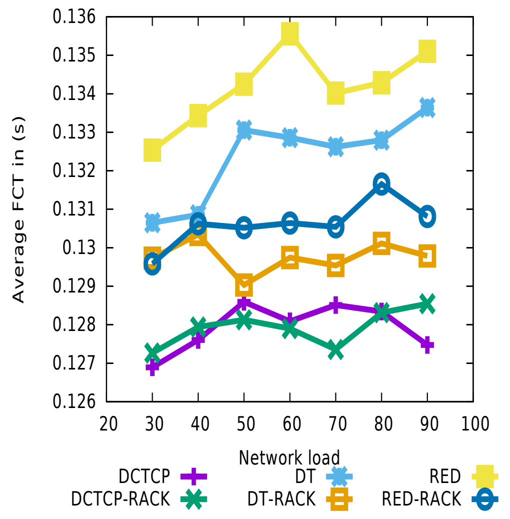
ability to regulate the queue at small operating regime. And yet T-RACKs can provide for added improvement to DCTCP performance.



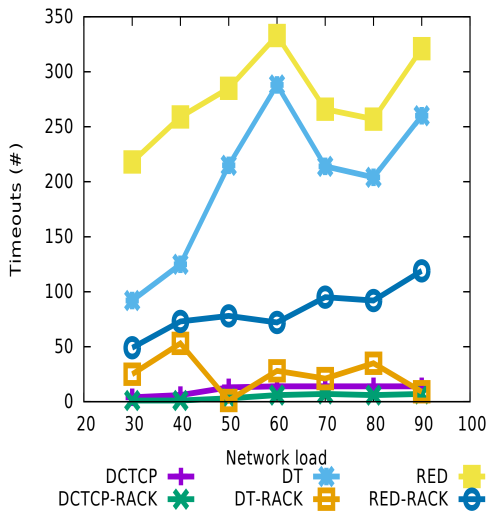
(a) *Small Flows: Average FCT*



(b) *Medium Flows: Average FCT*



(c) *All Flows: Average FCT*



(d) *All Flows: Number of RTO*

Figure 13: *Performance metrics with various network load in range (30%, 90%) of datamining workload (i.e., workload 2)*

4.3 Sensitivity to Choice of T-RACKs RTO Timer

We here repeat the last simulation experiment using websearch workloads with various values for the control variable α which defines the RTO value. The variable α is defined in terms of an RTT multiplicative factor and values in the range

[1, 5, 10, 50, 100] is used. The purpose of this simulation analysis is to assess the sensitivity of T-RACKs to the chosen value of α. The simulation settings, flow sizes, inter-arrival times and network loads are the same as in the previous setup. We report here the achieved FCT of small and all flows in each case for DropTail, RED and DCTCP. As shown in Fig 14, 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 unnecessary 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 should. We can see a value of 10 achieves a good trade-off between the two cases and can help improve the FCT by recovering faster to avoid large amount of spurious timeouts.

5 Implementation and Experiments

In this section, we discuss the implementation details of T-RACKs as a loadable kernel module in Linux Kernel then assess its performance by reproducing synthetically workloads encountered in production datacenters. T-RACKs is a transparent shim-layer residing between the TCP/IP stack (or VMs) and the link-layer (or Hypervisor). It was implemented by leveraging the NetFilter framework [24] which is an integral part of Linux OS. Netfilter hooks attach to the data path in the Linux kernel just above the NIC driver and below the TCP/IP stack. This imposes no modifications to the TCP/IP stack of the host OS nor guest OS and being a loadable module, it allows for easy deployment in current production datacenters. As shown in Fig 6, The module intercepts all incoming TCP packets destined to the host or its guests right before it is pushed up to TCP/IP stack handling (i.e., at the post-routing hook). First, the 4 identifying tuples are hashed and associated flow index into the Hash Table is calculated via Jenkins hash (JHash) [14]. Then, TCP packet headers are examined and the flag bits are used to choose the right course of action (i.e., SYN-ACK, FIN or ACK) following the logic in Algorithm 1. The module does not employ any packet queues to store the incoming packets, it only stores and updates flow entry states (i.e., ACK seq#, arrival time and so on) on arrival of segments. Since T-RACKs does not require fine-grained timers in micro-second scale, the timer used are the native Linux kernel timer (i.e., in millisecond-scale) traditionally used in the protocol stack. In addition, T-RACKs uses a single timer for all active flows (firing every 1 ms) to handle RTO events. These design choices help reduce the load on the end-host servers and make T-RACKs module as lightweight as possible.

To put T-RACKs to the test, we use a small-scale testbed consisting of 84 virtual servers interconnected via 4 non-blocking leaf switches and 1 spine switch. As shown in Figure 15, The testbed cluster is organized into 4 racks (rack 1, 2, 3 and 4). Each server per rack is connected to a leaf switch via 1 Gbps link. The spine switch is realized by running a "reference_switch" image on a 4-port NetFPGA card [25] which is installed on a desktop machine. The servers are loaded with Ubuntu Server 14.04 LTS with kernel version 3.18 which includes a functional version of DCTCP protocol [1]. The T-RACKs end-host module is invoked and installed on the host OS whenever necessary only. 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 2, to run the experiments with realistic traffic workloads shown in Figures 1a and 1b. In addition, we have installed the iperf program [13] to emulate long-lived background traffic (e.g., VM migrations, backups and so on) in certain scenarios. We setup different scenarios to reproduce an one-to-all and all-to-all w/wo background traffic. In one-to-all, clients running on the VMs in one rack send requests randomly to any of all other servers in the cluster. While in all-to-all scenario, all clients send requests to any of all other servers in the cluster. If background traffic is introduced, we run a continuous long-lived iperf flows in all-to-all fashion to evaluate T-RACKs under sudden and persistent network load spike. Finally, in the results, we classify flows with size $<= 100KB$ as small, $> 100KB$ *and* $<= 10MB$ as medium and $>= 10MB$ as large.

The objectives are: *i)* to verify if T-RACKs, helps short TCP flows finish faster and more flows meet their deadlines. *ii)* to verify how T-RACKs affects the performance of medium and large flows in terms of FCT and average throughput; *iii)* to quantify T-RACKs robustness in unexpected network loads (i.e., background) in the network.

5.1 Datacenter Workloads based Experimental Results

One-to-All scenario without Background Traffic: we run one-to-all scenario and 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 cluster and is set to randomly initiates 1000 requests to randomly picked servers on all other racks. Figures 16a, 16b and 16c show the average

FCT and missed deadlines for small flows as well as average FCT for all flows in websearch workload, respectively. While, Figures 16d, 16e and 16f, 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 in use in 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 educational and $\approx (69\%, -, 35\%)$ for private DC workloads. We notice that DCTCP improves FCT over its RENO and CUBIC counterparts and T-RACKs could improve its performance in terms of the missed deadlines in websearch. The average FCT of small flows in educational and private workloads has seen a slight increase of FCT with T-RACKs in certain cases. In these workloads, the network load is quite lite (as shown by the small FCT without T-RACKs) and hence T-RACKs added extra overhead surpasses its benefits. *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 selective small flows (only) T-RACKs intervention.

One-to-All scenario with Background Traffic: to put T-RACKs under a true stress, we run the same one-to-all scenario with all-to-all background traffic that shares the network with the running workload. We report similar metrics as in the aforementioned case. Figures 17a, 17b and 17c show the average FCT and missed deadlines for small flows as well as average FCT for all flows in websearch and Figures 17d, 17e and 17f show the average FCT for short flows in data mining, educational, private DC workloads, respectively. We observe the following: *i)* T-RACKs still 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 all-to-all scenario where all clients on all 4 racks initiate randomly 1000 requests to randomly picked servers in any of the 4 racks. Figures 18a, 18b and 18c show the average FCT for short flows in data mining, educational, private DC workloads, respectively. All-to-All introduces considerably higher network load, however, T-RACKs still can deliver significant improvements in the FCT regardless of the more complex nature of the All-to-All traffic.

In summary, the experimental results show the performance gains of T-RACKs especially for small flows, that constitute the lion's share in data centers, without harming larger flows. In particular, they show that:

- T-RACKs minimizes the variance of small TCP flows completion times and significantly reduce the missed deadlines.
- T-RACKs can maintain its gains even if bandwidth-hungry elephants are hogging the network.
- T-RACKs efficiently handles various workload and it is completely agnostic to TCP flavor.
- T-RACKs fulfilled its requirements with no assumptions about nor any modifications to in-network hardware nor the TCP/IP stack of guest VMs.

6 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 [15, 36, 31]. Specifically, [33] 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 [33]. However, 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 [2], or using receiver window based flow control [35] or deploying global traffic scheduling [3, 5, 22].

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. [12] 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 report due to its simplicity, and feasibility, is to recover losses by means of fast retransmit rather than waiting for long timeout. TCP-PLATO [31] 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.

7 Report Summary

In this report, we proposed an efficient cross-layer alternative to recover losses in timely manner before the occurrence of timeout. Our proposed approach proved to improve the FCT of time-sensitive flows and helps avoid throughput-collapse situations. A small-scale data-center setup was used to collect packet-level and TCP-socket traces to pinpoint clearly the root cause of long FCTs. Then we have design T-RACKs as a hypervisor-based (either sender-side or receiver-side) shim-layer residing between TCP in guest VMs and the network. T-RACKs allows TCP flows via transmission of fake dupACKS to efficiently recover from losses via FRR without waiting for RTO. T-RACKs is implemented as a linux-kernel loadable module and the testbed experiments in our 84-servers setup show that T-RACKs can improve the FCT and reduce missed deadlines for time-sensitive traffic and achieve high-link utilization by preventing TCP throughput-collapse. T-RACKs system is also shown to be light-weight due to its minimal footprint on end-hosts and is practical because it can be easily deployed in production environments. Finally, in public data centers, knowing that guest VMs and their networking stack is out of the control of the DC operator, T-RACKs proves to be especially adequate for such environments as it does require no modifications to guest TCP and no special hardware feature.

References

[1] M. Alizadeh. Data Center TCP (DCTCP). http://simula.stanford.edu/ alizade/Site/DCTCP.html.

[2] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). *ACM SIGCOMM Computer Communication Review*, 40:63, 2010.

[3] M. Alizadeh, S. Yang, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. Deconstructing datacenter packet transport. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks (HotNets)*, pages 133–138, 2012.

[4] M. Allman and V. Paxson. On estimating end-to-end network path properties. *SIGCOMM Compute Communication Review*, 29(4):263–274, Aug. 1999.

[5] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang. Information-agnostic flow scheduling for commodity data centers. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, NSDI'15, pages 455–468, Berkeley, CA, USA, 2015. USENIX Association.

[6] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement (IMC)*, IMC '10, pages 267–280, New York, NY, USA, 2010. ACM.

[7] T. Benson, A. Anand, A. Akella, and M. Zhang. Understanding data center traffic characteristics. *SIGCOMM Computer Communication Review*, 40(1):92–99, 2010.

[8] B. Cronkite-Ratcliff, A. Bergman, S. Vargaftik, M. Ravi, N. McKeown, I. Abraham, and I. Keslassy. Virtualized congestion control. In *Proceedings of the ACM SIGCOMM*, SIGCOMM '16, pages 230–243, New York, NY, USA, 2016. ACM.

[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 *Proceedings of the ACM SIGCOMM*, volume 09, pages 51–62, 2009.

[10] 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 *Proceedings of the ACM SIGCOMM*, SIGCOMM '15, pages 139–152, New York, NY, USA, 2015. ACM.

[11] 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 *Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference*, SIGCOMM '16, pages 244–257, New York, NY, USA, 2016. ACM.

[12] 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 the IEEE INFOCOM Conference*, pages 1–9. IEEE, apr 2016.

[13] iperf. The TCP/UDP Bandwidth Measurement Tool. https://iperf.fr/.

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

[15] Jiao Zhang, Fengyuan Ren, Li Tang, and Chuang Lin. Taming TCP incast throughput collapse in data center networks. In *Proceedings of IEEE International Conference on Network Protocols (ICNP)*, pages 1–10. IEEE, oct 2013.

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

[17] G. Judd. Attaining the promise and avoiding the pitfalls of TCP in the datacenter. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

[18] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. The nature of data center traffic. In *Proceedings of the ACM SIGCOMM conference on Internet measurement (IMC)*, New York, New York, USA, 2009.

[19] Linux Foundation. The Xen Project,leading open source virtualization platform. https://www.xenproject.org.

[20] M. Mattess, R. N. Calheiros, and R. Buyya. Scaling MapReduce Applications Across Hybrid Clouds to Meet Soft Deadlines. In *Proceedings of*

IEEE 27th International Conference on Advanced Information Networking and Applications (AINA), 2013.

[21] 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 the ACM SIGCOMM*, SIGCOMM '15, pages 537–550, New York, NY, USA, 2015. ACM.

[22] A. Munir, I. A. Qazi, Z. A. Uzmi, A. Mushtaq, S. N. Ismail, M. S. Iqbal, and B. Khan. Minimizing flow completion times in data centers. In *Proceedings of the IEEE INFOCOM Conference*, pages 2157–2165. IEEE, apr 2013.

[23] N. Dukkipati, N. Cardwell, Y. Cheng, M. Mathis. Tail loss probe (tlp): An algorithm for fast recovery of tail losses. https://tools.ietf.org/html/draft-dukkipati-tcpm-tcp-loss-probe-01.

[24] NetFilter.org. NetFilter Packet Filtering Framework for linux. http://www.netfilter.org/.

[25] netfpga.org. NetFPGA 1G Specifications. http://netfpga.org/1G_specs.html.

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

[27] V. Paxson, M. Allman, J. Chu, and M. Sargent. Computing TCP's Retransmission Timer, 2011. https://tools.ietf.org/html/rfc6298.

[28] 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 *Proceedings of the 6th Usenix Conference on File and Storage Technologies (Fast)*, 2008.

[29] S. M. Rumble, D. Ongaro, R. Stutsman, M. Rosenblum, and J. K. Ousterhout. It's Time for Low Latency. In *Proceedings of 13th Workshop on Hot Topics in Operating Systems (HotOS)*, pages 1–5, 2011.

[30] S. Savage, N. Cardwell, D. Wetherall, and T. Anderson. Tcp congestion control with a misbehaving receiver. *SIGCOMM Computer Communication Review (CCR)*, 29(5):71–78, Oct. 1999.

[31] 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 Journal on Selected Areas in Communications*, 32(1):65–76, jan 2014.

[32] tcpdump.org. Tcp dump. http://www.tcpdump.org.

[33] 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. *ACM SIGCOMM Computer Communication Review*, 39:303, 2009.

[34] G. Wang and T. S. E. Ng. The impact of virtualization on network performance of amazon EC2 data center . In *Proceedings of the IEEE INFOCOM Conference*, pages 1163–1171. IEEE Xplore, 2010.

[35] H. Wu, Z. Feng, C. Guo, and Y. Zhang. ICTCP: Incast congestion control for TCP in data-center networks. *IEEE/ACM Transactions on Networking*, 21:345–358, 2013.

[36] J. Zhang, F. Ren, L. Tang, and C. Lin. Modeling and Solving TCP Incast Problem in Data Center Networks. *IEEE Transactions on Parallel and Distributed Systems*, 26(2):478–491, feb 2015.

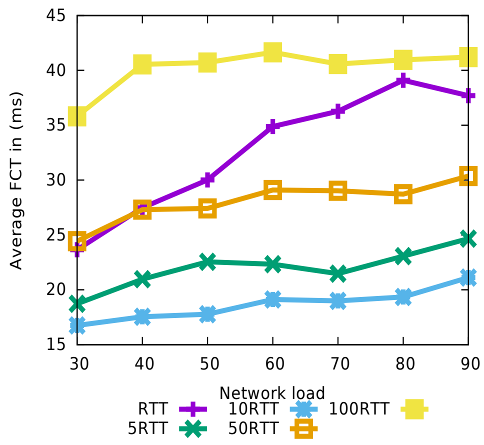
[37] Y. Zhang and N. Duffield. On the constancy of internet path properties. In *Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement (IMC)*, IMW '01, pages 197–211, New York, NY, USA, 2001. ACM.

Algorithm 1: T-RACKs Packet Processing

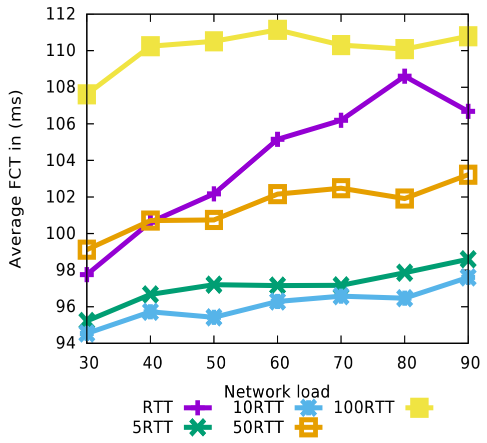
```
1  /* Initialization                                          */
2  Create flow cache pool;
3  Create flow table and reset flow information;
4  Initialize and insert NetFilter hooks;
   Input: $\alpha$ # of RTTs to wait before retransmitting ACKs
   Input: $\gamma$ the threshold in bytes to stop tracking flows
   Input: $\phi$ the dupACK threshold used by TCP flows
   Input: $t$: the current local time counted in jiffies
5  Define $x$: the exponential backoff counter
6  $\beta = \alpha * RTT + rand(RTT)$;
7  Function Outgoing Packet Event Handler (Packet P)
8      f=Hash(P);
9      if SYN(P) or f.inactive then
10         Reset Flow (f);
11         Extract TCP options (i.e, TStamp, SACK and so on);
12         Update the flow information and flag entry as active;
13     if $DATA(P)$ then
14         Update flow info (i.e, last seqno, .., etc);
15         f.active(t)=t;
16 Function Incoming Packet Event Handler (Packet P)
17     /* For ACK pkts: extract and update flow
          information using ACK headers                        */
18     if $ACK\_bit\_set(P)$ then
19         f=Hash(P);
20         if f.elephant then return ;
21         Extract required TCP header values ACK-seq, .. ,etc;
22         if New ACK then
23             Update flow entry and state information;
24             Update the last seen new ACK from receiver;
25             Reset $f.dupack = 0$;
26             Reset $f.ACK(t) = t$;
27             if $f.lastackno \geq \gamma$ then $f.elephant = true$ ;
28         else
29             if Duplicate ACK then
30                 $f.dupack = f.dupack + 1$;
31                 /* Drop extra dup-ACKs in T-RACKs mode         */
32                 if $f.resent > 0$ then Drop Dup ACK ;
33         Update the TCP headers (if necessary, Timestamps and SACK
             information;
```

Algorithm 2: T-RACKs Timeout Handler

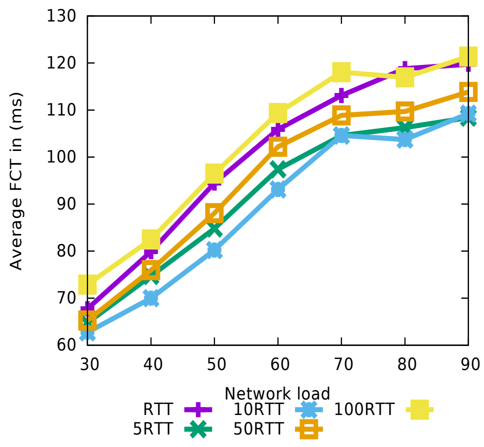
```
/* Initialization */
Create and initialize a timer to trigger every 1 ms;
Function Timer Expiry Event Handler
    for Flow (f) $\in$ FlowTable do
        if $!f.Active\ or\ f.elephant$ then Continue ;
        T = MAX(f.ACK(t), f.active(t));
        if $(t - T) \geq \beta$ then
            resend last ACK ($\phi - f.dupack$) times;
            set $f.resent(t) = t$;
            set $x = 2$;
            Continue;
        if $(t - f.resent(t)) \geq (\beta \ll x)$ then
            resend ACK one more time;
            $x = x + 1$;
            Continue;
        if $(t - f.ACK(t)) \geq TCPMinRTO$ then
            stop RACK recovery;
            soft reset flow (f) recovery state;
            Continue;
        if $(t - f.active(t)) \geq 1$ then deactive_flow(f) ;
```



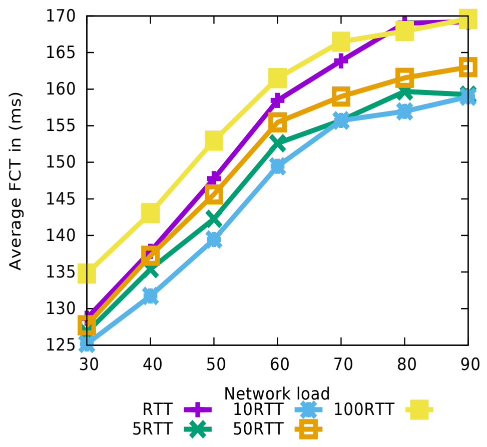
(a) *Small flows: Average FCT using DropTail*



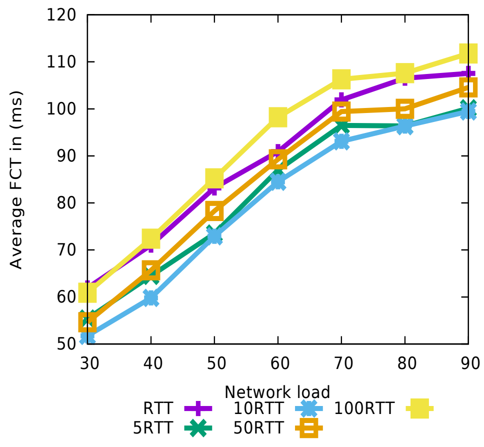
(b) *All flows: Average FCT using DropTail*



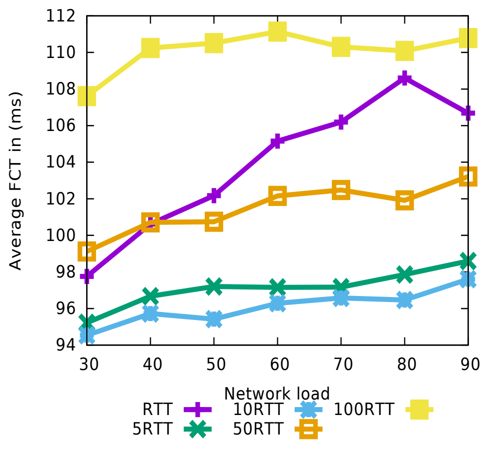
(c) *Small flows: Average FCT using RED*



(d) *All flows: Average FCT using RED*

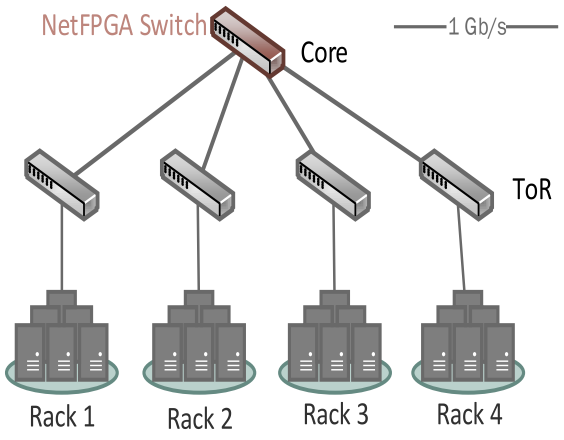


(e) *Small flows: Average FCT using DCTCP*



(f) *All flows: Average FCT using DCTCP*

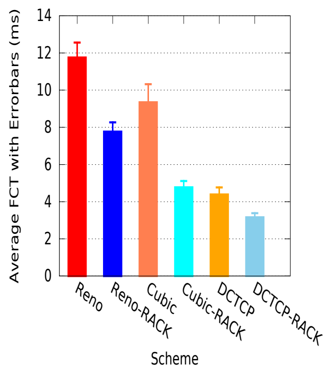
Figure 14: *Average FCT for small flows and all (small, medium and large) flows when α is varied in range [1, 100] for different AQMs (i.e., DropTail, RED and DCTCP)*



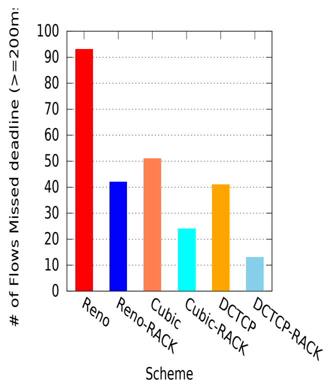
(a) *The testbed topology*

(b) *The actual testbed*

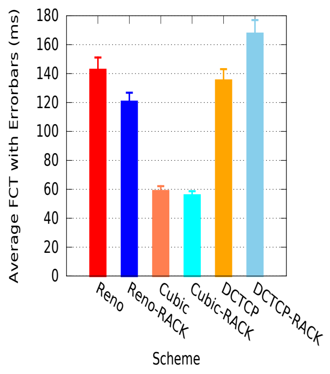
Figure 15: *Testbed setup of T-RACKs in small-scale cluster*



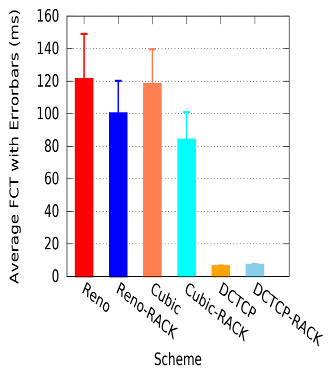
(a) *Small Flows: Average with Errorbar*



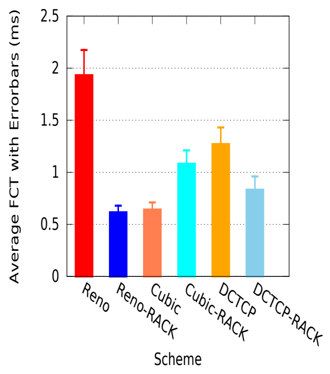
(b) *Small Flows: Missed Deadlines*



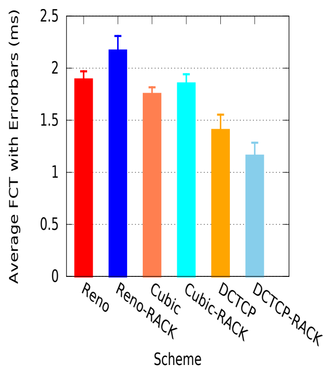
(c) *All Flows: Average with Errorbar*



(d) *Small Flows: Average (Datamining)*

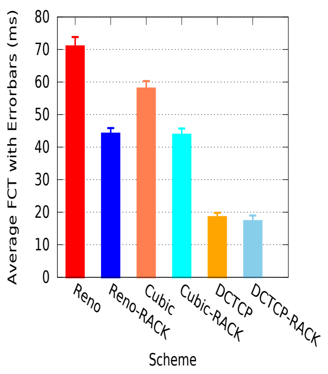


(e) *Small Flows: Average (Educational)*

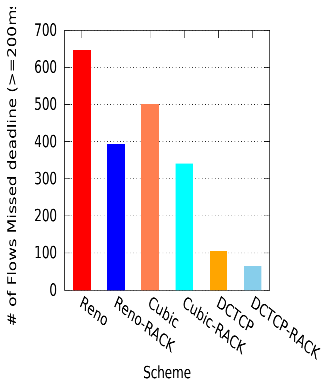


(f) *Small Flows: Average (Private DC)*

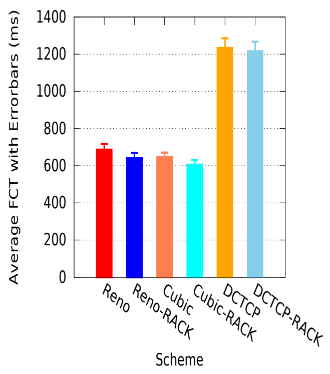
Figure 16: *Performance Metrics of all-to-all scenario without any background traffic*



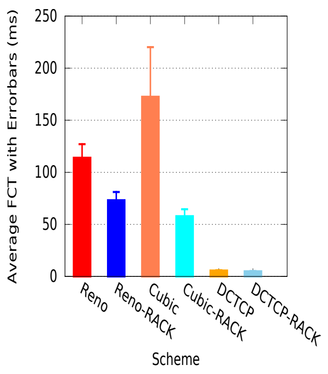
(a) *Small Flows: Average with Errorbar*



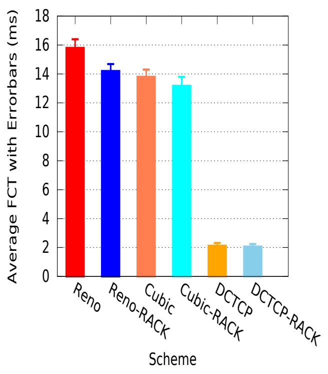
(b) *Small Flows: Missed Deadlines*



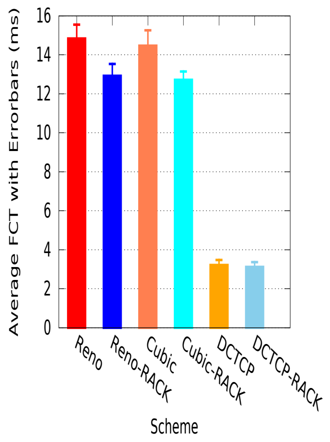
(c) *All Flows: Average with Errorbar*



(d) *Small Flows: Average (Datamining)*

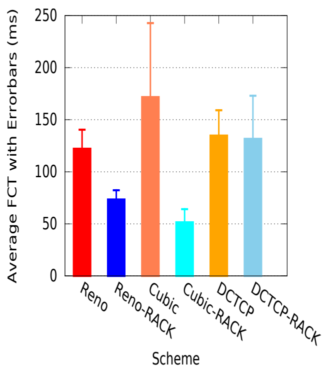


(e) *Small Flows: Average (Educational)*

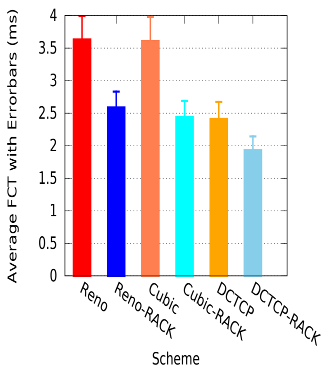


(f) *Small Flows: Average (Private DC)*

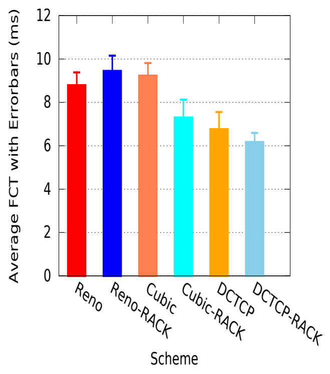
Figure 17: *Performance metrics of one-to-all scenario with background traffic*



(a) *Small Flows: Average (Datamining)*



(b) *Small Flows: Average (Educational)*



(c) *Small Flows: Average (Private DC)*

Figure 18: *Performance metrics of all-to-all scenario for datamining, educational and private DC workloads*