

Pulser: Fast Congestion Response using Explicit Incast Notifications for Datacenter Networks

Hamidreza Almasi

Hamed Rezaei

Muhammad Usama Chaudhry

Balajee Vamanan

University of Illinois at Chicago

ABSTRACT

Datacenter applications frequently cause incast congestion, which degrades both flow completion times of short flows and throughput of long flows. Without isolating incast, existing congestion control schemes (e.g., DCTCP) rely on existing ECN signal to react to general congestion, and they lose performance due to their slow, cautious, and inaccurate reaction to incast. We propose to isolate incast using Explicit Incast Notifications (EIN) that are generated by switches, similar to ECN. Our incast detection is fast and accurate. Further, we present our congestion control scheme, called *Pulser*, which drastically backs off during incast based on EIN, but restores sending rate once incast ends. Our real experiments and ns-3 simulations show that *Pulser* outperforms prior schemes, DCTCP and ICTCP, in both flow completion times and throughput.

1 INTRODUCTION

Datacenters provide fast, curated access to vast amounts of Internet data. Today’s datacenters host a mix of applications – foreground applications perform distributed lookup in response to user queries and background applications perform data update and reorganization. While foreground applications predominantly generate short flows and the nature of distributed lookup implies that their performance is sensitive to higher percentiles (i.e., tail) of short-flow completion times [9], background applications generate long lasting flows and require high throughput. Therefore, today’s datacenter networks optimize short-flow completion times and long-flow throughput.

The key to optimizing both flow completion times (of short flows) and the throughput (of long flows) fundamentally lies in *accurately* and *quickly* responding to congestion. Traditional TCP uses packet loss to modulate its sending rate and relies on duplicate ACKs and timeouts to infer packet loss. Because packet loss is often a late indication of congestion, today’s datacenter networks leverage some form of Active Queue Management (AQM) such as Explicit Congestion Notification (ECN), to quickly infer congestion. Current state-of-the-art datacenter networks use variants of DCTCP [2], which leverages ECN. ECN-enabled routers mark packets if their instantaneous queue length exceeds a predefined

threshold and DCTCP senders modulates their sending rate proportional to the fraction of observed ECN marks in the ACK packets.

While DCTCP senders respond to congestion faster than traditional TCP using early network feedback (i.e., ECN), DCTCP incurs packet drops when network queues buildup at a much faster *rate* than DCTCP senders can respond; we show this phenomenon later in our results. Indeed, many foreground datacenter applications that, by design, perform distributed lookup for small data items spread across hundreds or thousands of servers, and, therefore, cause frequent *incasts* (i.e., data from many input ports converges to one output port and cause rapid queue buildup). Today’s incast-heavy applications (e.g., Web Search) and high-bandwidth network topologies (e.g., fat-trees with low over-subscription factors) imply that congestion often happens *due to incasts* at the network edge, as reported by Google [22] and Microsoft [15]. Because incast causes a rapid queue buildup in a short time, DCTCP’s iterative, gradual window adaptation might not prevent buffer overflow. A more aggressive window adaptation algorithm or lower ECN threshold at the switch would cause throughput loss [5].

In this paper, we make the case for isolating incast from other general cases of congestion. Because incast congestion is the common case, accurate detection and timely response to incast can significantly improve network performance, as our results show. Because detecting incasts at the end-hosts would require multiple roundtrips and would be significantly less efficient due to the short incast time scales, we argue for detecting incasts at switches, as opposed to detecting at end-hosts. We present a novel algorithm for detecting incasts in a short time interval by monitoring the gradient of queue length over small time windows. Similar to ECN, switches set an Explicit Incast Notification (EIN) mark upon detecting incasts. Switches detect incast per output port and mark packets traversing through those ports.

We propose a DCTCP variant, called *Pulser*, which leverages EIN for window adaptation. *Pulser* resets the congestion window to a small value upon observing EIN marked ACKs. While incasts last only for a short time and contribute to a small fraction of the overall network load, drastically resetting the congestion window only to ramp-up soon after

would cause throughput loss. Therefore, Pulser restores the congestion window to its pre-incast value if subsequently received ACKs do not have EIN marks. The net effect is that Pulser has a braking phase when EIN marks are observed, which only lasts for a short time; after the incast episode, Pulser restores its pre-incast sending rate instead of a gradual increase. Fast and accurate incast detection is key to Pulser’s design, and without such detection, Pulser (or DCTCP) would lose throughput. ICTCP [25] addresses incast at the receiver without adding network support. Consequently, ICTCP’s end-host detection is slow and Pulser outperforms ICTCP (see section 5).

In summary, we make the following contributions:

- We propose a combination of in-network and end-host mechanisms that specifically target incast congestion, which is common but not efficiently handled by existing proposals.
- We introduce a novel, gradient-based incast detection logic in switches, which is fast and accurate.
- We propose a congestion control scheme that uniquely leverages our incast detection to improve both short-flow completion times and long-flow throughput.

Using a combination of real testbed and ns-3 [21] simulations, we show that Pulser improves both 99th-percentile short-flow completion times and long-flow throughput:

With simulations, Pulser:

- achieves 10% (1.12x) reduction in median and 50% (2x) reduction in 99th percentile FCT than DCTCP and ICTCP, on average, for loads greater than 20%. At higher loads, Pulser achieves up to 25% and 70% reduction in median and 99th percentile FCT, respectively.
- achieves 20% higher long-flow throughput than DCTCP and ICTCP, on average, for loads greater than 20%. Pulser achieves up to 50% higher throughput at higher loads.

With real testbed, Pulser:

- outperforms DCTCP by about 26% in 99th percentile flow completion times.
- achieves about 25% higher throughput than DCTCP.

The remainder of the paper is as follows. We start the motivation for our paper in section 2, following by our design in section 3. Sections 5 and 6 present our experimental methodology and results. We discuss related work in section 7 and conclude in section 8.

2 MOTIVATION

DCTCP is a pioneering work that made a key insight that an accurate, proportional response to congestion using ECN could improve both flow completion times and throughput. DCTCP assembles 1-bit ECN marks at the end-host for a sequence of packets to infer accurate queue length at

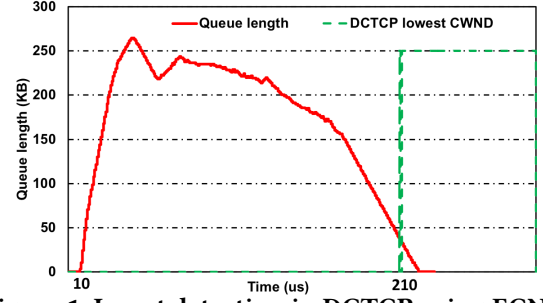


Figure 1: Incast detection in DCTCP using ECN

the bottleneck switch and uses the information to modulate the sending window [3] accordingly. DCTCP performs well for long flows or when incast is somewhat mild. However, DCTCP’s performs poorly with an incast-heavy traffic with many short flows. The queue size would increase rapidly during incast, and therefore, it is essential to *drastically* slow down all senders in order to avoid packet loss. However, DCTCP’s proportional response would require a few round-trips (RTTs), which is sub-optimal for incast.

Our at-scale ns-3 simulations capture this behavior. Figure ?? shows the evolution of a specific switch’s queue length (red line) over time (X-axis). Figure ?? also shows DCTCP’s reaction (green line), which is either 0 (DCTCP does not slow down) or 1 (DCTCP slows down due to ECN). We clearly see that even though incast starts at *time* = 10, DCTCP does not react until *time* = 210 when it is too late (i.e., DCTCP does not reduce its congestion window to the desired level until *time* = 210). Please see section 5 for topology and workload details. While one could think of reducing ECN threshold to improve DCTCP’s reaction to incasts, past papers have shown that smaller ECN thresholds detect incast spuriously and cause throughput loss [2, 5]. Later, we show that Pulser is able to react much faster than DCTCP (section 5).

Though datacenter traffic is heavy tailed with a small fraction of long flows accounting for the majority of bytes transferred, the growing popularity of online services (e.g., Google Search, Facebook) implies that the fraction of short flows and the intensity of incast is bound to increase. While the performance of all end-to-end rate control schemes degrade as the fraction of short flows increase, we contend that an AQM scheme that is customized for incast detection and a congestion control algorithm that leverages the scheme could substantially improve performance over the current state-of-the-art.

3 PULSER

Pulser consists of two parts: (1) fast and accurate incast detection and (2) end-to-end congestion control that leverages incast detection. We describe our novel incast detection in section 3.1 and our congestion control in section 3.2.

3.1 Explicit Incast Notifications (EIN)

Algorithm 1: EIN generation at switches

Result: Set or Reset EIN

Input : $Qlen$

Output: EIN

```

1 for Each packet “P” at dequeue do
2    $Gradient = (Qlen - Qlen_{prev}) / (T - T_{prev})$ 
3    $Qlen_{prev} = Qlen$ 
4    $T_{prev} = T$ 
5   Store Gradient in a sliding window
6   Calculate Average Gradient for “N” samples
7   if Average Gradient >  $EIN_{threshold}$  then
8     Set EIN
9      $EIN_{prev} = 1$ 
10  else
11    if  $EIN_{prev} == 1$  then
12      if  $Qlen > HighWaterMark$  then
13        Set EIN
14      end
15    else
16      Reset EIN
17       $EIN_{prev} = 0$ 
18    end
19  end
20 end

```

During incast, data from multiple input ports (e.g., > 8) gets forwarded to the same output port within a switch, which cause a steep increase in the output port’s queue length. Therefore, our incast detection logic uses the gradient of queue length as opposed to the queue length itself.

Similar to most ECN implementations, we perform incast detection at the dequeue side. At a high level, we calculate the gradient during each dequeue event w.r.t to previous event (i.e., slope between two consecutive packets that were dequeued). We maintain a sliding window of past N samples of gradient, where N is configurable. If the average gradient is more than a configurable threshold, then we mark outgoing packets by setting the new Explicit Incast Notification (EIN) bit. EIN requires one additional bit in the IP header (similar to CE bit for ECN), which is set by the switches and another bit in the TCP header for notify senders (similar to ECE for ECN).

Algorithm 1 shows our complete algorithm. In our implementation, we also set EIN when the current queue length exceeds a configurable *HighWaterMark*, which serves as hysteresis (see lines 11-18). We set *HighWaterMark* to be

higher than ECN threshold to avoid throughput loss. Our incast detection has two main parameters, N and $EIN_{threshold}$. We empirically found that using $N = 50$ and $EIN_{threshold} = 0.25 \times Line\ Rate$ provides optimal performance. Intuitively, our parameter settings mean that if the queue is building at the rate of $0.25 \times Line\ Rate$ for the past 50 samples on average, then we detect an incast episode. Consequently, we detect incast if either the queue builds up steeply in a very short time window or if the queue consistently builds up over a long time window. In either case, it is desirable to react strongly by setting the EIN bit to avoid buffer overflow. We also performed an exhaustive sensitivity study but do not show due to space constraints. Our incast detection is fast and accurate, as we show in section 5.

3.2 Congestion control

We design Pulser’s congestion control by leveraging EIN. If a Pulser sender gets a packet with EIN mark, the sender reduces its congestion window to a configurable, *safe* value after saving the current congestion window. Such a drastic response to incast congestion would likely ease congestion. Once incast finishes, the sender would stop receiving EIN marks. If the sender doesn’t observe any EIN marked packets for the current batch of packets, then the sender restores the window to its previous *saved* value. Equations 1 and 2 show how we modify the congestion window at the beginning and end of an incast episode, which we infer via EIN marks.

$$\begin{aligned} cwnd_{prev} &\leftarrow cwnd \\ cwnd &\leftarrow cwnd_{safe} \end{aligned} \tag{1}$$

$$cwnd \leftarrow cwnd_{prev} \tag{2}$$

We empirically found out that setting $cwnd_{safe} = 4 \times MSS$ provides optimal performance. We did a sensitivity study but do not show due to lack of space. As you can see, our congestion control is only a handful lines of code change over existing DCTCP implementation and is deployment friendly.

4 SIMULATION METHODOLOGY

We use ns-3 [21] to simulate a leaf-spine datacenter topology, which is commonly used in today’s datacenters [1]. In our topology, the fabric interconnects 400 servers using 20 leaf switches with each leaf switch connecting to 20 servers. The leaf switches are connected to 10 spines, resulting in an over-subscription factor of 2. The servers and switches are connected by 10 *Gbps* links with an unloaded link delay of 10 μs ; the unloaded Round-Trip Time (RTT) for the longest path (i.e., 4 hops) is 80 μs .

We model our workloads based on reported results [6], with a mix of short and long flows. Flow arrivals follow a Poisson distribution and the source and destination for each flow is chosen uniformly randomly. Our short flows' sizes are randomly chosen from 8 KB to 32 KB and we set long flow sizes to 1 MB. As is typical, long flows contribute to 30 % of the overall network load, which we vary in our experiments [4]. We also model incast traffic as per [24]. The flows and their destinations are chosen randomly and are varied during the experiment. Our default incast degree is 24 but vary it in our sensitivity analysis 5.4.

We compare four schemes: DCTCP, ICTCP, Pulser, and *Ideal*. Our DCTCP and ICTCP implementations use their recommended parameter settings (e.g. ECN threshold) and our results match their reported numbers. We implemented Pulser on top of DCTCP [2]. We implemented algorithm 1 in switches and our congestion control in end-hosts. We set: $cwnd_{safe} = 4 \times MSS$, $EIN_{threshold} = 0.25 \times Line\ Rate$, and $N = 50$ as default, after sensitivity studies (not shown due to lack of space). We also implemented an *Ideal* congestion control scheme where senders have *oracular* global knowledge and send at optimal sending rate. While the Ideal scheme is not practical, we show its results to set reasonable upper bounds on performance.

5 SIMULATION RESULTS

We summarize our evaluation of Pulser as follows:

- **Flow Completion Time (FCT):** We compare the median and 99th percentile short-flow completion times of Pulser with DCTCP, ICTCP, and Ideal. Pulser achieves 10% (1.12x) reduction in median and 50% (2x) reduction in 99th percentile FCT than DCTCP and ICTCP, on average, for loads greater than 20%. At higher loads, Pulser achieves up to 25% and 70% reduction in median and 99th percentile FCT, respectively.
- **Throughput:** We compare the long-flow throughput of Pulser with DCTCP, ICTCP, and Ideal. Pulser achieves 20% higher long-flow throughput than DCTCP and ICTCP, on average, for loads greater than 20%. Pulser achieves up to 50% higher throughput at higher loads.
- **Queue length analysis:** We analyzed how the queues buildup in Pulser and DCTCP. Pulser reduces queue lengths drastically (by up to 2x) compared to DCTCP.
- **Sensitivity to incast:** Pulser's improvements increase with increasing incast degree and is robust across a range of typical incast degrees.

We provide a more exhaustive analysis below.

5.1 Flow Completion Time

Figure 2 and figure 3 compare the median and tail (99th percentile) flow completion times of DCTCP, ICTCP, Pulser, and

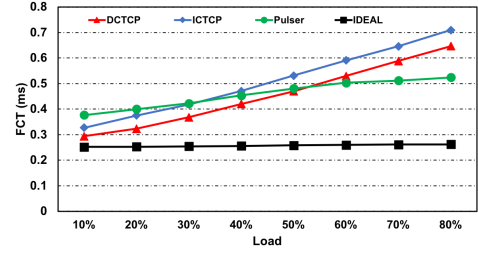


Figure 2: Median flow completion time

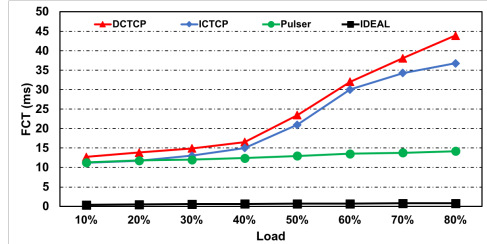


Figure 3: 99th %-ile flow completion times

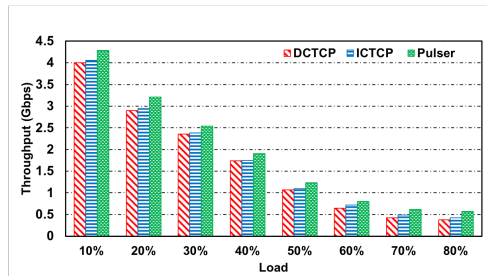


Figure 4: Throughput comparisons

Ideal. We show flow completion times along Y-axis versus network load on X-axis. As load increases, all schemes incur more queuing and their FCTs degrade. While Pulser achieves reduction in both median and tail FCT, Pulser's achieves better reduction in tail flow completion times than in median flow completion times. Because datacenter applications are more sensitive to tail FCT than median, Pulser's makes the right trade-off.

Compared to DCTCP, Pulser reduces tail flow completion time of about 51% for loads greater than 40% (typical operating point of most datacenters). Compared to ICTCP, Pulser reduces flow completion time by about 46% at higher loads. Because incast congestion is not an issue at lower loads, Pulser does not significantly outperform at lower loads. *Ideal* method outperforms all other schemes, which shows that there is significant improvement to be achieved.

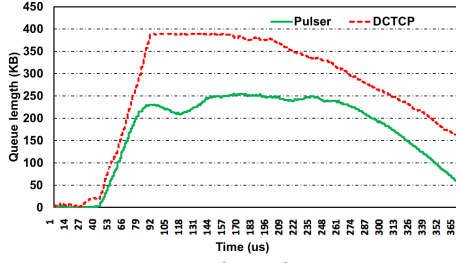


Figure 5: Queue length over time

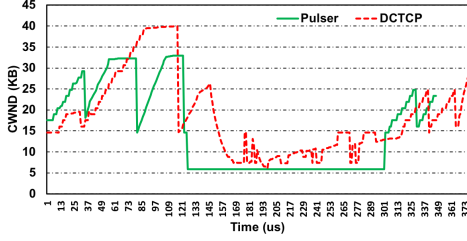


Figure 6: Congestion window at a long flow sender

5.2 Throughput

In this section we compare Pulser’s throughput with ICTCP, DCTCP and ideal scheme. Long flow throughput suffers for all schemes at higher loads due to increased (incast) congestion at higher loads. As we can see from figure 4, Pulser achieves higher throughput across all loads: First, Pulser reduces the number of packet drops of those background flows that share links that experience high incast congestion, as compared to other schemes. Second, when incast is finished, Pulser uses the last congestion window before incast as the new congestion window, without resorting to gradual window increase (e.g., slow start). Pulser’s ON/OFF window modulation helps senders to restore their pre-incast sending rate pretty quickly. Pulser achieves 16% and 22% higher throughput compared to ICTCP and DCTCP respectively (in 40% load and beyond).

5.3 Queue length

In this section, we analyze the queuing behavior of Pulser and relate it to Pulser’s congestion control (i.e., evolution of congestion window over time). For this experiment, we run our workload with 60% load. Figure 5 shows the queue length at an aggregator switch’s output port (Y-axis) over time (X-axis). We analyze DCTCP (red) vs. Pulser (green). We see that, Pulser reduces the queue buildup by as much as 50% (2x).

To connect Pulser’s queuing behavior to our congestion control, we compare the congestion window evolution versus time (at the sender) for DCTCP and Pulser in figure 6.

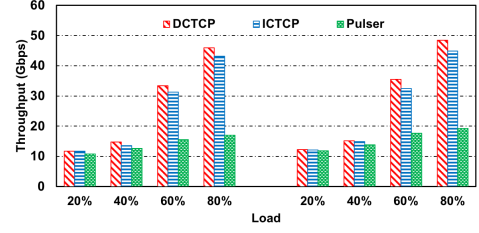


Figure 7: Sensitivity of 99th percentile flow completion times to incast degree

At $time = 120\mu s$, incast starts. While DCTCP gradually reduces the congestion window and oscillates around due to the absence of a precise signal that indicates incast, Pulser leverages a more precise EIN to backup almost instantly. When the incast finishes at $time = 300\mu s$, Pulser instantly recovers. By instantly backing off, the Pulser’s long-flow sender minimizes queuing delay, which helps short flows. By restoring its previous sending rate after incast, Pulser sender achieves better throughput.

5.4 Sensitivity to incast degree

We analyze the sensitivity of our results to different incast degrees. For this study, we compare Pulser’s tail flow completion time to those of DCTCP and ICTCP for varying incast degrees. We vary incast degree as 24 (default), 32, and 40. Figure 7 shows the 99th percentile flow completion times for varying incast degrees, normalized to our default case (i.e., incast degree of 24).

As expected for both incast degrees, all methods experience increasing tail flow completion time with load increments. In both cases, Pulser outperforms DCTCP and ICTCP with a substantial margin of at least 2X for 60% and 80% loads. Lower loads do not suffer from high incast congestion, and, therefore, there is limited opportunity for improvement. Similar to higher loads, higher incast degree provide more opportunity for Pulser. Nevertheless, Pulser’s relative performance improvement remains robust across varying loads and incast degrees.

6 REAL IMPLEMENTATION

Our real testbed consists of three Dell 7040 *Optiplex* servers with 32 GB of memory, Intel Quad core processors (3.4 GHz i7) and 1 Gbps NICs. Two servers act as clients and generate traffic to the third server, which acts as an aggregator (leaf server). Because EIN requires switch support, we use another server with two network interfaces as a software switch (kernel version 4.4.0). The two client servers are connected by a physical Netgear *Prosafe* switch to our software switch, which connects to the aggregator. Further, to generate a realistic incast scenario with only two servers, we place 8

Metric	DCTCP	Pulser
<i>Avg. flow completion time (s)</i>	1.99	1.59
<i>99th percentile flow completion time (s)</i>	13.32	9.85
<i>Throughput (Mbps)</i>	28	35

Table 1: Real implementation results

VMs in each of the two client servers; the VMs run Ubuntu 12.04 LTS (kernel version 3.2.18) with 2GB of memory. We rate limit the client VM’s NICs to 50 Mbps. The two client machines each generate $50 \times 8 = 400\text{Mbps}$ of traffic to the physical switch, which connects to the software switch over a 1 Gbps link (i.e., there is no bottleneck). However, the link between the software switch and the aggregator is rate limited to 50 Mbps, creating a realistic incast (i.e., there is 800 Mbps of incoming traffic into the software switch but the outgoing port is only 50 Mbps, which creates a realistic incast degree of 16). We use *iperf3* to generate traffic. We generate a background 40 MB long flow from one of the client VMs. The other 15 client VMs generate synchronous bursts of short 100KB flows, with random jitter. We run the experiment for 80 minutes and measure the flow completion times of short flows and throughput of long flows.

Table 1 shows the flow completion times – both average (not median) and 99th percentile – and throughput comparison between DCTCP and Pulser in our real testbed. Our real testbed is smaller in scale, and, therefore, the intensity of incast and the corresponding tail effects are somewhat less pronounced in our real testbed than in our at-scale simulations. Nevertheless, Pulser outperforms DCTCP by about 20% and 26% in average and 99th percentile flow completion times, respectively. Similarly, Pulser achieves about 25% higher throughput than DCTCP. While we do not have access to a datacenter-scale testbed, our substantial performance gains in the small testbed shows the potential of Pulser in a more realistic setting.

7 RELATED WORK

While Internet Congestion control is a well-studied research area, datacenter congestion control continues to garner interest in the networking community and there are a number of recent papers on datacenter congestion control. We have discussed DCTCP and ICTCP in earlier sections. We will summarize other related work in this area.

Rate Control Protocol (RCP) [10] is an alternative to window-based TCP protocols in which switches directly inform the senders of their fair share sending rate by observing the rates of all intervening flows. But, RCP does not isolate incast, and requires switch support, which is not available today. Similar to Pulser, TIMELY [17] uses a gradient-based approach. However, unlike Pulser, TIMELY is RTT-based, its detection logic is not customized for incast, and it is implemented at end-hosts. Therefore, TIMELY’s detection is unlikely to

be as fast and as accurate as our approach. DCQCN [27] leverages ECN for RDMA and performs rate-based congestion control. Our incast detection and congestion control ideas are complimentary to DCQCN and they would likely improve DCQCN’s incast performance. QCN [19] provides congestion control based on network feedback (similar to DCTCP/ECN) but operated at the Ethernet layer and doesn’t isolate incast. NumFabric [18] provides other more flexible bandwidth allocations other than TCP’s fair share. Express-Pass [8] and NDP [12] provide receiver-driven congestion control; Pulser, in contrast, is switch-driven and isolates incast congestion from other forms of congestion (e.g., congestion in network caused by flow collisions). A number of proposals [4, 7, 11, 14, 16, 23, 26] focus on flow scheduling and prioritize critical flows (e.g., short flows) whereas our main focus is on incast congestion control. Similarly, other load balancing proposals [1, 13, 20] are complimentary to Pulser.

8 CONCLUSION

Incast congestion is a dominant form of congestion in datacenter networks. Prior approaches do not isolate and detect incast in the network, and, therefore, existing end-host congestion control could not aggressively respond to incast without losing throughput. We proposed Explicit Incast Notification (EIN), a gradient-based incast detection at network switches, which is both fast and accurate. Leveraging EIN, we introduced our congestion control scheme, called Pulser, which quickly backs off during incast for short time intervals without hurting latency and ramps up soon after without losing throughput. Using simulations and a real implementation, we showed that Pulser outperforms DCTCP and ICTCP. As data and Internet traffic continue to grow exponentially, incast is likely to become even more dominant in datacenters, requiring an incast-specific AQM such as EIN and a congestion-control schemes such as Pulser.

REFERENCES

- [1] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. 2014. CONGA: Distributed Congestion-aware Load Balancing for Datacenters. In *Proceedings of the 2014 ACM Conference on SIGCOMM (SIGCOMM ’14)*. 503–514.
- [2] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 conference (SIGCOMM ’10)*. ACM, New York, NY, USA, 63–74. <https://doi.org/10.1145/1851182.1851192>
- [3] Mohammad Alizadeh, Adel Javanmard, and Balaji Prabhakar. 2011. Analysis of DCTCP: Stability, Convergence, and Fairness. In *Proceedings of SIGMETRICS*. 73–84.

- [4] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. 2013. pFabric: Minimal Near-optimal Datacenter Transport. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM (SIGCOMM '13)*. ACM, New York, NY, USA, 435–446. <https://doi.org/10.1145/2486001.2486031>
- [5] Wei Bai, Li Chen, Kai Chen, and Haitao Wu. 2016. Enabling ECN in Multi-service Multi-queue Data Centers. In *Proceedings of NSDI*. 537–549.
- [6] Theophilus Benson, Aditya Akella, and David A. Maltz. 2010. Network Traffic Characteristics of Data Centers in the Wild. In *Proceedings of IMC*. 267–280.
- [7] Li Chen, Kai Chen, Wei Bai, and Mohammad Alizadeh. 2016. Scheduling Mix-flows in Commodity Datacenters with Karuna. In *Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference (SIGCOMM '16)*. 174–187.
- [8] Inho Cho, Keon Jang, and Dongsu Han. 2017. Credit-Scheduled Delay-Bounded Congestion Control for Datacenters. In *Proceedings of SIGCOMM*. 239–252.
- [9] Jeffrey Dean and Luiz André Barroso. 2013. The Tail at Scale. *Commun. ACM* 56, 2 (Feb. 2013), 74–80. <https://doi.org/10.1145/2408776.2408794>
- [10] Nandita Dukkipati, Masayoshi Kobayashi, Rui Zhang-Shen, and Nick McKeown. 2005. Processor Sharing Flows in the Internet. In *Proceedings of the 13th International Conference on Quality of Service (IWQoS'05)*. Springer-Verlag, 271–285.
- [11] Peter X. Gao et al. 2015. pHost: Distributed Near-optimal Datacenter Transport over Commodity Network Fabric. In *Proceedings of CoNEXT*. 1:1–1:12.
- [12] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. 2017. Architecting Datacenter Networks and Stacks for Low Latency and High Performance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. ACM, 29–42.
- [13] Keqiang He, Eric Rozner, Kanak Agarwal, Wes Felter, John Carter, and Aditya Akella. 2015. Presto: Edge-based Load Balancing for Fast Datacenter Networks. In *Proceedings of SIGCOMM*. 465–478.
- [14] Chi-Yao Hong, Matthew Caesar, and P. Brighten Godfrey. 2012. Finishing flows quickly with preemptive scheduling. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication (SIGCOMM '12)*. ACM, 127–138. <https://doi.org/10.1145/2342356.2342389>
- [15] Srikanth Kandula, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken. 2009. The Nature of Data Center Traffic: Measurements & Analysis. In *Proceedings of IMC*. 202–208.
- [16] Radhika Mittal, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Universal Packet Scheduling. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation (NSDI'16)*. 501–521.
- [17] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. 2015. TIMELY: RTT-based Congestion Control for the Datacenter. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '15)*. ACM, 537–550.
- [18] Kanthi Nagaraj, Dinesh Bharadia, Hongzi Mao, Sandeep Chinchali, Mohammad Alizadeh, and Sachin Katti. 2016. NUMFabric: Fast and Flexible Bandwidth Allocation in Datacenters. In *Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference (SIGCOMM '16)*. 188–201.
- [19] R Pan, B Prabhakar, and A Laxmikantha. 2007. QCN: Quantized congestion notification an overview.
- [20] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. 2011. Improving datacenter performance and robustness with multipath TCP. In *Proceedings of the ACM SIGCOMM 2011 conference (SIGCOMM '11)*. ACM, New York, NY, USA, 266–277. <https://doi.org/10.1145/2018436.2018467>
- [21] George F Riley and Thomas R Henderson. 2010. The ns-3 network simulator. In *Modeling and tools for network simulation*. Springer, 15–34.
- [22] Arjun Singh et al. 2015. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *Proceedings of SIGCOMM*. 183–197.
- [23] Balajee Vamanan, Jahangir Hasan, and T.N. Vijaykumar. 2012. Deadline-aware Datacenter TCP (D2TCP). In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '12)*.
- [24] Vijay Vasudevan, Amar Phanishayee, Hiral Shah, Elie Krevat, David G Andersen, Gregory R Ganger, Garth A Gibson, and Brian Mueller. 2009. Safe and effective fine-grained TCP retransmissions for datacenter communication. In *ACM SIGCOMM computer communication review*, Vol. 39. ACM, 303–314.
- [25] Haitao Wu, Zhenqian Feng, Chuanxiong Guo, and Yongguang Zhang. 2010. ICTCP: Incast Congestion Control for TCP in Data Center Networks. In *Proceedings of CoNEXT*. 13:1–13:12.
- [26] David Zats, Tathagata Das, Prashanth Mohan, Dhruba Borthakur, and Randy Katz. 2012. DeTail: Reducing the Flow Completion Time Tail in Datacenter Networks. In *Proceedings of SIGCOMM*. 139–150.
- [27] Yibo Zhu et al. 2015. Congestion Control for Large-Scale RDMA Deployments. In *Proceedings of SIGCOMM*. 523–536.