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

Hamidreza Almasi, Hamed Rezaei, Muhammad Usama Chaudhry, and Balajee Vamanan
Department of Computer Science, University of Illinois at Chicago, USA
Email: {halmas3, hrezae2, mchaud30, bvamanan}@uic.edu

***Abstract*—Datacenter applications frequently cause incast congestion, which degrades short flows' flow completion times and long flows' throughput. Existing congestion control schemes (e.g., DCTCP) do not explicitly detect and isolate incast. Instead, they rely on existing Explicit Congestion Notification (ECN) to react to general congestion. They, therefore, lose performance due to slow, cautious, and inaccurate reaction to incast. We propose a novel algorithm that detects incasts and notifies senders using a new *Explicit Incast Notification* (EIN). We show that our incast detection is fast and accurate. Next, we present our congestion control scheme, called *Pulser*, which isolates incasts using EIN. Unlike DCTCP, which gradually adjusts sending rate, *Pulser drastically* backs off during incast and *rapidly* restores sending rate once incast ends (i.e., like a pulse). Our real experiments and ns-3 simulations show that *Pulser* outperforms prior schemes, DCTCP and ICTCP, in both flow completion times and throughput.**

Keywords— Datacenter; TCP; Incast; Tail latency; ECN

I. 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 [1], background applications generate long lasting flows and require high throughput. Therefore, today's datacenter networks optimize short flows' completion times and long flows' throughput.

The key to optimizing both Flow Completion Times (FCT) 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 modulate their sending rate proportional to the fraction of observed ECN marks in the ACK packets.

978-1-7281-1434-7/19/$31.00

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. Many foreground datacenter applications, by design, perform distributed lookup of small data items that are spread among hundreds or thousands of servers. Therefore, they cause frequent *incasts* (i.e., data from many input ports converges to one output port and causes 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 [3] and Microsoft [4]. Because incast causes a rapid queue buildup in a short time, DCTCP's iterative, gradual window adaptation does not prevent buffer overflow. Alternatively, tweaking DCTCP's window adaptation algorithm to drastically cut the window or lowering ECN threshold in the switches would worsen throughput due to overreaction to not-so-severe (non-incast) congestion [5].

We make the key insight that because existing schemes do not *explicitly detect* and *isolate* incast from other general forms of congestion, it is not possible for existing schemes to aggressively cut the sending rate without losing throughput. In other words, incast detection and aggressive response go *hand-in-hand*. In this paper, we make the case for isolating incast from other general cases of congestion and we design a protocol that aggressively cuts sending rate in response to incasts. 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 round-trips and would be significantly less efficient due to the short incast time scales (e.g., $50\mu s$ [6]), we argue for detecting incasts at switches, as opposed to detecting at end-hosts. We present a *novel algorithm* for detecting incasts within a short time interval by monitoring the gradient of queue length. 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 via ACKs. While

incasts last for a *short* time and constitute a small fraction of the overall network load, they occur frequently [6]. So, if we drastically reduce the congestion window and slowly ramp-up the window after *each* incast, we would lose substantial throughput. Therefore, *Pulser* restores the congestion window to its *pre-incast* value if subsequent 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 the window to its pre-incast value, instead of a gradual increase. Fast and accurate incast detection is key to *Pulser*'s design, and without such detection, *Pulser* would either lose throughput or incur long latency tails. ICTCP [7] addresses incast at the receiver without adding network support. Consequently, ICTCP's end-host detection is slow and *Pulser* outperforms ICTCP (see section V).

In summary, we make the following contributions:

- We propose a combination of in-network and end-host mechanisms that explicitly detect and isolate incast congestion, which is common but is not efficiently handled by existing proposals.
- We introduce a novel, gradient-based incast detection algorithm, which is fast and accurate.
- We propose *Pulser*, a congestion control scheme that uniquely leverages our incast detection to improve both short flows' completion times and long flows' throughput.

Using a combination of real testbed and ns-3 [8] simulations, we show that *Pulser* improves both 99^{th}-percentile short-flow completion times and long-flow throughput:

With simulations, *Pulser*:

- achieves 10% (1.12x) reduction in median and 50% (2x) reduction in 99^{th} 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 99^{th} percentile FCT, respectively.
- achieves 20% higher long-flow throughput than DCTCP and ICTCP, on average, for loads greater than 20%.

With real testbed, *Pulser*:

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

The remainder of the paper is organized as follows. We start with background and motivation in Section II. We then present our design in Section III. We present our methodology and evaluation in Sections IV, V and VI. We discuss related work in Section VII and conclude in Section VIII.

II. Background and Motivation

DCTCP is a pioneering work that made a key insight that a *proportional* response to congestion, inferred via ECN, is key to improving both flow completion times and throughput. DCTCP aggregates ECN marks at the end-host to accurately estimate the extent of queuing at the bottleneck switch and uses the estimate to modulate the sending window [9]. DCTCP performs well for long flows or when incast is somewhat mild (e.g., small fan-in). However, DCTCP performs poorly

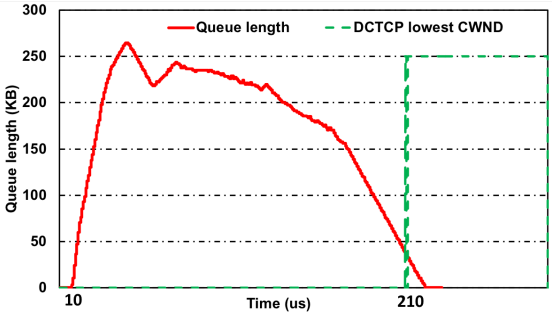

Figure 1: Incast detection in DCTCP using ECN

with an incast-heavy traffic with many short flows (e.g., large fan-in). 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 multiple round-trip times (RTTs), which is too slow for incast.

Our at-scale ns-3 simulations capture this behavior. Figure 1 shows how the queue length of a port (red line), which experiences incast, changes over time (Section V discusses our methodology). Figure 1 also shows DCTCP's reaction (green line) as a binary pulse — 0 indicates insufficient slow down and 1 indicates appropriate slow down. We clearly see that even though incast starts around $10\ us$, DCTCP does not slow down enough until $210\ us$, which it is too late. While it is tempting to reduce the ECN threshold so that DCTCP responds earlier, several past papers have shown that small ECN thresholds cause throughput loss [2, 5]. Later, we will show that *Pulser* reacts to incast much faster than DCTCP (Section V).

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 mechanism that is customized for incasts and an associated congestion control algorithm that leverages the mechanism* could substantially improve performance over the current state-of-the-art algorithms.

III. *Pulser*

Pulser consists of two parts: (1) fast and accurate incast detection and (2) end-to-end congestion control that leverages incast detection. Recall that while current proposals do not isolate incast congestion and only gradually react to general congestion, we *explicitly* detect incasts using EIN and *Pulser drastically* cuts the sending rate upon seeing EIN marks. We describe our novel incast detection (i.e., EIN marking) in section III-A and our congestion control in section III-B.

A. Explicit Incast Notifications (EIN)

During incast, data from multiple input ports (e.g., 8 or higher) gets forwarded to the same output port within a switch, which causes a steep increase in the output port's queue length

Algorithm 1: Incast detection (EIN generation) at switches

Result: Set or Reset 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      EIN_prev ← EIN
6      EIN ← 0
7      Calculate Average Gradient for last "N" samples
8      if Average Gradient > EIN_threshold then
9          EIN ← 1
10     else
11         if EIN_prev == 1 then
12             if Qlen > HighWaterMark then
13                 EIN ← 1
14             end
15         end
16     end
17 end
```

in a short time. Therefore, our incast detection logic uses the *gradient* of queue length rather than the queue length. At a high level, our algorithm marks packets when the gradient of queue length exceeds a threshold (i.e., when the queue fills rapidly).

Algorithm 1 shows our complete incast detection algorithm. Similar to current ECN implementations, we mark packets (i.e., detect incasts) when packets are dequeued. For each packet dequeue event, we calculate the gradient of queue length (line 2) and calculate the average gradient for the last N samples (line 7). If the average gradient is more than $EIN_{threshold}$, we mark packets by setting the new *Explicit Incast Notification* (EIN) bit (line 9). EIN requires an additional bit in the IP header (similar to CE bit for ECN) and it is set by switches; we need another bit in the TCP header of ACKs to notify senders (similar to ECE for ECN). As a safety measure, we also set EIN if we have previously set EIN and if the current queue length exceeds a configurable $HighWaterMark$ (lines 10–14). We set $HighWaterMark$ to be equal to ECN threshold as a lower value would incur throughput loss.

Our incast detection has two main parameters, N and $EIN_{threshold}$. These parameters are *not* independent — smaller N values demand relatively larger $EIN_{threshold}$ and vice versa (i.e., if the queue builds up either rapidly over a short duration or gradually over a long duration, senders must slow down). Further, these parameters depend on the nature of incast (i.e., incast duration). If incasts last for a short duration, we need a small N to quickly react to incasts. Conversely, if incasts last for a long duration, a larger N would provide a more accurate detection (a small N would lead to many false positives and cause loss of throughput). A good rule of thumb is to set these parameters to be close to the duration of typical incasts. A recent study from Facebook [6] reported that 80% of incasts are shorter than $60\mu s$. Accordingly, we pick $N = 50$ (dequeuing 50 MTU size packets on a 10 $Gbps$ link takes $60\mu s$) and $EIN_{threshold} = 0.25 \times LineRate$ as default values. We performed a sensitivity study for these paramters. While we do not show these plots due to space constraints, we found our performance to be robust for a range of incast degrees, load values, and traffic patterns. Our studies also showed stable performance for N values from 10 to 100.

B. Congestion control

We design *Pulser*'s congestion control by leveraging EIN bit. 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 does not 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.

$$cwnd_{prev} \leftarrow cwnd \qquad cwnd \leftarrow cwnd_{safe} \tag{1}$$

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

We empirically found that setting $cwnd_{safe} = 4 \times MSS$ provides optimal performance (MSS stands for maximum segment size) across various loads. Thus, *Pulser* requires only handful lines of code change over DCTCP and is deployment friendly.

IV. Experimental Methodology

We use ns-3 [8] to simulate a leaf-spine datacenter topology, which is commonly used in today's datacenters [10]. 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 in [11], 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. Our long flows contribute to 80 $\%$ of the overall network load, which we vary in our experiments [12]. We also model incast traffic as per [13]. The flows and their destinations are chosen randomly and are varied during the experiment. We generate at least 100,000 flows in each experiment. Our default incast degree is 24 but we vary it in our sensitivity analysis in section V-D.

We compare four schemes: DCTCP, ICTCP, *Pulser*, and *Ideal*. Our DCTCP and ICTCP implementations use their

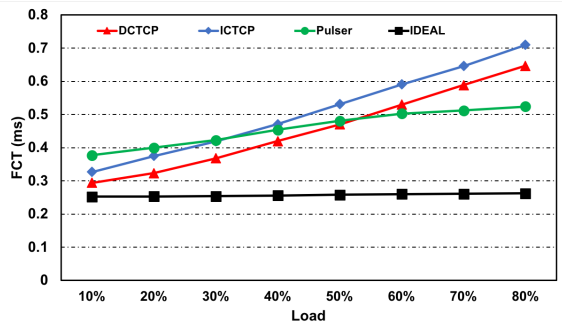

Figure 2: Median flow completion time

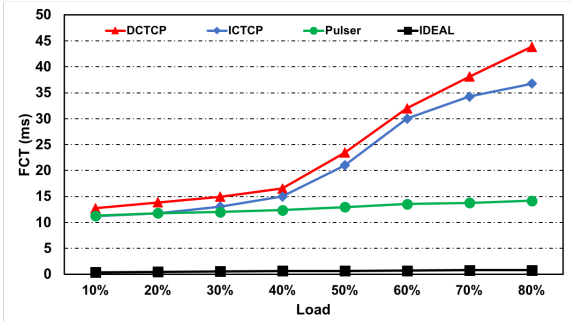

Figure 3: 99^{th} %-ile flow completion times

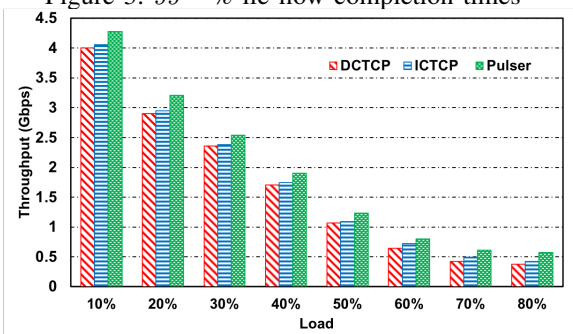

Figure 4: Throughput comparisons

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 $cwnd_{safe} = 4{\times}MSS$, $EIN_{threshold} = 0.25{\times}LineRate$, and $N = 50$ as default values, as discussed in Section III-A. We also implemented an *Ideal* congestion control scheme where senders have *oracular* global knowledge and send at optimal (safe) sending rate. While the Ideal scheme is not practical, we show its results to set a loose upper bounds on performance.

V. Results

We summarize our evaluation of *Pulser* as follows:

- **Flow Completion Time (FCT)**: We compare the median and 99^{th} percentile short-flow completion times of *Pulser* with DCTCP, ICTCP, and Ideal. *Pulser* achieves 10% reduction in median and 50% reduction in 99^{th} 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 99^{th} 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 20% 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.

A. Flow Completion Time

Figure 2 and Figure 3 compare the median and tail (99^{th} percentile) flow completion times of DCTCP, ICTCP, *Pulser*, and 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. *Ideal* has perfect knowledge of future flow arrivals, and therefore, holds back packets without injecting them into the network if they would end up being queued. In other words, *Ideal* does not incur queuing in the network, at all loads. Because Figure 2 and Figure 3 capture only network queuing delays (not source queuing delays), *Ideal's* FCT is the same as the minimum FCT, for all loads. While *Pulser* achieves reduction in both median and tail FCT, *Pulser* achieves a 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* makes the right trade-off.

Because incast congestion is not an issue at lower loads, *Pulser* does not significantly outperform other schemes at lower loads. Compared to DCTCP, *Pulser* reduces tail flow completion time by about 51% at loads greater than 40% (typical operating point of most datacenters). Compared to ICTCP, *Pulser* reduces flow completion time by about 46% at higher loads.

B. Throughput

In this section we compare *Pulser*'s throughput with ICTCP and DCTCP. Long flow throughput decreases with load due to increased congestion (including incast). As we see from figure 4, *Pulser* achieves higher throughput across all loads. First, *Pulser* reduces the number of packet drops of those background flows whose paths suffer high incast congestion. Second, when incast finishes, *Pulser* simply restores the pre-incast congestion window, instead of a slow window increase (e.g., slow start or congestion avoidance). In other words, *Pulser*'s ON/OFF window modulation helps both at the beginning and at the end of incast congestion. Overall, *Pulser* achieves 16% and 22% higher throughput compared to ICTCP and DCTCP, respectively.

C. Queue length

In this section, we analyze the queuing behavior of *Pulser* and relate it to *Pulser*'s congestion control (i.e., evolution of

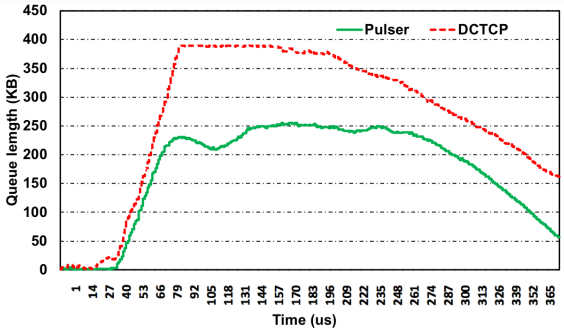

Figure 5: Queue length over time

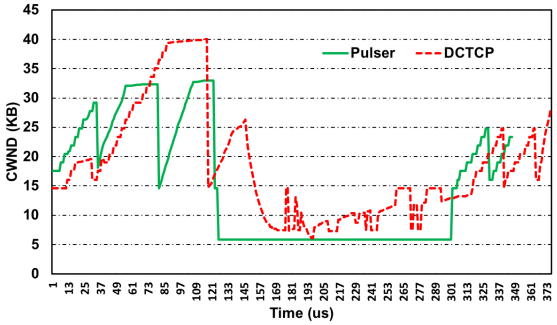

Figure 6: Congestion window at a long flow sender

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. 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.

D. 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 99^{th} percentile flow completion times for varying incast degrees, normalized to our default case (i.e., incast degree of 24).

All schemes experience increasing tail flow completion times with load increments, irrespective of incast degree. *Pulser* outperforms DCTCP and ICTCP with a substantial margin of at least 2X for 60% and 80% loads, for both the two incast degrees. At lower loads, there is not a significant amount of incast congestion, and, therefore, there is limited

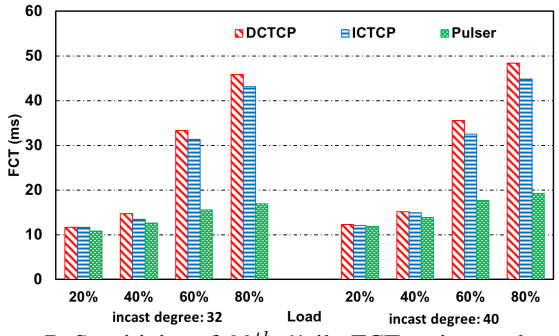

Figure 7: Sensitivity of 99^{th} %-ile FCT to incast degree

opportunity for improvement. High incast degrees, similar to high loads, provide more opportunity for *Pulser*. Nevertheless, *Pulser*'s relative performance improvement remains robust for a range of typical loads and incast degrees.

VI. 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 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 $50 \times 8 = 400\ Mbps$ 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 I shows the flow completion times – both average and 99^{th} percentile flow completion times – and throughput for DCTCP and *Pulser* in our real testbed. Because our real testbed is smaller in scale, the intensity of incast and the corresponding tail effects are somewhat less pronounced than in our at-scale simulations. Nevertheless, *Pulser* outperforms DCTCP by about 20% and 26% in average and 99^{th} 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 show the potential of *Pulser* in a more realistic setting.

Metric	DCTCP	Pulser
Avg. flow completion time (s)	1.99	1.59
99^{th} *percentile flow completion time* (s)	13.32	9.85
Throughput (Mbps)	28	35

Table I: Real implementation results

VII. Related Work

While Internet Congestion control is a well-studied research area, datacenter congestion control continues to garner interest in the networking community. 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) [14] 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. Further, RCP requires substantial modifications to commodity switches, which impedes deployment. Similar to *Pulser*, TIMELY [15] 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 likely not as fast, and not as accurate, as *Pulser*. DCQCN [16] leverages ECN for RDMA and performs rate-based congestion control. Our incast detection and congestion control ideas are complementary to DCQCN and they would likely improve DCQCN's incast performance. QCN [17] provides congestion control based on network feedback (similar to DCTCP/ECN) but operates at the Ethernet layer and doesn't isolate incast. Because QCN operates at layer-2 and most large datacenters rely on IP routing, QCN is not applicable to large datacenters. Extending QCN to layer 3 is not straightforward. S-ECN [18] probabilistically marks packets based on instantaneous gradient of queue length at switches and aggregates several marks at the end-hosts to gradually reduce the congestion window. In contrast, *Pulser* relies on a more accurate incast detection using average gradient at switches, and therefore, *Pulser* can achieve a near instantaneous throttling and restoring of sending rates. NumFabric [19] provides other more flexible bandwidth allocations other than TCP's fair share. ExpressPass [20] and NDP [21] 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 [12, 22–27] focus on flow scheduling and prioritizing critical flows (e.g., short flows) whereas our main focus is on incast congestion control. Finally, load balancing proposals [10, 28] are complementary to *Pulser*.

VIII. Conclusion

Incast congestion is a dominant form of congestion in datacenter networks. Prior approaches do not explicitly detect and isolate incast in the network, and, therefore, existing end-host congestion control mechanisms cannot 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 restores the rate after incast without losing throughput. Using simulations and a real implementation, we showed that *Pulser* outperforms DCTCP and ICTCP. As data and traffic intensity continue to grow exponentially, incast is likely to become even more dominant in datacenters, requiring an incast-specific AQM such as EIN and an associated congestion-control scheme such as *Pulser*.

References

[1] J. Dean and L. A. Barroso, "The tail at scale," *Commun. ACM*, vol. 56, no. 2, Feb. 2013.
[2] M. Alizadeh *et al.*, "Data center tcp (dctcp)," in *SIGCOMM*, 2010.
[3] A. Singh *et al.*, "Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network," in *SIGCOMM*, 2015.
[4] S. Kandula *et al.*, "The nature of data center traffic: Measurements & analysis," in *IMC*, 2009.
[5] W. Bai, L. Chen, K. Chen, and H. Wu, "Enabling ecn in multi-service multi-queue data centers," in *NSDI*, 2016.
[6] Q. Zhang, V. Liu, H. Zeng, and A. Krishnamurthy, "High-resolution measurement of data center microbursts," in *IMC*, 2017.
[7] H. Wu, Z. Feng, C. Guo, and Y. Zhang, "Ictcp: Incast congestion control for tcp in data center networks," in *CoNEXT*, 2010.
[8] G. F. Riley and T. R. Henderson, "The ns-3 network simulator," in *Modeling and tools for network simulation*, 2010.
[9] M. Alizadeh, A. Javanmard, and B. Prabhakar, "Analysis of dctcp: Stability, convergence, and fairness," in *SIGMETRICS*, 2011.
[10] M. Alizadeh *et al.*, "Conga: Distributed congestion-aware load balancing for datacenters," in *SIGCOMM*, 2014.
[11] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *IMC*, 2010.
[12] M. Alizadeh *et al.*, "pfabric: Minimal near-optimal datacenter transport," in *SIGCOMM*, 2013.
[13] V. Vasudevan *et al.*, "Safe and effective fine-grained tcp retransmissions for datacenter communication," in *SIGCOMM*, 2009.
[14] N. Dukkipati, M. Kobayashi, R. Zhang-Shen, and N. McKeown, "Processor sharing flows in the internet," in *IWQoS*, 2005.
[15] R. Mittal *et al.*, "Timely: Rtt-based congestion control for the datacenter," in *SIGCOMM*, 2015.
[16] Y. Zhu *et al.*, "Congestion control for large-scale rdma deployments," in *SIGCOMM*, 2015.
[17] R. Pan, B. Prabhakar, and A. Laxmikantha, "Qcn: Quantized congestion notification an overview," *IEEE802*, 2007.
[18] D. Shan *et al.*, "Micro-burst in data centers: Observations, analysis, and mitigations," in *ICNP*, 2018.
[19] K. Nagaraj *et al.*, "Numfabric: Fast and flexible bandwidth allocation in datacenters," in *SIGCOMM*, 2016.
[20] I. Cho, K. Jang, and D. Han, "Credit-scheduled delay-bounded congestion control for datacenters," in *SIGCOMM*, 2017.
[21] M. Handley *et al.*, "Re-architecting datacenter networks and stacks for low latency and high performance," in *SIGCOMM*, 2017.
[22] B. Vamanan, J. Hasan, and T. Vijaykumar, "Deadline-aware datacenter tcp (d2tcp)," in *SIGCOMM*, 2012.
[23] C.-Y. Hong, M. Caesar, and P. B. Godfrey, "Finishing flows quickly with preemptive scheduling," in *SIGCOMM*, 2012.
[24] D. Zats *et al.*, "Detail: Reducing the flow completion time tail in datacenter networks," in *SIGCOMM*, 2012.
[25] L. Chen, K. Chen, W. Bai, and M. Alizadeh, "Scheduling mix-flows in commodity datacenters with karuna," in *SIGCOMM*, 2016.
[26] H. Rezaei *et al.*, "Slytherin: Dynamic, network-assisted prioritization of tail packets in datacenter networks," in *ICCCN*, 2018.
[27] H. Rezaei, M. U. Chaudhry *et al.*, "Icon: Incast congestion control using packet pacing in datacenter networks," in *COMSNETS*, 2019.
[28] K. He *et al.*, "Presto: Edge-based load balancing for fast datacenter networks," in *SIGCOMM*, 2015.