Taming Latency in Data centers via Active Congestion-Probing

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

***Abstract*—In cloud environments, interactive applications deployed in data centers often generate swarms of short-lived data transfers (or flows) that face dramatic competition for the scarce switch buffer space from other short-lived as well as the long-lived flows. In the presence of bloated queues, such short-lived flows often experience multiple packet losses per round-trip time which often triggers the timeout-based loss recovery mechanism. A direct consequence of this is an inflated application response time that turns out to be orders of magnitude larger than what it should be. A data center aware TCP protocol (DCTCP) was designed as a new TCP specifically to address this issue, however, it does not consider its co-existence with other transport protocol (e.g., CuBIC and NewReno of Linux). In such situations, which are abundant in multi-tenant data centers, the legacy large initial congestion window sizes (e.g., 10 segments), induce multiple packet losses at the onset of a TCP flow, which forces timeout and even binary exponential backoff. In this paper, we propose a novel Hypervisor-based, application-transparent approach for active congestion probing to enable the hypervisor to infer on-path congestion before the TCP connection is fully established for new traffic to avoid such massive packet losses and timeout. The so-called ProBoSCIS mechanism does not require any changes to TCP, works with all versions of TCP and does not need any special network hardware features other than those that exist in today's data center commodity switches. We show its effectiveness via ns2 simulation and demonstrate its practical feasibility by implementing and deploying it in a small-scale data center test-bed. We show the significant reduction in application latency by adopting ProBoSCIS in a series of real experiments.**

***Index Terms*—Congestion Control, Active Probing, Latency, TCP-ECN.**

I. Introduction

Cloud-driven online data-intensive (COLDI) applications built on top of distributed frameworks such as Hadoop [1] or Spark [2] are legion in today's data centers. Such applications rely on distributed storage and parallel processing systems to handle large data sets by dividing the workload among many workers, collecting then the results in real time as they are produced. In general their aim is to construct a complete (or partial) result by aggregating the many fractions of results obtained from the workers as quickly as possible, to be able to respond to interactive users in near real time. As a direct consequence, a timely data transfer is crucial to building highly scalable and responsive COLDI applications. Typically any missed deadlines, due to excessive network delays or retransmissions due to packet losses, result in revenue loss [3]. For example in search engines, missed deadlines affect the quality of the search results, and eventually affect the user perception of the search engine. Many solutions were proposed to improve the performance of such delay-sensitive applications in private data center networks by applying resource provisioning, or service preemptive prioritization policies that enable higher priority applications to preempt lower priority ones [4]. In practice, such approaches only address the performance problems partially (if at all); in large-scale multi-tenant clouds, the network is reportedly still the major performance bottleneck for most distributed applications as it is still not completely virtualized like other resources (CPU cores, RAM, and Storage). In addition, while the aforementioned approaches may improve the average latency of the flows, the performance of such interactive applications depends much more on the variation of the latency among flows than just on the average or the median. For example, it is found in many measurement studies in real data centers (e.g., [3]) that the 90^{th} percentile of the Flow Completion Times (FCTs) can be three to 4 orders of magnitude worse than the median or average.

TCP is still the predominant transport protocol in data centers, and in the presence of small switch buffers a few long-lived (elephant) flows or a swarm of short-lived (mice) flows can quickly and fully occupy the available buffer space. Hence, TCP flows face constantly bloated buffers and/or incast congestion, which in conjunction with the very small RTTs lead to excessive timeouts for some flows[1]. Noting that in most "out-of-the-can" TCP implementations the timeout timer is set to 150$\tilde{3}$00 ms, the FCT of such flows is bloated dramatically in data centers. Since, TCP is not designed to handle these complex congestion events [5–8], DCTCP [6] was among the most prominent TCP re-designs for data centers. DCTCP has seen a wide adoption in private data centers with homogeneous protocol stacks as it shows very good performance in such environments. Nevertheless, it fails to live up to the expectations in multi-tenant clouds with heterogeneous TCP implementations that respond differently than DCTCP to congestion [9]. For instance, DCTCP fails

[1]Typically small buffers and RTTs imply small flight sizes. In the presence of multiple non uniform losses per flight, the flow is left with an insufficient number of packets per flight to trigger recovery via three duplicate ACKs.

to handle packet losses that are caused by the use of large initial congestion windows, which is standard in heterogeneous networks. In this paper, we propose to develop a non-intrusive system that brings the same performance gains shown by DCTCP in private clouds [6] to public and multi-tenant clouds in a deployment friendly manner, without requiring changes to the guest operating system nor to the network switches. More precisely, we propose to solve the afore-mentioned issues while fulfilling the following requirements: (R1) Our system must help latency-sensitive applications improve their FCT; (R2) It should not achieve this at the expense of elephant flows (like in preemptive systems), i.e., it should not sensibly degrade elephant flows performance; (R3) It must comply with the VM autonomy principle by being transparent to the tenants protocol stack. In other words, it should not modify anything that is controlled by the tenant (e.g., TCP/IP stack of guest VM). If changes are needed then they are applied to the hypervisors that are fully under the control of the cloud operator; (R4) and finally it must be simple enough to appeal to vendors and data center operators alike; in other words, it should not require changes to the expensive-to-replace switching devices.

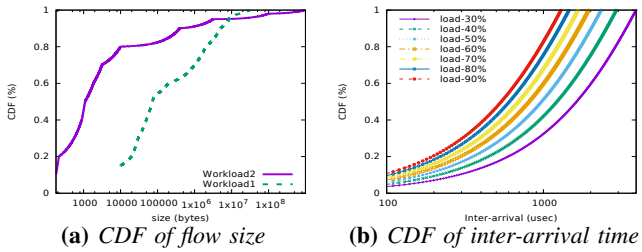
To achieve this we propose the so-called ProBoSCIS[2] scheme that uses a simple live active probing at source connection to ascertain that enough buffer space is available along the path to accommodate the imminent traffic with a standard initial TCP congestion window, or otherwise quenches the source rate via the receiver window to ensure that the imminent traffic avoids using the large initial congestion window. ProBoSCIS is implemented is in the hypervisor at the sender and receiver and relies only on ECN capability in the switches to probe for congestion.

In the remainder, we highlight the source of performance degradation in data centers in Section II, then present ProBoSCIS mechanism in Section III. In Section IV, we discuss our simulation results, and present in Section VI, the experimental results from a small testbed implementation of ProBoSCIS. We discuss related work in Section VII, and finally conclude the paper in Section VIII.

II. The Major Source of Increased Latencies

In this section, we use workloads extracted from production data centers and conduct an experimental analysis of network latencies. We focus on packet losses that lead to recovery via Retransmission Timeouts (RTO), and call them in the sequel Non-Recoverable Losses (NRL) as they cannot be recovered via three duplicate ACKs (DUPACKs). Because the TCP minRTO in most existing operating systems is set between 100 and 300 ms (e.g., 200ms in Linux), NRLs inflate artificially the latency by 3 to 4 orders of magnitudes compared to the RTT usually experienced in data centers. These losses can be described as follows: 1) **Steady State Losses (SSL)**: The tail-end packet(s) of some flows are dropped due to buffer overflow. In this case, the sender would not be able to receive three DUPACKs to trigger the Fast Retransmit and Recovery (FRR) mechanism; and 2) **Connection Setup Losses (CSL)**: A burst of packets gets dropped due to the large initial sending window of TCP. In this case, the sender may lose the whole window or a large portion of it, leading to a similar result as above. When these two cases take place in a congested network, the sender is forced to wait for the expiry of the RTO to re-transmit the lost packet(s), adding thus the waiting time for RTO to the usually small nominal latency in data centers.

[2]ProBoSCIS stands for Probing the Buffer On Source Connection for Incast Stalling.



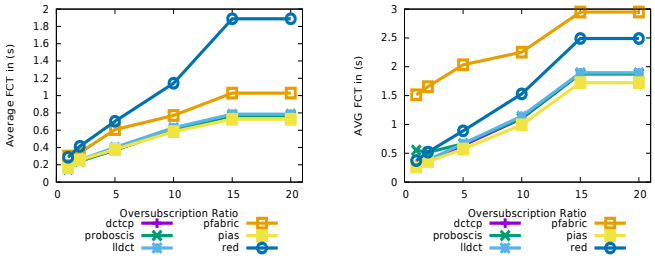
(a) *CDF of flow size* **(b)** *CDF of inter-arrival time*

Figure 1: *Flow characteristics: (a) Actual Flow size distribution (b) Inter-arrival times for various network load*

To study these phenomena, we conduct a series of ns2 simulations to quantify the impact of SSL and CSL. More specifically we seek to understand how well RED-ECN (baseline) and Least Attained Service (LAS) [10] flow scheduling (i.e,. LLDCT [11], pFabric [12], and PIAS [13]) perform when these schemes experience SSL or CSL. We use a traffic generator to statistically reproduce real-world workloads (e.g., Workload 1 is Websearch [6] and Workload 2 is Datamining [14]).

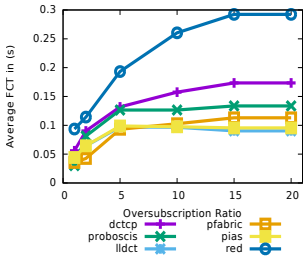
Fig. 1 shows the flow size and inter-arrival time distributions for various loads. We use a spine-leaf topology of 9 leaf and 4 spine switches and 16 servers per rack to form a network of 144 nodes. To stress the buffers in the network, we run the experiments in 50% loaded network, with various over-subscription ratios, ranging from 1 to 20, from the leaf-level to the spine-level. We consider a flow to be a small flow if its size is no larger than 100KB. Figure 2 shows the FCT of small flows and the FCT of all flows for both Websearch and Datamining workloads for various over-subscription ratios from leaf level to spine level.

We conduct the same experiments again by varying the initial congestion window size ($ICWND$) in the range of [1-20] for a fixed over-subscription ratio of 1:5. Figure 3 shows the FCT of small flows and that of all flows for Websearch and Datamining workloads. **The results show that the FCT of small flows is highly sensitive to the choice of initial window, regardless of the congestion control scheme used**, whereas the FCT of large flows remains consistent over the various initial window values for each scheme. This suggest that fine tuning the initial window is necessary in order to achieve better performance, especially for small flows, which abound in data centers as shown in Figure 1a. In addition, all the schemes other than RED-ECN and pFabric achieve significant performance gains on the FCT in Websearch and Datamining workloads. However, again the schemes imple-

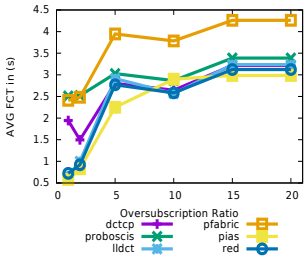


(a) *Small Flows: Websearch*

(b) *All Flows: Websearch*

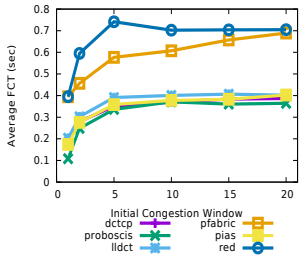


(c) *Small Flows: Datamining*

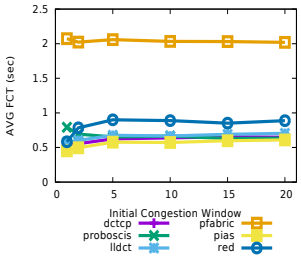


(d) *All Flows: Datamining*

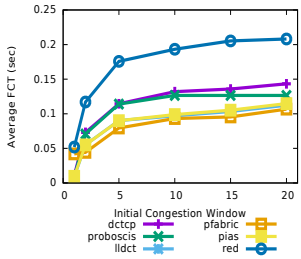
Figure 2: *Average FCT for small flows and all flows when oversubscription ratio is varied in range [1, 20].*



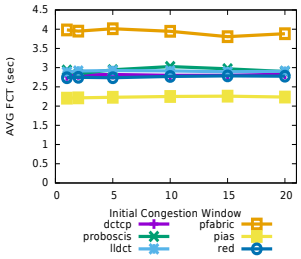
(a) *Small Flows: Websearch*



(b) *All Flows: Websearch*



(c) *Small Flows: Datamining*



(d) *All Flows: Dataminig*

Figure 3: *Average FCT for small flows and all flows when ICWND is varied in range [1, 20]*

menting packet-tagging achieve better results in Datamining. The results also show that proportional ECN-marking schemes such as DCTCP and ProBoSCIS are also effective.

These results show that the FCTs of small and large flows for all schemes are sensitive to the offered load and hence the buffer occupancy levels. However, all the schemes other than RED-ECN achieve almost the same considerable improvements in the FCT in Websearch workload. In stark contrast, in Datamining their performance vary greatly because of the heavily skewed flow size distribution with a majority of small flows. Noticeably, the schemes implementing Least Attained Service (LAS) achieve the best results. Similar gains can also be easily obtained by adopting a simple ECN schemes such as DCTCP.

Inspecting the traffic and results more closely, we find that: SSL and CSL primarily affect the performance of small (mice) flows; we also observe that LAS-based schemes perform worse when they encounter SSL and CSL in highly congested networks; and finally an improper TCP initial congestion window setting may result in the degradation of mice flows performance. These observations lead us to the question "How to design a solution that minimizes the delay of mice flows while being able to leverage sending at a large initial congestion window without incurring losses". The answer to this question is a mechanism that can actively probe the network to measure its congestion level and approximately choose an appropriate size for the initial congestion window of flows that are starting or restarting afresh. Other flows can handle the congestion properly once congestion avoidance takes over.

As DCTCP is simple and has shown good performance in homogeneous networks, we aim to propose an efficient solution to improve the performance of DCTCP in handling the SSL and CSL events. Our solution is transparent to the guest VMs and assumes no extra available hardware other than the commonly found features in commodity switches. The proposed solution consist of the following two major elements: 1) to address SSL, source rates are quenched based on a constant feedback information from the switches when pre-configured thresholds are exceeded; and, 2) to address CSL, the initial sending rate of the sources are scaled proportionally to the current level of network congestion.

III. The Proposed Methodology

In essence, ProBoSCIS is a simple mechanism that tries to bring the basic principles of the well-known CSMA-CD congestion management to the transport layer by: *1)* "Listening" to the network condition before injecting any new traffic into the network; *2)* Continuously monitoring the network for worsening conditions to adjust the sending rates accordingly. ProBoSCIS, is a hypervisor-to-hypervisor end-to-end network probing scheme that detects imminent congestion and throttles the senders by adjusting the receiver window field in incoming ACKs. The mechanism relies on ECN marking to convey the level of network congestion to the hypervisor. As ECN is readily available in all modern commodity data center switches, the scheme, does not require any change to the switches. Furthermore, being hypervisor-based, the resulting system does not require any changes to TCP itself and is transparent to the guest VMs. ProBoSCIS is distributed on the sender's hypervisor and the receiver's hypervisor. In normal operation, in the case of SSLs, the sender injects normal TCP data packets into the network, and the receiver echoes back the congestion markings on the ACKs, the sender uses these to estimate a level of congestion along the path and throttles the TCP sender in the VM by applying a proportional adjustment to the receiver's window field of the incoming packet (ACK). For the CSLs, as there are no packets in flight to elicit ECN markings and no ACKs to echo them, we resort to injecting a packet train from hypervisor-to-hypervisor before each SYN packet. These packets act as probes to carry the ECN marks

to the receiver who will then apply directly the proportional window resizing onto the outgoing SYN-ACK packet.

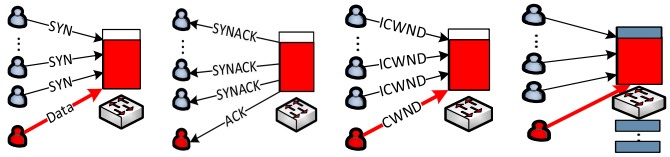


Figure 4: *TCP: from left to right, Syn; 1 RTT later Syn-Ack; In the next RTT, Ack and data with initial cwnd worth of data; leading to congestion and packet losses*

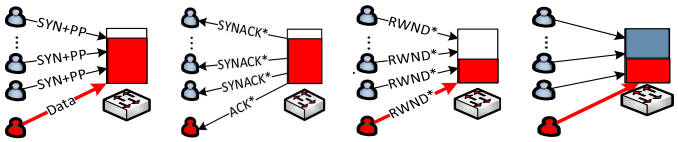


Figure 5: *ProBoSCIS: from left to right, Syn + probing packets; 1 RTT later Syn-Ack* with congestion-proportional rwnd* value; Ack and rwnd* worth of data in the next RTT leading to a buffer with enough room to absorb the burst*

To illustrate the principles behind ProBoSCIS, we consider the example of an elephant flow actively sending through a bottleneck link on an end-to-end path when a set of N new short-lived flows open connections through the same bottleneck. Figures 4 and 5 put in contrast the sequence of events for the CSL scenario in a pure TCP, versus TCP with-ProBoSCIS. In the case of pure TCP, the new N flows take one RTT to successfully open the TCP connection then send immediately an initial *Cwnd* (ICWND)-worth of segments over the next RTT. With most modern TCP implementations being tailored for the Internet, this would amount to injecting an extra $N\times$ kMSS in one RTT ($k = 10$ in Linux). This large burst of packets combined with the small switch buffers in the presence of a bloated packet queue at the bottleneck link results systematically in bursty packet losses that often are only recoverable via timeout.

In contrast, ProBoSCIS handles this case as follows: 1) firstly, the use of ECN and flow throttling would avoid the bloated queue as the continuous flow of ECN marks received by the steady-state elephant flow (shown in red) results in the associated sender's hypervisor adjusting actively and proportionally the receive window to limit its sending rate in the VM; 2) secondly, to avoid the initial massive bursts, the sender's hypervisor injects a train of small sized Packet Probes (PP) before the SYN packet. Their arrival at the receiver with ECN markings enables this latter to adjust the rate of the sender by adjusting the initial receive window of the SYNACK packet on its way back.

A. ProBoSCIS Algorithms

ProBoSCIS has a sender and a receiver algorithms both implemented in the hypervisor. The sender generates the probes and implements the DCTCP congestion control logic, while the receiver collects the probes and adjusts the RWND of the SYN-ACK to determine the initial window used by the sender. Algorithm 1 shows a summary of the sender's algorithm. It consists of the initialization (lines $1-3$) of the different parameters used by the algorithm, followed then by the two main functions: *i)* Incoming packets event handler (lines $5-12$): this function deals with incoming SYN-ACKs that are sent after the receiver-side released the SYN-ACK with a proportional window. The corresponding stored copy of the SYN in the SYN segments list is discarded and the SYN-ACK is release to the source VM. After connection setup, to enforce DCTCP, the incoming ECN-Echo packets are counted and α is updated as in [6]. Based on this, the receive window of the incoming ACKs is adjusted whenever necessary. This fulfils the solution for the SSL problem; *ii)* Outgoing packets event handler (lines $14-19$): This function is responsible for handling SYN segments arrival from the VM. This activates the flow entry creation in the hypervisor Flow_Table. The hypervisor then crafts a probe packet and sends it γ times followed by the SYN. A copy of the SYN is inserted to the SYN segments list; *iii)* Timeout Expiry Event Handler (line $24-40$): The handler goes through the SYN packets in the list and for each packet, if the period since their last departure T exceeds β_1, then the probes are resent where the amount increases linearly. Otherwise, if T exceeds β_2, which suggests that the SYN has been lost, then both the SYN and probes are resent. If T exceeds a pre-set maximum trial time MAX_TIME then the flow entry is hard reset.

A summary of the receiver algorithm is given in Algorithm 2. It consists of the initialization (lines 13) then the following two main functions: *i)* Incoming packets event handler (lines $5-12$): this function deals with incoming SYNs to open the connection, which activates the flow entry in the Flow_Table. It also performs the counting of the incoming ECN marked Probe Packets and updates parameter β accordingly. *ii)* Outgoing packets event handler (lines $14-19$): this function is responsible for handling SYN-ACK segments. When it receives a SYN-ACK from the receiver VM, it checks if enough probes have been received (i.e., more than $K\%$ of the total γ) and scales the outgoing receiver window in proportion to the received ECN marks in the probes (as indicated by $f.beta \leq \kappa_2$). Otherwise, i.e., if too many probes have been lost or the congestion level is too high, the receiver window of the SYN-ACK is updated to 1 MSS only, which assumes a severe congestion in the network. This behaviour suffices as the solution for the CSL problem; *iii)* Timeout Expiry Event Handler (line $19-26$): The handler goes through the SYNACKs in the SYNACK segments list and for each SYNACK if the congestion state conditions are satisfied then the SYNACK is admitted without update. Otherwise, if a total period since the last arrival of the SYNACK T exceeds β_3 then it will not be kept any further and hence it is admitted to the network.

Algorithm 1: ProBoSCIS Sender Algorithm

Input: α the moving average of the ECN echo packets
Input: κ_1 the marking threshold to adjust source rate
Input: γ the number of initial probes
Input: β_1, β_2 the timeouts for probes and SYN, respectively

```
/* Initialization                                          */
Create flow cache pool;
Create flow table and reset flow information;
Create and initialize a timer to trigger every 1 ms;
Initialize and insert NetFilter hooks;
Function Incoming Packet Event Handler (Packet P)
    key=hash(P);
    f=Flow_Table.find(key);
    /* SYNACKs: activate the flow if
       necessary and clear the entry for
       that flow                                           */
    if SYN(P) && ACK(P) then
        if !f.isactive() then f.activate_flow() ;
        f.synack_recv=now();
    if f.is_active() && ECN_Echo(P) then
        update f.ECNEcho_count and f.α;
        if ACK(P) && f.α ≥ κ1 then update f.α and
          RWND using DCTCP logic ;
Function Outgoing Packet Event Handler (Packet P)
    key=hash(P);
    f=Flow_Table.find(key);
    /* SYNs: active the flow, send the
       probes and update SYN timestamp                     */
    if SYN(P) then
        f.activate_flow();
        f.send_probes(γ);
        syn_list.insert(copy(P));
        f.syn_sent=now();
Function Timer Expiry Event Handler
    /* Timeout: Handle SYN packets waiting
       in the queue                                        */
    for each P ∈ syn_list do
        key=Hash(P);
        f=Flow_Table.find(key);
        T = MAX(now () - f.syn_sent, 0);
        if T ≤ MAX_TIME then
            if T ≥ β1 then
                resend gamma × f.probe_retry probes;
                f.probe_retry++;
            else if T ≥ β2 then
                resend gamma × f.probe_retry probes;
                f.probe_retry++;
                resend SYN(P) one more time;
                f.syn_retry++;
        else
            stop SYN recovery and hard reset flow (f);
```

Algorithm 2: ProBoSCIS Receiver Algorithm

Input: β the moving average of the ECN packets
Input: κ_2 the marking threshold to adjust source rate
Input: β_3 the timeouts for SYNACK, respectively

```
/* Initialization                                          */
Create flow cache pool;
Create flow table and reset flow information;
Initialize and insert NetFilter hooks;
Function Incoming Packet Event Handler (Packet P)
    key=hash(P);
    f=Flow_Table.find(key);
    /* SYNACKs: activate the flow if
       necessary and clear the entry for
       that flow                                           */
    if !f.isactive() then f.activate_flow() ;
    if SYN(P) then f.syn_recv=now() ;
    if f.is_active() && ECN(P) then
        f.ECN_count++;
        if PROBE(P) then
            drop(P)
Function Outgoing Packet Event Handler (Packet P)
    key=hash(P);
    f=Flow_Table.find(key);
    /* SYN-ACKs: activate the flow, update
       Rwnd , and store a copy of the SYN                  */
    if SYNACK(P) then
        f.activate_flow();
        /* Update Rwnd of SYNACK if the
           congestion level is below κ2                    */
        if f.syn_probes ≥ γ/K && f.β ≤ κ2 then
            TCP(P).rwnd=min(1 MSS, f.β × ICWND);
            synack_list.insert(P)
        else TCP(P).receive_window=1 MSS ;
        f.synack_sent=now()
Function Timer Expiry Event Handler
    for each P ∈ synack_list do
        key=Hash(P);
        f=Flow_Table.find(key);
        T = MAX(now() - f.synack_sent, 0);
        if (f.probes ≤ γ/K && f.alpha2 ≤ κ2) || T ≥ β3 then
            send(P) to source;
            f.synack_sent = now(); synack_list.delete(key);
```

B. System Design and Implementation

ProBoSCIS sender and receiver algorithms are integrated into a single hypervisor-level shim-layer implemented into the data processing path of the data center end-hosts (servers). This can be achieved in two ways: 1) by building a Kernel Module that leverages the NetFilter framework [15] to intercept and process the incoming and outgoing packets; or 2) by augmenting the hypervisor vswitch data-path (e.g., Open vSwitch [16]) with the packet processing functions.

In both cases, the module or the shim-layer would implement a combined version of the sender and receiver algorithms. That is the module would implement one function for incoming and outgoing processing which handles the events of SYN, SYN-ACK, PP packets arrival and timer expiry handlers (not shown in the algorithms for brevity). Note that in high speed large volume data centers like public ones, the functions can be built as custom-made network interface card.

In ProBoSCIS, all incoming and outgoing traffic to the guest VMs pass through the ProBoSCIS module for further processing. When the conditions for modifying either the SYN-ACK at the receiver or the ACKs at the sender are met, the receive window and checksum field are updated and the new ACK or SYN-ACK is shown as ACK* and SYN-ACK*. The ProBoSCIS module, at connection-setup, hashes the flow,

extracts the relevant information and stores it into a flow-table indexed by the 4-tuples of the flow (i.e., source IP, dest. IP, source Port and dest. Port). It stores various state information including the window scale factor, number of probes, number of non-ECN, ECN marks, and so on. Flow entries are cleared from the table when the connection is closed (i.e., FIN is sent by a guest VM). TCP checksum value is recalculated at the end-hosts whenever the receive window field is updated (this takes only one addition and one subtraction). The shim-layer (or module) resides right above the NIC driver for a non-virtualized setup and right below the hypervisor to support VMs in cloud data centers.

C. Practical Aspects of ProBoSCIS

In addition to being relatively simple and able to resolve the major causes leading to non-recoverable losses, ProBoSCIS can achieve a good performance both in simulation and real deployment. For example, ProBoSCIS can reduce the average FCT by one order of magnitude and the tail FCT by two orders of magnitude in our testbed experiments. Also, because it is hypervisor-based, and avoids any network stack alteration in the guest OS, it turns out to be readily deployable in existing production data centers with minor interruption to critical operations. We also expect it to be of low overhead and have minimal impact on background traffic. This is because of the small-sized, header-only packets used for probing are invoked only during connection setup.

One major concern is the use of window scaling by TCP, which according to TCP specifications [17], is achieved either via a three-byte scale option added to the TCP header in all segments or via a peer-to-peer exchange of window scaling factor at the stage of connection-establishment during the SYN segments exchange. However, most TCP implementations including Linux adopt the latter approach to avoid the extra network overhead (i.e., bandwidth) of the former. The scaling may be unnecessary for networks with small bandwidth-delay product (BDP) of 31.25Kbyte (i.e., 1 Gb/s for an RTT of 250μs). However, with the introduction of links operating at higher speeds e.g., 40 Gb/s (i.e., BDP=1.25 Mbyte) and 100 Gb/s (i.e., BDP=3.125 Mbyte), the scaling factor becomes necessary to utilize the bandwidth effectively. To this end, both the sender and receiver Algorithms of ProBoSCIS capture and store in the flow table the scale factor at the connection establishment phase. As a result, both need to take this scaling factor into account by explicitly rescaling the incoming and outgoing receive window field, respectively before and after applying the throttling. This operation takes a simple shift operation.

Algorithms 1 and 2 rely on ECN markings to infer the congestion in the network, hence the RED marking thresholds in the switches have to be set appropriately [18**?**]. RED has been criticized for its sensitivity to its parameter settings [19, 20]. RED specification [19] gives a guideline on choosing the RED parameters, however the choice of the optimal settings greatly depend on the attributes of the underlying network (e.g., link speeds, buffer size and the RTT). Although, in this work, we also rely on Weighted RED AQM for ECN marking which is commonly available starting from the entry-level data center switches, we use a simple and straightforward settings similar to DCTCP [6] where we set the parameters to perform marking based on the instantaneous queue occupancy exceeding the quarter of the full buffer capacity. The reason being that DCTCP AQM marking has been verified analytically [21] and practically [6] in data center environments.

IV. Simulation Analysis

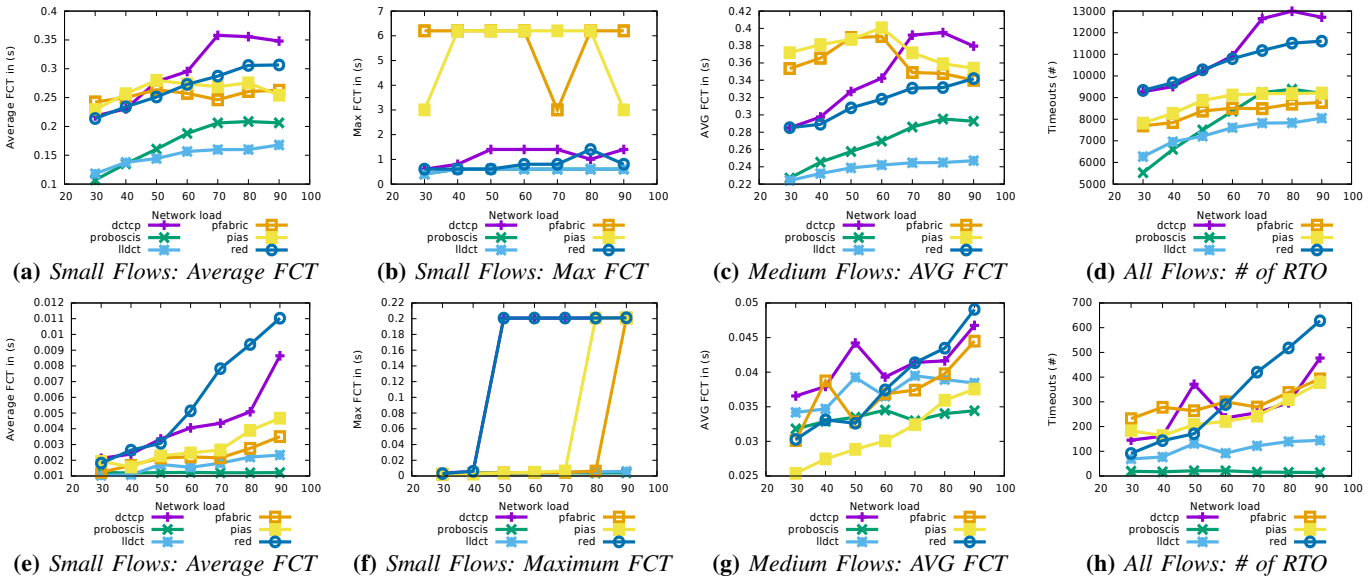
We will first study the performance of ProBoSCIS via simulation in relatively large data centers for different traffic loads and flow size distributions. For this we use ns2 and compare ProBoSCIS performance to that of the state-of-the-art schemes. Later, we discuss the implementation and experimental results from a deployment in a small test-bed.

We use a 4 spine-9 leaf topology using link capacities of 10 Gbps for the leaf (end-hosts) connections and per-hop link delays of 50 μs. The up-links are over-subscribed with a ratio of 1:5 (the typical ratio in current production data centers is in range of 3-20+).

We examine scenarios that cover Websearch and Datamining workloads, described previously, with and without background traffic. The flow size distribution for Websearch and Datamining 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 different arrival rates (λ) to simulate various network loads. The network load is varied (via $lambda$) in the range of (30% to 90%). For TCP, the minRTO is set to the default RTO_{min} of 200 ms and the initial congestion window size is 10 MSS, with a persistent connection used for successive requests. Finally, buffer sizes on all links are set to accommodate 100 packets. In ProBoSCIS, we set the number of probe packets γ to 10 and the ECN marking threshold to 20 packets. The performance metrics are the FCT for small flows and that of all flows, and the total number of timeouts experienced.

Figure 6 shows the average FCT and missed deadlines for small flows, average FCT for medium flows as well as the total timeouts for all flows in Websearch and Datamining workloads. We note that in both workloads small flows experience a dramatic increase in the FCT when they timeout, regardless of the mechanism in use – viz. congestion control and AQM (RED DCTCP), information-aware (LLDCT and pFabric), or information-agnostic packet scheduling (PIAS).

In contrast, ProBoSCIS helps small flows the most in improving their FCT by reducing the number of timeouts. We also note that overall FCT improves for all flows for two reasons: i) ProBoSCIS actually does not distinguish mice and elephant, thus, all flows benefit from its mechanisms, and ii) small flows finish quicker, leaving network resources for larger ones. We notice that Datamining workload, with almost 80% of the flows having a size of less than 10KB, experiences lesser timeouts overall, and DCTCP also can improve the FCT and reduce timeouts due to its ability to maintain a small queue. We have also conducted simulation experiments (not shown



(a) *Small Flows: Average FCT* **(b)** *Small Flows: Max FCT* **(c)** *Medium Flows: AVG FCT* **(d)** *All Flows: # of RTO*

(e) *Small Flows: Average FCT* **(f)** *Small Flows: Maximum FCT* **(g)** *Medium Flows: AVG FCT* **(h)** *All Flows: # of RTO*

Figure 6: *Performance metrics with various network loads in the range (30%, 90%) for Websearch and Datamining workloads*

here) to assess the sensitivity of ProBoSCIS to its parameter γ by varying the latter from 5 to 30 in steps of 5 and found ProBoSCIS to achieve good performance for all these values.

V. Linux Kernel Implementation

In this section, we discuss the implementation details of ProBoSCIS as a loadable kernel module in the Linux Kernel.

ProBoSCIS 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 [15] which is an integral part of Linux OS. Netfilter hooks attach to the data path in the Linux kernel just above the NIC driver. This imposes no modifications to the TCP/IP stack of the host OS nor guest OS and being a loadable kernel module, it allows for easy deployment in current production data centers. 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) [22]. 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 Algorithms 1 and 2.

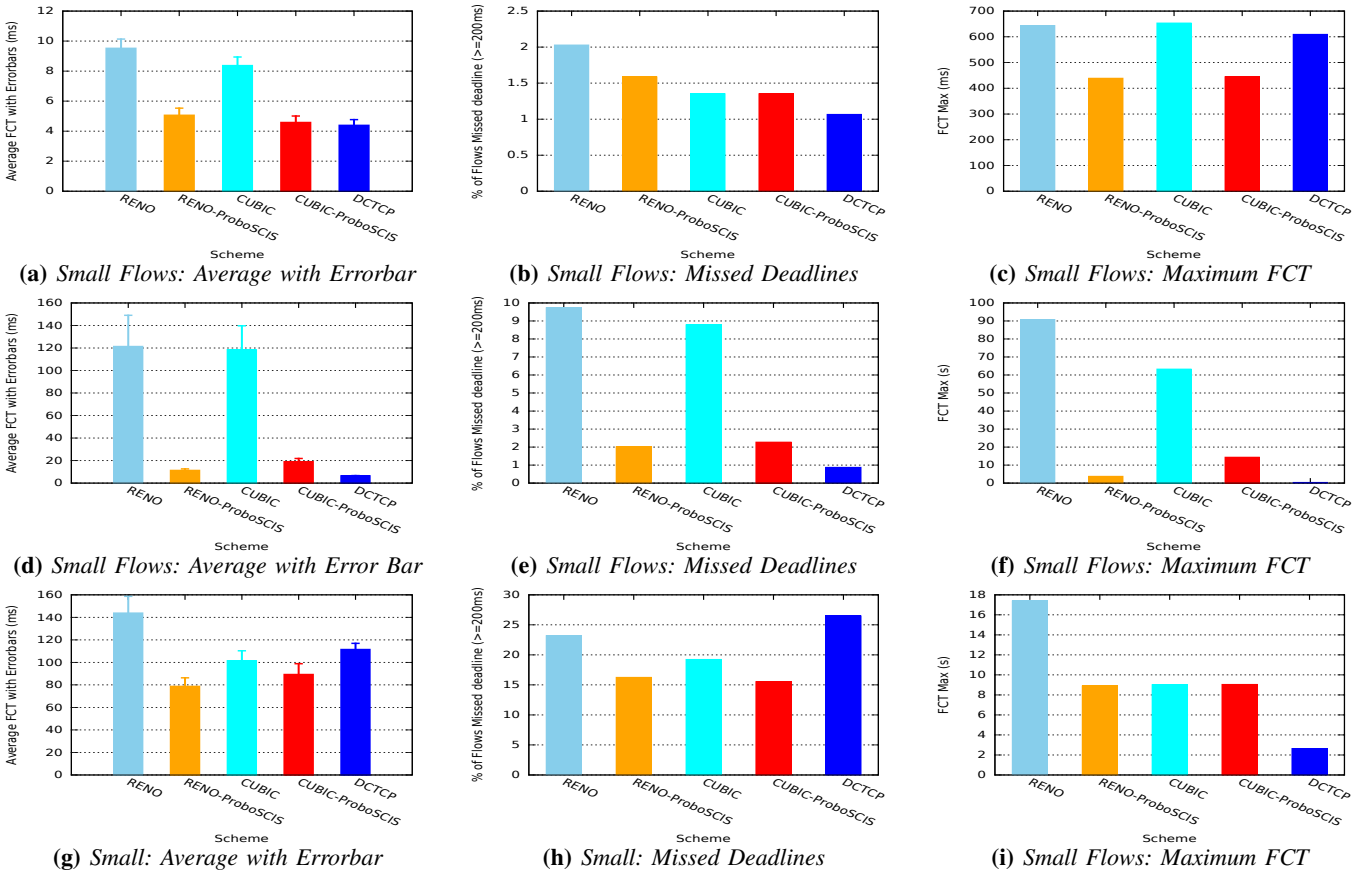
The module does not employ any packet queues to store the incoming packets, it only stores and updates flow entry states (i.e., ECN mark counts, arrival time and so on) on arrival of segments. Since ProBoSCIS does not require fine-grained timers in the micro-second scale, there is no overhead imposed on the end-host due to timer frequency interrupt handling. We collected various system load statistics during the experiments with ProBoSCIS and there was no noticeable increase in the load of the system due to ProBoSCIS. We could not replicate LLDCT because the authors did not make the code publicly available. Moreover, it requires floating-point operations not supported in kernel space (e.g., the division $k = \frac{W_c}{W_{max}}$). In addition, ProBoSCIS 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 ProBoSCIS as lightweight as possible.

VI. Testbed Evaluation

We use a small-scale testbed consisting of 84 virtual servers interconnected via 4 non-blocking leaf and 1 spine switches. The testbed cluster is organized into 4 racks (rack 1, 2, 3 and 4) as shown in Figure 8. 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 [23] which is installed on a desktop machine. The servers are loaded with Ubuntu Server 14.04 LTS with kernel version (3.18) which implements a full version of DCTCP protocol. The ProBoSCIS end-host module is invoked and installed on the host OS whenever necessary only. Unless otherwise mentioned, ProBoSCIS runs with the default settings (i.e., RTO of 4 ms and elephant threshold set to 100 KB[3]).

We again use the traffic generator to run the experiments with realistic traffic workloads. In addition, we have installed the iperf program [24] to emulate long-lived background traffic (e.g., VM migrations, backups) in certain scenarios. We setup different scenarios to reproduce a one-to-all and all-to-all, with and without background traffic. In the one-to-all scenario, clients running on the VMs in one rack send requests randomly to any of all other servers in the cluster. While in the all-to-all scenario, all clients send requests to any of all other servers in the cluster. If background traffic is

[3]These two parameters aren't shown in the algorithm for brevity. The RTO is used to recover from lost PPs. The elephant threshold is an addition to identify and excluded elephant flows from the window adjustment to reduce processing overhead and keep the flow table small. The rationale is that such flows usually are handled well by TCP via 3 DUPACKs as they have a large enough flight size.

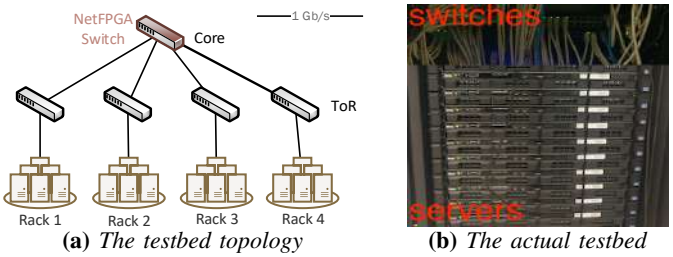


(a) *Small Flows: Average with Errorbar* **(b)** *Small Flows: Missed Deadlines* **(c)** *Small Flows: Maximum FCT*

(d) *Small Flows: Average with Error Bar* **(e)** *Small Flows: Missed Deadlines* **(f)** *Small Flows: Maximum FCT*

(g) *Small: Average with Errorbar* **(h)** *Small: Missed Deadlines* **(i)** *Small Flows: Maximum FCT*

Figure 7: *Scenarios without background traffic: (a-c) onet-to-all Websearch, (d-f) one-to-all Datamining and (g-i) all-to-all Websearch*



(a) *The testbed topology* **(b)** *The actual testbed*

Figure 8: *Testbed evaluation of ProBoSCIS in a small data center*

introduced, we run continuous long-lived iperf flows in an all-to-all fashion to evaluate ProBoSCIS under sudden and persistent network load spikes. We classify flows with sizes: (small, $<= 100KB$), (medium, $> 100KB$ *and* $<= 10MB$) and (large, $>= 10MB$). The senders are created by creating multiple virtual ports at the end-hosts and then an iperf or Apache client/server process is binded to the vport. This allow us to emulate traffic originating from any number of VMs to ease creating scenarios with large number of flows in the network.

The objectives are: *i)* to verify if ProBoSCIS, helps short TCP flows finish faster with more flows meeting their deadlines. *ii)* to verify how ProBoSCIS improves the performance of tail-end flows whose completion determines the quality of the result; *iii)* to quantify ProBoSCIS robustness in unexpected network loads (i.e., background traffic).

A. Experimental Results and Discussion

One-to-All scenario without Background Traffic: We report the the average FCT for small flows and that of 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 7a, 7b and 7c show the average FCT with errorbars, missed deadlines and tail-latency (max) for small flows in Websearch workload, respectively. While, Figures 7d, 7e and 7f, show the same metrics for the Datamining workload. We can make the following observations: *i)* for all workloads, ProBoSCIS helps small flows regardless of the TCP variant in use, on both the average and variation of FCTs. Compared to Reno and Cubic, ProBoSCIS reduces the average FCT of small flows by $\approx (47\%, 46\%)$ for Websearch and $\approx (91\%, 84\%)$ for Datamining. The achieved FCT of Reno and Cubic with ProBoSCIS is quite comparable to that of DCTCP (yet this latter must modify the TCP stack in the VM and can achieve this feat only in homogeneous all DCTCP networks); *ii)* moreover, for Websearch workload, ProBoSCIS reduces the missed deadlines of Reno for short flows by $\approx 22\%$. The missed deadlines are marginally comparable in this case among the

different TCP variants. However, for Datamining workload, ProBoSCIS improves the two metrics by $\approx (80\%, 75\%)$ respectively, which closes the missed-deadline gap with DCTCP to only $\approx (10\%, 15\%)$ for Reno and Cubic, respectively. *iii)* ProBoSCIS further improves the tail-end (i.e., maximum) FCT of small flows. The improvements in the max FCT for Reno and Cubic with ProBoSCIS are $\approx (32\%, 32\%)$ and $\approx (96\%, 90\%)$ in Websearch and Datamining workloads, respectively. The improvements are significant for Datamining because ProBoSCIS can spare short flows much waiting for long RTOs.

All-to-All scenario without Background Traffic: In this test run, all end-points are set to communicate with all other end-points to evaluate the performance of ProboSCIS in such complex traffic pattern. In this case, we do not start the background traffic. We report similar metrics as in the aforementioned cases. Figs 9a, 9b and 9c show the average FCT with errorbars, missed deadlines and tail-latency for small flows in the Websearch workload. The results suggest that ProBoSCIS still improves the average FCT of small flows for Webearch workload, regardless of the TCP congestion control in use. As shown compared to Reno and Cubic, ProBoSCIS reduces the average FCT of small flows by $\approx (95\%, 96\%)$ and their missed deadlines by $\approx (34\%, 43\%)$ for Websearch workload. Moreover, the performance of all flows (including medium and large) is improved by $\approx (50\%)$ compared to Reno and the same tail performance for Cubic. In this scenario, DCTCP misses its target by showing a large average FCT and a large number of flows that missed their deadlines.

One-to-All scenario with Background Traffic: to put ProBoSCIS under a true stress, we run the same one-to-all scenario with all-to-all background traffic that shares the network with the test workload. We report similar metrics as previously. Figures 9a, 9b, 9c and 9d show the average FCT, missed deadlines, maximum FCT for small flows as well as average FCT for all flows in the Websearch workload. The results suggest that ProBoSCIS still improves the average FCT of small flows for Webearch workload regardless of TCP congestion control in use. As shown compared to Reno and Cubic, ProBoSCIS reduces the average FCT and missed deadlines of small flows by $\approx (95\%, 96\%)$ for Websearch workload. The tail-end performance of small TCP flows are also improved by a factor of $\approx (80\%, 87\%)$. Not only mice flows benefit in this scenario as shown by the figures, the performance of all flows (including medium and large) is also improved by factors of $\approx (81\%, 85\%)$. Moreover, DCTCP can not perform well in scenarios where background traffic exists in the network. For instance, Cubic with ProBoSCIS can improve over DCTCP on average and max FCT of small flows by $\approx (85\%, 88\%)$, on missed deadlines by $\approx 76\%$ and on average FCT of all flows by $\approx 92\%$. This is significant.

In summary, the simulations and experimental results show the performance gains achieved by ProBoSCIS are especially welcome for small flows, that usually constitute the lion's share in data center traffic. In particular, they show that: 1) ProBoSCIS minimizes the variance of small TCP flows completion times and reduce the missed deadlines; 2) it can maintain its gains even in the presence of heavy background traffic; and 3) ProBoSCIS can handle various workloads regardless to the TCP variant in use.

VII. 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 centers [25–27].

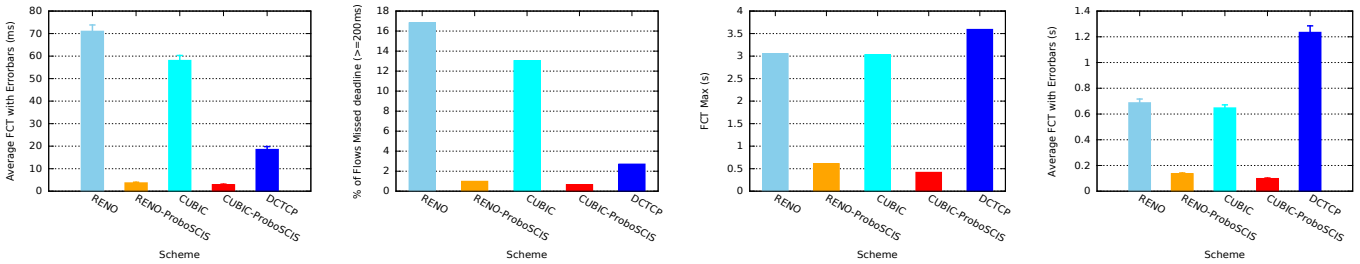
Specifically, [5] showed that frequent timeouts can harm the performance of latency-sensitive applications. Numerous solutions have been proposed. These fall into one of four main approaches. The first mitigates the consequence of long waiting times of RTO, by reducing the default minRTO to the range 100 μs - 2 ms [5]. However, while very effective, this approach affects the sending rates of TCP by forcing it to cut the congestion window to 1; it relies on a static minRTO value which can be ineffective in heterogeneous networks; and it imposes modifications to TCP stack on the 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 VMs [6], or using receiver window based flow control [7] or finally by deploying global traffic scheduling [11–13]. These works achieved their goals and have shown they could improve 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 various parameters and sometimes require application-side information.

The third approach is to enforce flow admission control to reduce TimeOut probability. [28] 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 by means of fast retransmit rather than waiting for long timeout.

TCP-PLATO [26] proposed changing TCP state-machine to tag specific packets using IP-DSCP bits which are preferentially queued at the switch to reduce their drop-probability enabling DUPACKs to be received to trigger FRR instead of waiting for timeout. Even though TCP-PLATO is effective in reducing 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 logic of sender and receiver.

Recent works [29, 30] also proposed implementing DCTCP in the hypervisor to unify all TCP variants in data centers. However, this approach tracks full DCTCP state information and implements full state machine in the hypervisor. On the other hand, ProbOSICS tries to minimize such overhead by tracking the minimal amount of necessary state information. Moreover, it implements only the TCP retransmission mechanism.



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

Figure 9: *Performance metrics of one-to-all scenario using Websearch workload with background traffic*

VIII. Conclusion and Future Work

In this paper, we proposed ProBoSICS, a probing mechanism to examine congestion level during the connection establishment phase prior to transmitting data. This allows ProBoSICS to curb RTOs that are the result of various loss events including SSL or CSL in highly congested network. Simulation and test-bed implementation experimental results demonstrate that ProBoSCIS is effective in improving the performance of TCP flows in data centers and is practical.

References

[1] J. Dean and S. Ghemawat, "MapReduce : Simplified Data Processing on Large Clusters," *CACM*, pp. 1–13, 2008.

[2] Apache.org, "Spark: Lightning-fast cluster computing," http://spark.apache.org.

[3] T. Hoff. Google: Taming The Long Latency Tail - When More Machines Equals Worse Results. Http://highscalability.com/blog/2012/3/12/google-taming-the-long-latency-tail-when-more-machines-equal.html.

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

[5] 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," in *SIGCOMM*, 2009.

[6] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center TCP (DCTCP)," in *SIGCOMM*, 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] 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 *SIGCOMM*, 2015.

[9] S. M. Irteza, A. Ahmed, S. Farrukh, B. N. Memon, and I. A. Qazi, "On the coexistence of transport protocols in data centers," in *Proceedings of IEEE ICC*, 2014. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=6883814

[10] I. A. Rai, G. Urvoy-Keller, M. K. Vernon, and E. W. Biersack, "Performance analysis of las-based scheduling disciplines in a packet switched network," *SIGMETRICS Perform. Eval. Rev.*, vol. 32, no. 1, pp. 106–117, 2004.

[11] 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*, 2013.

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

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

[14] 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 *SIGCOMM*, 2009.

[15] NetFilter.org. NetFilter Packet Filtering Framework for linux. Http://www.netfilter.org/.

[16] OpenvSwitch.org. Open Virtual Switch project. Http://openvswitch.org/.

[17] J. Postel. (1981) RFC 793 - Transmission Control Protocol. Http://www.ietf.org/rfc/rfc793.txt.

[18] Pica8. Pica8 Pronto-3295 switch technical specifications. Http://www.pica8.com/documents/pica8-datasheet-48x1gbe-p3290-p3295.pdf.

[19] RED, "Red parameters setting." [Online]. Available: http://www.icir.org/floyd/red.html#parameters

[20] C. Hollot, V. Misra, D. Towsley, and Wei-Bo Gong, "A control theoretic analysis of RED," in *INFOCOM*, 2001.

[21] 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.

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

[23] netfpga.org. NetFPGA 1G Specifications. Http://netfpga.org/1G_specs.html.

[24] iperf. The TCP/UDP Bandwidth Measurement Tool. Https://iperf.fr/.

[25] 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.

[26] S. Shukla, S. Chan, A. S.-W. Tam, A. Gupta, Y. Xu, and H. J. Chao, "TCP PLATO: Packet Labelling to Alleviate Time-Out," *IEEE JSAC*, pp. 65–76, 2014.

[27] W. Chen, F. Ren, J. Xie, C. Lin, K. Yin, and F. Baker, "Comprehensive understanding of TCP Incast problem," in *in proceedings of INFOCOM*, 2015.

[28] 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.

[29] 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.

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