Do Switches Still Need to Deliver Packets in Sequence?

Ufuk Usubütün, Fraida Fund, Shivendra Panwar
Department of Electrical and Computer Engineering
NYU Tandon School of Engineering
Brooklyn, NY 11201
{usubutun, ffund, panwar}@nyu.edu

***Abstract*—Internet switches become harder and costlier to build for higher line rates and switch capacities. In-sequence delivery of packets has traditionally been a constraint on switch designs because TCP loss detection was considered vulnerable to out-of-sequence arrivals. For this reason, extremely efficient and simple designs, such as the Load Balanced Birkhoff-von Neumann Switch, were considered impractical. However, we reevaluate this constraint considering modern TCP implementations with loss detection algorithms like Recent Acknowledgment (*RACK*) that are more resilient to out-of-order arrivals. In a set of testbed experiments representative of wide area core networks, we evaluated the performance of TCP flows traversing a load balanced switch that reorders some packets within a flow. We show that widely deployed and standard TCP implementations of the last decade achieve similar performance when traversing a load balanced switch as they do when there is no reordering. Furthermore, we also verified that an increase in the line rate leads to favorable conditions for time based loss detection methods, such as the one used in *RACK*. Our results, if further validated, suggest that switch designs that were previously thought to be unsuitable can potentially be utilized, thanks to the relaxation of the in-sequence delivery constraint.**

***Index Terms*—TCP, Switching, packet reordering, dupthresh, RACK**

I. Introduction

The conventional wisdom in network engineering for the last several decades was that routers and switches should, whenever possible, avoid reordering packets within a flow. This is because classical TCP inferred packet loss using a triple duplicate acknowledgement (ACK) rule, which rendered it vulnerable to any alterations in the packet arrival order. A packet that was delayed by more than 3 positions within its flow would be erroneously marked as lost and trigger a recovery response, typically causing unnecessary packet retransmissions and unneeded congestion window (*cwnd*) reductions [1].

This has historically led to a stringent in-order delivery requirement for all packet switch designs, either forcing more complex designs, or requiring the incorporation of output resequencing buffers [2], [3]. Many powerful switch designs were considered impractical because they resulted in out-of-order packet sequences [4]. As a motivating example, consider the Load Balanced Birkhoff-von Neumann switch [5], an extremely simple, scalable and efficient architecture featuring a load balancer that spread incoming packets into parallel virtual output queues (VOQs) before the switching stage. While the idea was masterly, the design resulted in packets that belong to the same TCP flow to be spread into independent parallel queues. As a result, these packets experienced different wait times and potentially left the switch out-of-sequence. This led to extensive work on ensuring in-sequence delivery for this switch [6], [7]. Some modern switch architectures today feature parallel paths for processing packets [8], yet they are designed to bring packets back into sequence at the cost of additional complexity and delay, explicitly motivated by the effect of reordering on TCP.

This research was supported by the New York State Center for Advanced Technology in Telecommunications and Distributed Systems (CATT), NYU Wireless, the Cisco University Research Program Fund and by the National Science Foundation (NSF) under Grant No. CNS-2148309 and OAC-2226408.

However, two relevant aspects of the Internet related to packet reordering have evolved in noteworthy ways.

Advanced loss detection algorithms that are resilient to bounded amounts of reordering or delay variation appeared and became widespread: The adoption of network load balancing and multi-path routing approaches [9] and emergence of lossy wireless links and link layer retransmissions [10], both of which increase the frequency with which packets are reordered, led TCP developers to make significant improvements on ways to differentiate real packet loss from reordering by deploying new recovery algorithms [11]. Two instances of advanced recovery algorithms appeared and are widely deployed. These were *dupthresh* [12] with an adaptive threshold heuristic (e.g. [13], [14]) and Recent Acknowledgement (*RACK*) [15].

Adaptive versions of the *dupthresh* algorithm introduced reactive ways to grow the duplicate ACK counting threshold to accommodate bounded levels of packet reordering. *RACK* on the other hand, diverged from previous protocols as it followed a temporal approach that detected loss using the lateness of individual packets while mostly disregarding the order in which they were delivered. This temporal approach made it possible for *RACK* to be able to accommodate reordered packets without a reactive adaptation, given the delay variation was bounded within short time frames.

Although *RACK* is used by default in modern Linux and Windows systems, it is not well represented in the academic literature. One recent paper [16] evaluates the performance of *RACK* in comparison with the fixed threshold *dupthresh* for artificially generated reordering patterns, but otherwise it has

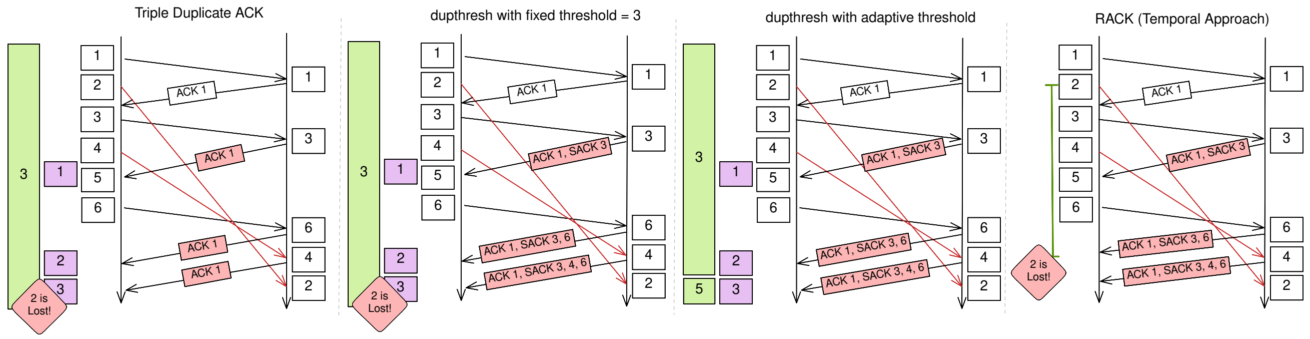


Fig. 1. Comparison of recovery algorithms when packet number 2 is delayed within its sequence. Green and purple boxes for the ordinal methods represent the detection threshold and the current count for loss. The green line for the temporal approach represents the time threshold in use. Fixed threshold ordinal approaches declare 2 as lost when the threshold is surpassed. Adaptive dupthresh uses SACKs to grow its threshold: The arrival of SACK 4, fills a 'hole' in the sequence of ACKed packets and triggers a detection threshold update as described. Finally, *RACK* uses the time window to declare 2 as lost.

not been well studied. We suspect that as a consequence, the academic literature studying packet reordering, network load balancing and switching largely overlooks that the most widely deployed TCP algorithms today are tolerant to some degree of reordering, and continue to assume the widespread use of the triple duplicate ACK rule.

Network capacities grew from tens of Mbps to hundreds of Gbps: We noted that *RACK* uses a temporal method that accommodates delay variations within short time frames. Classical queuing theory suggests that, for an M/M/1 queuing system, with the same utilization, an increase in the line rate has an inverse relationship with the expected delay of a packet [17]. We can also show that this also leads to a narrower delay distribution. For a scenario with independent parallel queues, such as in a load balancer, this means, as the line rate goes up, it is more likely for packets in parallel queues to experience delays that are smaller and therefore similar. Thus, the delay spread of packets reordered by going through such a system is likely to narrow as line rates increase.

These changes motivate us to reevaluate the assumption that switch designs should deliver packets strictly in-sequence, even at the cost of additional complexity. In this work, we first consider the implications of the above mentioned developments, especially in the context of the reordering produced by a load balanced switch. Then, we conduct a series of experiments on the CloudLab testbed [18] to evaluate the performance of modern TCP loss detection algorithms in an environment typical of a high-capacity core network switch. The results indicate that under certain conditions, the performance of TCP with *RACK* (the current default loss detection protocol) through a load balanced switch is similar to the performance of an equivalent switch without reordering.

This work suggests that the traditional wisdom of in-order delivery requirements might be outdated, and demonstrates that reordering produced by load balanced switches can be much better tolerated at today's high line rates thanks to the use of time based approaches. Future work on characterization and deeper understanding of reordering behavior may open the door to new switch fabric designs that were previously considered impractical for this reason. It can potentially have similar implications for data center networks and wireless communication protocols.

The rest of this paper is organised as follows: In Section II we look at how TCP evolved in how it detects loss. In Section III we consider the implications of increasing line rates for Load Balanced Switches and their delay characteristics. In Section IV we experimentally validate our expectations and in Section V we discuss the results we obtained and comment on the implications of our work and paths to future work.

II. The Evolution of TCP Loss Detection

While the literature on network switch design focused on producing devices that delivered packets in-sequence with the assumption that triple duplicate ACK algorithm was in use, TCP implementations saw significant changes on how reordering is handled. Recovery algorithms, charged with detecting packet loss and maintaining a (re)transmit queue, were decoupled from congestion control algorithms and developed independently [11]. While the advancements in congestion control are well-represented in the academic literature, there is little published work on the developments in loss recovery. In this section, we discuss the fundamental ideas behind these new loss recovery algorithms (omitting some minor details of the algorithms and heuristics in the implementations).

Widely deployed recovery algorithms, for until a decade ago, were built on the original idea used by the triple duplicate ACK rule. They followed ordinal approaches [10] that counted notifications of non-reception and assumed loss in case this count exceeded a threshold. For the classical triple duplicate ACK rule, the threshold was fixed to 3 and cumulative ACKs (cumACK) were used for counting. In the early 2000s, an algorithm widely referred to as *dupthresh* extended the ACK counting idea with the use of Selective Acknowledgements (SACK) [12]. In case there were 'gaps' in the sequence of received packets, SACKs allowed the receiver to explicitly notify which segments were received rather than sending repeated cumACKs. Adjusting to this new information, *dupthresh* designated a segment s as lost, if more than the threshold amount of segments were SACKed above s.

Dupthresh with a fixed threshold, however, did not provide extra protection from issues stemming from reordering. In real implementations *dupthresh* was implemented with adaptive threshold heuristics which were able to detect reordering and adjust the threshold. These heuristics are generally built on two premises: 1) If a flow experiences reordering, it is likely to experience it again. 2) The extent of packet reordering is likely to be comparable to the extent in the past. For example, the Linux kernel detects the existence of reordering in case a higher sequence is delivered (i.e. SACKed) before some lower never-retransmitted sequence and uses the distance between the highest SACKed segment and the highest cumulatively ACKed segment as an estimate of the extent of reordering. This distance value is then used as the new threshold value. A comparison of the algorithms mentioned are provided in Figure 1. Further details of adaptation heuristics are implementation specific [19] and will be omitted.

The ordinal approach to loss detection with adaptive threshold heuristics has two shortcomings with respect to reordering resilience: i) Adaptive threshold growth is reactive and requires a number of iterations to accommodate the reordering extent, making it a reordering-tolerant but not a resilient method. ii) The adaptive threshold value is usually upper bounded (e.g., by 300 in Linux) since keeping large thresholds significantly delays detection of actual loss. These limitations are partially mitigated by a temporal approach, which we will discuss next.

Temporal approaches to loss recovery break away from the notion of counting missing deliveries and instead function by keeping a timer for each packet transmission. They designate a packet as lost, in case it is not (S)ACKed within a window of time. The practice of individually timing acknowledgements implies that the order in which packets are received is no longer relevant. Recent Acknowledgement (*RACK*) [15] is a temporal loss detection algorithm which appeared in mid-2010s and is now the default way to detect loss in Linux and Windows [20]. *RACK* uses a time window called `rack_timeout` which is calculated by the following relation where both terms in the summation are dynamic.

$$\texttt{rack\_timeout} = \texttt{srtt} + \texttt{reo\_wnd} \qquad (1)$$
$$\text{where} \quad \texttt{min\_rtt/4} \leq \texttt{reo\_wnd} < \texttt{srtt}$$

`srtt` and `min_rtt` represent values of smoothed RTT and the minimum RTT maintained by the kernel. `srtt` is a quantity that averages the RTT at the time of transmission as it is updated very frequently; `min_rtt` is the lowest round-trip delay measured so far. `reo_wnd` is a variable controlled by *RACK* which is set to its minimum value of `min_rtt/4` by default. It is gradually grown only if the sender receives Duplicate SACKs (DSACK) that are are explicit notifications of unnecessarily retransmitted packets. Temporal approaches are particularly powerful over the adaptive packet counting approaches for the following reason: If reordering occurs in core network switches with very high line rates, the extent of reordering is likely to be very large in terms of packet displacement, requiring a loss to be assumed and the packet counting threshold to adapt in order to accommodate. However, the same reordering is small in terms of delay variation (we elaborate on this Section III), and would likely be within the existing `reo_wnd` without requiring adaptation. The adaptive `reo_wnd` growth mechanism is also deliberately designed to be slow and is reset to its smallest value after 16 recovery episodes by the designers to make sure detection of an actual loss is not significantly delayed. Therefore it is desirable for any connection to have delay variation within `rack_timeout` with the smallest value of `reo_wnd`.

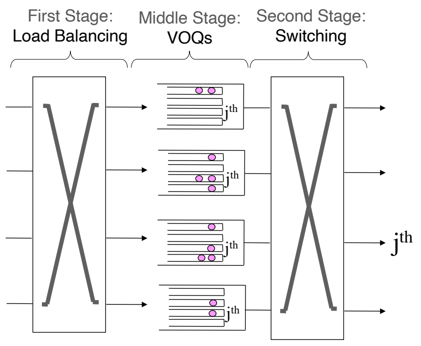


Fig. 2. Load Balanced Switch of size $N = 4$

Observe that `rack_timeout` is of the order of the RTT. If we consider flows going through core switches located on a backbone network, we can easily expect their RTTs to be on the order of tens of milliseconds, if not larger due to queuing delays experienced in access networks. On the other hand, assume we have a switch architecture that produces reordering on this path. If we can guarantee that delay variation produced by this switch on consecutive packets of the same flow is much smaller compared to the RTT, the switch is then likely to have little to no impact on TCP performance when *RACK* is used.

III. Impact of Core Network Capacity Growth on Load Balanced Switches

With this context of new TCP loss detection algorithms, and especially the current temporal approach to loss detection, we now seek to characterize the reordering that may be induced in TCP flows by a load balanced switch.

As a motivating example, we consider the Load Balanced Birkhoff-von Neumann (LB-BvN) Switch, a simple, scalable and efficient switch architecture proposed by Chang et al. [5] that does not require a scheduler. Its simplicity is especially desirable for devices in core networks requiring extremely fast operations. The architecture, as shown in Fig. 2, includes two stages of crossbars that forward packets in round-robin fashion, and virtual output queues (VOQ) in the middle stage. The first stage uniformly spreads packets into the middle stage achieving load balancing while the second stage achieves switching. The middle stage contains VOQs for each output.

While the operation and performance metrics of the switch is beyond the scope of this paper, the parallel VOQs in the middle stage are key to the study of packet reordering. The fact that consecutive packets from the same flow from input i to output j are assigned to independent queues by the load balancing stage and experience different queuing delays creates a potential for packet reordering that could undermine

TCP performance using the triple duplicate ACK rule. The extent of potential reordering and its mitigation by using properly sized output resequencing buffers was studied in [21].

However, if we consider a temporal recovery algorithm such as *RACK*, then the TCP performance would be determined by the variation of wait times of consecutive packets of the same flow at the individual parallel queues. As discussed earlier, if the variation of the delays caused by the load balancing stage is small enough compared to the total end-to-end RTT, the impact on the performance of TCP employing *RACK* will be negligible. The fact that backbone network line rates increased significantly provides a favorable condition for temporal methods as this has an impact on the probability distribution of total wait time experienced at the middle stage VOQs. Let us review how delays through the switch are affected by line rate according to queueing theory.

Mean Delay of a Queuing System: The performance results for the M/M/1 queue suggest that increasing both arrival and service rates by scaling them up by a factor K, that is increasing the line rate while keeping the utilization constant, results in the same average queue occupancy distribution. However, it causes the average wait time to be scaled down by a factor of K.

Tail Delay of an M/M/1 Queue: For an M/M/1 queue, let W be the random variable capturing the total wait time of a packet through the system. The tail probability of W can be expressed with the following relation where λ and μ are the arrival and services rates [17]. If we consider a line speedup by a factor of K over arrival and service rates for base line rates, λ_0 and μ_0, we get.

$$P(W \geq \tau) = e^{-(\mu-\lambda)\tau} = e^{-K(\mu_0-\lambda_0)\tau} \tag{2}$$

Thus increasing the line rate K, also compresses the queue delay tail distribution by a factor of K.

Implication for Parallel VOQs: Assume we have a load balancer as in Fig. 2 that uniformly spreads all arrivals for the j^{th} output into N parallel VOQs working at $1/N$ times a line rate K times the base line rate. Let us simply model the j^{th} VOQs using independent and identical M/M/1 queues, each with total delay W_i:

$$P(W_i \geq \tau) = e^{-\frac{K}{N}(\mu_0-\lambda_0)\tau}, \quad i \in \{1, 2, \cdots, N\} \tag{3}$$

Assume that a burst of N packets destined to the same output arrive at a given time and that they are all placed in N separate VOQs. In that case, the delay experienced by the last packet to leave can be expressed as $Y = \max\{W_1, W_2, \cdots, W_N\}$. Y can be computed as:

$$\begin{aligned} P(Y < \tau) &= P(W_1 < \tau, W_2 < \tau, \cdots, W_N < \tau) && (4)\\ &= \prod_{i=1}^{N} P(W_i < \tau) && (5)\\ &= (1 - e^{-\frac{K}{N}(\mu_0-\lambda_0)\tau})^N && (6)\end{aligned}$$

where (5) follows from independence. Rearranging terms, we can obtain the following expression that describes the time window $\hat{\tau}$ needed so that of all packets complete service within $\hat{\tau}$ with probability p.

$$\hat{\tau} = -\frac{N ln(1 - \sqrt[N]{p})}{K(\mu_0 - \lambda_0)} \tag{7}$$

We can see that a K-fold increase in the line rate leads to a K times smaller time window $\hat{\tau}$, as compared to the base line rate, for the now randomly ordered burst to leave the switch with the same probability p. At speeds of hundreds of Gbps that are seen today in core networks, this model suggests, the resulting delay variation spread would shrink compared to the aggregate end to end propagation, processing and other queuing delays incurred by the packets.

Note that this analytical argument assumes a packet arrival process that is independent of the delay or reordering encountered at the switch. In reality, TCP produces closed loop traffic that depends on both packet reordering and delay. The arrival process, further, will deviate from a Poisson process, depending in part on the number of flows it is interleaved with. This, therefore, requires an experimental validation of the result we derived using classical M/M/1 systems.

IV. Experimental Validation

To evaluate the performance of current TCP implementations when flows traverse a load balanced switch, we designed and executed a sequence of experiments on the CloudLab [18] testbed. In this section, we describe in detail our experimental evaluation and its results. We also make available[1] all of the experimental artifacts necessary (including resource profile, source code for experiment execution and data analysis, and other materials) so that others may reproduce our results.

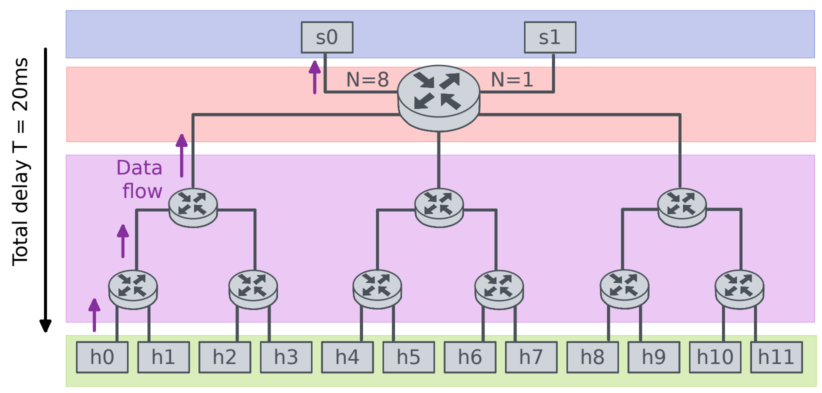
A. Experiment setup

Topology: Our experiment topology, as realized on real bare metal servers and Ethernet links on CloudLab, is illustrated in Fig. 3. The direction of data flow, as shown in the diagram, is from the bottom toward the top of the topology. In the bottom-most level (green-shaded area in diagram), flows are generated at twelve end hosts. Next, these flows are mixed at a sequence of intermediate routers (purple-shaded area in diagram). The last router along the path (red-shaded area in diagram) represents the switch where we evaluate the impact of load balancing; on leaving this router, depending on their destination, data flows may traverse an egress interface with 8 parallel queues, or an egress interface with 1 queue. Finally (blue-shaded area in diagram), two more hosts serve as the endpoints for flows traversing the router's 8 queue interface and 1 queue interface, respectively.

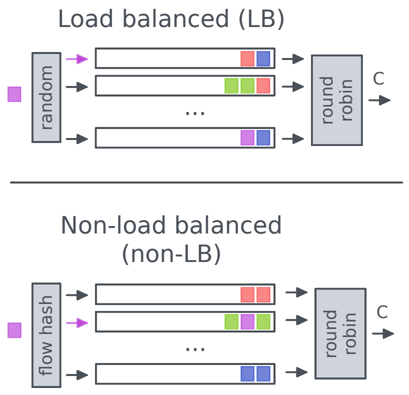
Base delay: To represent an Internet scenario where flows experience some base delay (besides queuing delay), we set a static delay of $T = 20$ms for all flows by using `netem` to add $T/4$ to each interface on the *reverse* path of data flow.

TCP endpoints: The hosts representing TCP endpoints, shown as `h0-h11`, `s0`, and `s1` in Fig. 3, are bare-metal

[1]All of the experiment materials are available in the following repository: http://github.com/ufukusubutun/Reordering_Switch.



(a)



(b)

Fig. 3. Details of our experiment setup, including (a) an overview of the experiment topology, and (b) an illustration of the two alternative realizations of parallel queues used in the experiment.

servers running Ubuntu 20.04 with Linux Kernel v5.4, using the TCP CUBIC congestion control algorithm. Depending on the experiment, we configure these to use one of the three of the loss detection algorithms described in Section II - either *dupthresh* with a fixed threshold of 3, *dupthresh* with an adaptive threshold, or *RACK*.

Main router: The router that is the main object of this study is a bare metal server running Ubuntu 20.04, and forwards data flows to `s0` and `s1` through two egress interfaces. The interface that forwards traffic to `s1` has a single FIFO queue, with a service rate of C and buffer size of 2 BDP, i.e. $2T \times C$. The interface that forwards traffic to `s0` has $N = 8$ parallel queues. In the load balanced (LB) configuration (which we realize using `iptables` with `tc-htb`), each of the parallel queues has a buffer size of 2 BDP/N, and each arriving packet is placed randomly into one of the queues, with uniform probability. In the non-load balanced (non-LB) configuration (which uses `tc-fq`), flows are hashed to queues so that packets belonging to the same flow will always be placed in the same queue, and the total queue size is limited to 2 BDP. The two configurations are illustrated in Fig. 3. In either case, queues are served on a round robin schedule with a quantum size of two full-sized packets, and total service rate C.

Intermediate routers for mixing: The intermediate routers where flows are mixed are realized as bare-metal servers running Ubuntu 18.04, with IPv4 forwarding enabled and static routing rules to forward traffic flows toward either `s0` or `s1`. On each of these, the egress interface is a FIFO queue with a service rate of $5C/3$, and a buffer size of 2 BDP, i.e. $2T \times 5C/3$.

Flow generation: To represent realistic Internet traffic, flow generating processes at `h0` through `h11` continuously generate TCP traffic to either `s0` or `s1` using `iperf3`, with a random flow size sampled from a wide area network TCP traffic model [22]. Half of the generators connect to `s0` and the other half to `s1`. To decouple the network utilization from the behavior of the TCP loss detection protocol (which affects flow completion time), the time from the start of one flow to the start of the next flow is random, with a mean wait time that is scaled with respect to the bottleneck line rate C in order to avoid overlapping flows in the same flow generating process. To scale the level of traffic in the network, we vary F, the number of flow generating processes at each host endpoint.

Measurement and data analysis: We capture packet headers at each ingress/egress interface of the main router. By analyzing the traces and matching individual packets at ingress and egress, we can compute flow-level statistics including flow completion time and retransmission ratio, as well as packet level statistics such as per-packet delay through the switch and reordering metrics. We use the metrics called *displacement* and *reordering density* as defined in [23] to measure reordering. The *displacement* metric captures packet level displacement of each segment by comparing the sequences entering and leaving the system. If segments A, B, C, D enter the switch where D enters last and they leave the switch with the order A, D, B, C, the corresponding *displacements* $d(\cdot)$ for each packet are $d(A) = 0$, $d(B) = 1$, $d(C) = 1$ and $d(D) = -2$. If a packet arrives in the correct order, its *displacement* would be 0, it would be negative if it arrives early, and positive if it arrives late. We are interested in late arrivals, as those are the packets that trigger a recovery response. The *reorder density* metric is a discrete probability distribution of frequency of packets with respect to their *displacements*. We calculate the *displacement* of each packet within the same TCP flow and then calculate the *reordering density*.

B. Results

In the series of experiments we present here, we aim to evaluate the effect of reordering produced by the load balancer as a function of line rate. We therefore ran experiments at different line rates $C \in \{1, 2, 4.5\}$ Gbps while keeping the number of flow generators fixed at $F = 150$. Our experiments only go up to $C = 4.5$ Gbps as this is the highest line rate at which we can reliably capture packet headers on this experiment infrastructure. In Fig 4(a-d)&(f) we present the flow completion time and the ratio of retransmissions to all transmissions as CDFs over all flows larger than 1 MB for all algorithms. For better visual representation of the results, we only show long flows as shorter flows were not affected by significant packet reordering and did not suffer the effects of reordering on TCP. The results are labeled as *RACK*, *adapThresh* and *3Thresh* for *RACK*, *dupthresh* with an adaptive threshold and *dupthresh* with a fixed threshold of 3, respectively. Furthermore, each plot shows results when the main router is in the load balanced (LB) configuration and when it is in the non-load balanced (non-LB) configuration. During the experiments, use of $C = 1, 2$ and 4.5 Gbps with $F = 150$ flow generators resulted in average link utilizations of approximately $(51 \pm 4)\%$ at the emulated switch output for all scenarios. Due to the closed loop nature of TCP it is not possible to generate an exact target utilization, because flow completion times and retransmissions affect link utilization. For meaningful comparison across line rates, we choose to compare the case of an equal number of flow generators that follow the same flow size trace, while the experiment duration is scaled inversely proportional to the line rate. This allows us to test the same flow trace over different line rates.

In Fig 4(a-d)&(f), observe that all three algorithms do equally well when the non-LB configuration is in use, that is when there is no reordering in the network. However, there are significant differences in how the algorithms perform in the LB configuration. We see that *RACK* achieves a performance comparable with non-reordered flows as the line rate is increased, and is only modestly inferior at the lower line rates of 1 Gbps. *adapThresh* is somewhat inferior to *RACK* in the LB scenario, with *3Thresh* showing significantly worse performance. The results also show a significant performance improvement when using current methods over the classical fixed threshold of 3. For the case of $C = 4.5$ Gbps under the LB configuration, the median flow completion time of *3Thresh* is 1.59 times that of *RACK*. This is strongly related to the fact that the median flow running *3Thresh* experiences approximately 189 times the retransmission ratio of *RACK* (see Fig. 4(f)).

In Fig 4(e) we show the distribution of delay experienced by packets traversing the switch for different line rates when *RACK* was in use with the LB configuration. In line with our expectations, the delay distribution of packets traversing the switch shrinks with the line rate. Given that the minimum delay in these experiments was $T = 20$ ms, 76%, 84% and 99% of packets, experienced shorter delays than the value of `min_rtt/4` = 5 ms for 1, 2 and 4.5 Gbps scenarios respectively when *RACK* was used under the LB configuration. This quantity is important as it means a higher percentage of packets are expected to arrive within the time window of `rack_timeout` as the line rate increases.

The TCP performance in the experiments was determined as a result of the interaction between the arrival rate, reordering and loss detection. The arrival rate to the load balancer determines the extent of reordering produced by the system, the degree of reordering in turn determines the outcome of TCP loss detection, and the loss detection mechanism determines the arrival rate through TCP congestion control and retransmissions. Changing the loss detection algorithm, therefore, changes the extent of reordering. In Table I, we present statistics of *reordering density* (RD), as defined in the previous section, and the average utilization of the link connecting the main router to `s0`, for the LB configuration at $C = 4.5$ Gbps as an example. The RD measurements allow us to characterize the observed reordering pattern by quantizing the percentage of packets that arrived 3, 30 or 300 positions later than their correct orders within their flows. The high utilization for 3Thresh is likely due to a high number of retransmissions (see Fig. 4(f))

TABLE I
REORDERING DENSITY DISTRIBUTION WITHIN TCP FLOWS AND RESULTING LINK UTILIZATION FOR $C = 4.5$ GBPS AND $F = 150$

Algorithm	$RD > 0$	$RD > 3$	$RD > 30$	$RD > 300$	Link Util.
RACK	32.81%	22.90%	6.32%	0.05%	47.2%
adapThresh	34.49%	23.75%	7.29%	0.10%	47.9%
3Thresh	33.26%	22.46%	5.77%	0.04%	54.7%

V. DISCUSSION AND FUTURE WORK

Do switches still have to deliver packets in sequence? Our results suggest that the resilience of TCP recovery algorithms, especially of time-based *RACK*, to patterns of reordering produced by a load balanced switch increases with the line rate. The results in Section IV-B suggest that under typical conditions for a core network switch today, the performance of TCP using *RACK* loss detection with reordering due to load balancing is similar to the performance of TCP with traditional loss detection without reordering. This also validates our suggestion that the conventional wisdom regarding load balanced switch architectures needs to be revisited in light of recent advances in TCP loss detection. It would be interesting to investigate the behavior at higher, more realistic line rates and in cases where multiple reordering switches are traversed. Alterations to our experiment methodology can enable testing of both. We also intend to approach the problem analytically.

These findings open the door to multiple future research directions: It is important to consider how low delay variants such as TCP BBR would behave under packet reordering. Implementations of QUIC are also expected to use mechanisms similar to adaptive *dupthresh* and *RACK* [24]. Our insight, therefore, could potentially also hold for QUIC. Finally, our findings might have implications for data center networks, wireless communication protocols for multi-RAT (multiple Radio Access Technologies) networks and mobile ad hoc networks, in all of which packet reordering may also be encountered.

REFERENCES

[1] M. Laor and L. Gendel, "The effect of packet reordering in a backbone link on application throughput," *IEEE Network*, vol. 16, no. 5, 2002.

[2] Y. Oie, T. Suda, M. Murata, and H. Miyahara, "Survey of switching techniques in high-speed networks and their performance," *International Journal of Satellite Communications*, vol. 9, no. 5, pp. 285–303, 1991.

[3] X. Li, Z. Zhou, and M. Hamdi, "Space-memory-memory architecture for clos-network packet switches," in *IEEE ICC 2005.*

[4] A. Smiljanic, R. Fan, and G. Ramamurthy, "RRGS-round-robin greedy scheduling for electronic/optical terabit switches," in *GLOBECOM'99.*, vol. 2. IEEE, 1999, pp. 1244–1250.

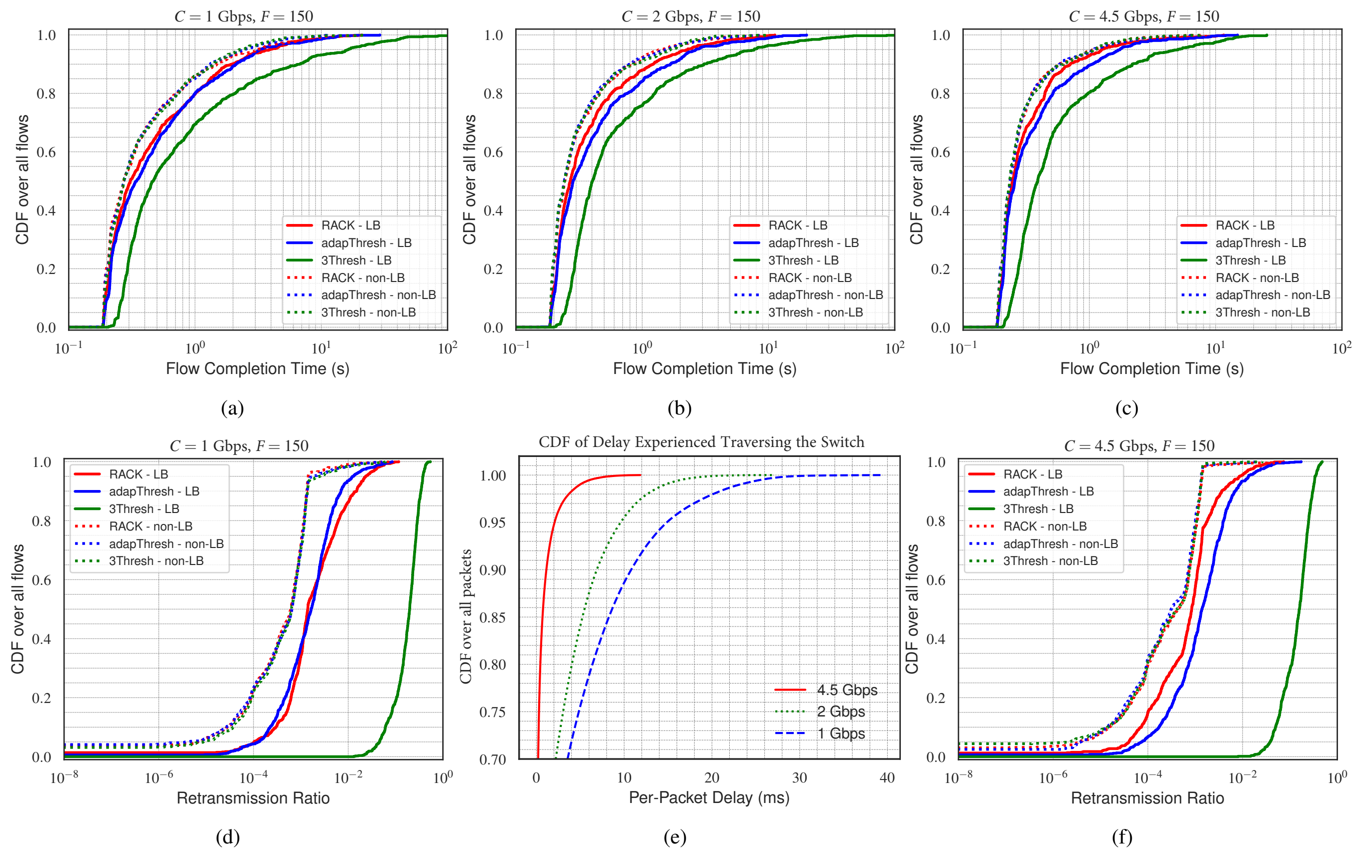


Fig. 4. (a), (b) & (c) CDF of completion times of each flow, (d) & (f) CDF of ratio of retransmissions to all packets of each flow when 150 flow generators were running at each node at capacities 1, 2 and 4.5 Gbps. Only flows larger than 1 MB are included. (e) CDF of delay experienced by each packet through the LB configuration when *RACK* was in use.

[5] C.-S. Chang, D.-S. Lee, and Y.-S. Jou, "Load balanced Birkhoff–von Neumann switches, part i: One-stage buffering," *Computer Communications*, vol. 25, no. 6, pp. 611–622, 2002.

[6] C.-S. Chang, D.-S. Lee, Y.-J. Shih, and C.-L. Yu, "Mailbox switch: a scalable two-stage switch architecture for conflict resolution of ordered packets," *IEEE Transactions on Communications*, vol. 56, no. 1, pp. 136–149, 2008.

[7] I. Keslassy and N. McKeown, "Maintaining packet order in two-stage switches," in *Proceedings. 21st Annual Joint Conf. IEEE Computer and Communications Societies*, vol. 2, 2002, pp. 1032–1041 vol.2.

[8] Cisco, "Cisco CRS Carrier Routing System 8-Slot Line Card Chassis Enhanced Router System Description," June 2020. [Online]. Available: https://www.cisco.com/c/en/us/td/docs/routers/crs/ec/8_slot/system/description/b-crs-8-slot-line-card-chassis-system-description.html

[9] N. M. Piratla and A. P. Jayasumana, "Reordering of packets due to multipath forwarding-an analysis," in *2006 IEEE International Conference on Communications*, vol. 2. IEEE, 2006, pp. 829–834.

[10] K.-c. Leung, V. O. Li, and D. Yang, "An overview of packet reordering in transmission control protocol (TCP): Problems, solutions, and challenges," *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, no. 4, pp. 522–535, 2007.

[11] Y. Cheng and N. Cardwell, "Making Linux TCP fast," in *Netdev*, 2016.

[12] E. Blanton, M. Allman, L. Wang, I. Järvinen, M. Kojo, and Y. Nishida, "A Conservative Loss Recovery Algorithm Based on Selective Acknowledgment (SACK) for TCP," RFC 6675, Aug. 2012.

[13] M. Zhang, B. Karp, S. Floyd, and L. Peterson, "RR-TCP: a reordering-robust TCP with DSACK," in *11th IEEE International Conference on Network Protocols, 2003. Proceedings.*, 2003, pp. 95–106.

[14] A. Reddy, S. Bhandarkar, E. Blanton, and M. Allman, "Improving the Robustness of TCP to Non-Congestion Events," RFC 4653, Aug. 2006.

[15] Y. Cheng, N. Cardwell, N. Dukkipati, and P. Jha, "The RACK-TLP Loss Detection Algorithm for TCP," RFC 8985, Feb. 2021.

[16] F. Weinrank, M. Tuxen, and E. P. Rathgeb, "RACK for SCTP," in *2020 IEEE 28th ICNP*. IEEE Computer Society, Oct 2020, pp. 1–6.

[17] D. Bertsekas and R. Gallager, *Data Networks*, 2nd ed. Prentice Hall, 1996.

[18] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra, "The design and operation of Cloudlab," in *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '19. USA: USENIX Association, 2019, p. 1–14.

[19] M. Johannessen, "Investigate reordering in Linux TCP," Master's thesis, Dept. of Informatics, Univ. of Oslo, Oslo, 2015, accessed on: Jun., 17, 2021. Available: http://home.simula.no/~paalh/students/MadsJohannessen.pdf.

[20] Y. Cheng, N. Cardwell, N. Dukkipati, and P. Jha, "RACK: a time-based fast loss recovery draft-ietf-tcpm-rack-02." Presented at IETF 100 tcpm, Singapore. [Online]. Available: https://datatracker.ietf.org/meeting/100/materials/slides-100-tcpm-draft-ietf-tcpm-rack-01.pdf

[21] S. Yang, B. Lin, P. Tune, and J. J. Xu, "A simple re-sequencing load-balanced switch based on analytical packet reordering bounds," in *IEEE INFOCOM 2017*, 2017, pp. 1–9.

[22] P. Jurkiewicz, G. Rzym, and P. Boryło, "Flow length and size distributions in campus Internet traffic," *Computer Communications*, vol. 167, pp. 15–30, 2021.

[23] N. M. Piratla, A. P. Jayasumana, and A. A. Bare, "Reorder density (rd): A formal, comprehensive metric for packet reordering," in *Proceedings of the 4th IFIP-TC6 International Conference on Networking Technologies, Services, and Protocols; Performance of Computer and Communication Networks; Mobile and Wireless Communication Systems*, ser. NETWORKING'05, Berlin, Heidelberg, 2005, p. 78–89. [Online]. Available: https://doi.org/10.1007/11422778_7

[24] J. Iyengar and I. Swett, "QUIC Loss Detection and Congestion Control," RFC 9002, May 2021.