ICON: Incast Congestion Control using Packet Pacing in Datacenter Networks

Hamed Rezaei, Muhammad Usama Chaudhry, Hamidreza Almasi, and Balajee Vamanan
{hrezae2, mchaud30, halmas3, bvamanan}@uic.edu

University of Illinois at Chicago

***Abstract*—Datacenters host a mix of applications which generate qualitatively distinct traffic patterns and impose varying network objectives. Online, user-facing applications generate many-to-one, incast traffic of mostly short flows, which are sensitive to tail of Flow Completion Times (FCT). Data analytics applications generate all-to-all traffic (e.g., Web search) of mostly short flows that saturate network bisection and the job completions require all flows to complete. Background applications (e.g., Map-Reduce) generate large flows and are throughput sensitive due to the sheer amount of data that they transfer over the network. While datacenter fabric provides good bisection bandwidth to handle all-to-all traffic, incast traffic is bottle-necked at edge switches and causes queue buildup at the port connected to the receiver. Because datacenter switches use shallow buffers to reduce cost and latency, the queue buildup problem is further exacerbated as the shallow buffers easily overflow causing packet drops and expensive TCP timeouts. To address these issues we propose ICON, a novel scheme which reduces incast-induced packet loss by setting a fine-grained control over sending rate by pacing traffic. We propose two variants: an application agnostic version that simply paces packet smoothly over round trip time (RTT) and an application aware version that paces packets based on application knowledge (e.g., incast degree). Compared to existing state-of-the-art congestion control schemes, ICON achieves 77% lower 99^{th} percentile flow completion times for short flows and 18% higher throughput for long flows on average. Further, ICON drastically reduces 99^{th} percentile packet drops by a factor of about 3 on average.**

I. Introduction

Today's datacenters provide fast and reliable access to data across the Internet. Datacenters have become the *de facto* platform for storing and accessing vast amounts of Internet data. Datacenters host a mix of applications: foreground applications, which mostly perform distributed lookup in order to respond to user queries; and background applications, which perform data replication, synchronization, and update. Because foreground applications perform distributed look-ups, the queries must wait for *most* responses (e.g., 99%) from servers, and, therefore, their performance is sensitive to the tail (i.e., 99^{th} percentile) flow completion times [1]. While foreground applications generate relatively short flows (e.g., 16KB) and are sensitive to the tail flow completion times, background applications generate long-lasting flows (e.g., VM migrations) and require higher throughput.

Foreground applications perform online queries for *mostly small* data items that are distributed across many servers, and therefore, cause *incast* (i.e., many servers send their data to one receiver which causes rapid queue buildup at the switch port connected to the receiver). As today's datacenters host many incast-heavy applications (e.g., Web search, social networks), incast-induced congestion is common and drastically affects performance. Incast causes packet drops and costly timeouts, resulting in longer tail flow completion times. Incast not only affects flow completion times, but it also decreases throughput as it builds up long queues at switch ports. Long queues and timeouts affect throughput as well. Therefore, the key to improving both flow completion times and throughput lies in alleviating the congestion from incast.

Both load balancing and congestion control play a key role in achieving network objectives. Perfect load balancing is vital because poor load balancing would result in congestion hotspots, which worsen tail flow completion times and throughput due to excessive packet loss. Fortunately, recent proposals achieve near-optimal load balancing [2–4]. On the other hand, congestion control schemes strive to improve tail flow completion by modulating the flow rate using active queue management (AQM) signals such as explicit congestion notification (ECN).

TCP uses packet loss as the main signal of congestion. TCP relies on timeouts or duplicate ACKs to detect packet loss, and, throttles its sending rate when a packet is lost. However, packet loss is often a *late* signal of congestion as it triggers the congestion control mechanism only after packets are already dropped, which is very expensive. Recent congestion control proposals [5–9] address this problem by leveraging ECN, which informs senders about congestion before the buffers overflow. ECN-enabled switches mark packets if their queue lengths exceed a predefined threshold. In fact, many providers have already deployed DCTCP [5] in their datacenters.

While AQM schemes drastically speed up congestion response in the general case, they are still slow to respond to incasts. For instance, we still incur packet drops with an incast-heavy workload in our experiments. The buffers build-up at a much faster rate during incasts than during other congestion episodes. However, the ECN-based schemes require several round-trip times (RTT) to respond. Figure 1 shows the CDF of the duration of incasts from our test run. Around 90% of packet bursts (TCP incast) last only around 40 μs which is of the same order as RTT in many production datacenters. The short incast episodes are also reported by other studies [10].

Congestion control protocols that use ECN (e.g., DCTCP

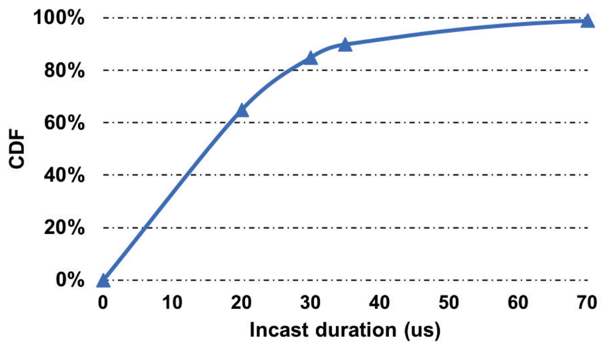

Fig. 1: Incast duration in web-search workload

) improve throughput of background flows as the flows are long and can afford several RTTs for congestion response. However, if packets of short, foreground flows are dropped, the schemes do not improve flow completion times. Note that we cannot solve the problem by aggressively cutting the sending rate upon seeing an ECN mark; intermittent burstiness is common in datacenters, so such an aggressive response would degrade throughput. We cannot solve the problem by setting a smaller ECN threshold as a smaller threshold would also degrade throughput for the same aforementioned reason [11].

In this paper, we target incast congestion, which causes most packet drops in datacenter networks; existing schemes that rely on feedback from the receivers do not address incast congestion. We observe that incast fills up a large fraction of switches' ports in a small amount of time (e.g., a few milliseconds) and are incredibly hard to predict [12]. As our results show, although feedback-based schemes can improve the performance to *some* extent, they are far from achieving optimal flow completion times. In contrast, we propose to alleviate the incast via pro-active means: pacing the senders' packets to smooth-out the bursts that arrive at the switch port and leveraging application knowledge (i.e., incast degree) to *proactively* control the rate. Thus, our proactive approach, ICON, prevents congestion from happening in the first place, as opposed to reacting to congestion in the aftermath.

We present ICON, which proactively alleviates the congestion due to incasts, by pacing packets in a TCP window over RTT, instead of bursting out the packets together, as is done today. While ICON paces packets over RTT, we introduce another optimization to further reduce congestion: we reduce the rate as per the number of incast senders (i.e., incast degree) before pacing the packets (i.e., if n is the incast degree, then $\frac{window}{RTT} = \frac{capacity}{n}$). We leverage the receiver (sink) server to get information about the number of senders. Because the receiver (sink server) is the initiator of the query, it knows the incast degree and it can piggyback this information to *all* senders. Fortunately, modern NICs include support for packet pacing, which makes our idea attractive to datacenters [13, 14].

Our proposed scheme nearly *eliminates* incast-induced packet drops by adding negligible complexity at the end hosts' transport layer. Because incast traffic has an "ON-OFF" pattern [15], ICON needs to remember the incast degree at the sender side and not start with a full line rate at the beginning of "ON" phase. We elaborate our design in III. ICON does not require any support from switches and is readily deployable.

In summary, we make the following contributions:

- We propose an end-to-end congestion control scheme, called ICON, that targets incast, which is common but not efficiently handled by existing proposals.
- We employ pacing to proactively handle incasts.
- Our application-aware optimization uses incast degree to pace senders' packets and nearly *eliminates* packet drops from incast.
- Because packet pacing is widely supported in modern NICs, our proposal is readily deployable in today's datacenters.

Through extensive simulations using ns-3 simulator [16], we show that ICON achieves 77% (4.3x) reduction in 99^{th} percentile flow completion time compared to ICTCP, on average, for all loads. At higher loads (80%), ICON achieves up to 7% and 80% reduction in median and 99^{th} percentile flow completion times, respectively. Further, ICON achieves 18% higher throughput than ICTCP, on average, for all loads.

The rest of the paper organized as follows. We provide background and motivation for our idea in section II. We present design of ICON in section III. We discuss experimental methodology and results in Sections IV and V, respectively. Finally, we discuss related work in section VI and conclude in section VII.

II. Background and motivation

TCP incast was introduced in [17] in the context of distributed storage. Similar to Web-search workload, incast is common in distributed storage clusters when a server requests files that are stored in multiple servers. Several previous papers on congestion control and load balancing show that congestion caused by incast is a major problem in datacenter networks. Recent proposals [12, 18] make changes to transport protocol stack to tackle incast. Because end-to-end designs are easier to deploy than designs that require changes to the network (i.e., routers and switches), end-to-end solutions are the preferred choice.

ICTCP [12] specifically targets incast problem in datacenter networks. ICTCP-enabled end-hosts measure available bandwidth and then set the appropriate receive window based on the calculated throughput. The central idea is to measure the difference between observed throughput and estimated throughput and use the difference to affect the sender rate via TCP *receive window*. ICTCP requires at least 2 RTTs to adjust the window size which is indeed slow to respond to incast. Also, if the initial congestion window is set to a large value (e.g., 10 MSS or higher), ICTCP would still suffer from excessive TCP timeouts as many packets get dropped during the first RTT. Furthermore, while ICTCP performs well if the initial congestion window is set to a small (e.g., 2 MSS in their paper) value, a small congestion window hurts throughput, as we later show in section V.

DCTCP [5] is a well-known congestion control scheme, which makes the key insight that a proportional response to congestion using ECN marks could improve both tail flow completion times and throughput. DCTCP leverages 1-bit ECN marks in packet headers at the end host to infer the queue length at the bottleneck port. It uses this information to modulate the sending window based on the number of marked packets [5]. DCTCP works well for long, background flows. However, if the incast degree (number of parallel senders) is high and the flows are short, DCTCP performs poorly. We compare ICON to DCTCP in section V.

The problem with both DCTCP and ICTCP is that queue size would increase rapidly during incast, and therefore, it is essential to slow down all senders in a sub-RTT scale. However, DCTCP's proportional response would require several RTTs for the senders to slow down and ICTCP requires at least 2 RTTs to react. Therefore, both ICTCP and DCTCP are not fast enough to react tackle incasts. There are other proposals on congestion control [19–21], load balancing [2–4] and packet scheduling [8, 22–24] that focus on minimizing flow completion times and maximizing throughput. While these proposals are effective against other general forms of congestion, they do not handle incasts well. We argue that incasts must be handled separately than other general forms of congestion.

While rate control is one aspect of incast-induced congestion, the burstiness of data transmission from senders further worsen the congestion episodes. A TCP sender, without pacing, causes burstiness at the switch output port. The lack of pacing combined with synchronized sends from several servers (i.e., a search query synchronizes the responses from several servers that results in an incast) results in over-subscription of switch port's capacity. Therefore, pacing packets is a straightforward way to alleviate incast congestion.

Because most incasts lasts shorter than a RTT [10] (see figure 1), a congestion control scheme that handles incasts must be proactive: the senders must set their *correct* rates *without* relying on feedback from receivers. Prediction based on past statistics is a possibility. Unfortunately, previous papers have shown that incasts are notoriously unpredictable [12]. Therefore, we require hints from the application (e.g., incast degree) to set the correct sending rates. Fortunately, the software architecture of incast-heavy applications is such that it is quite feasible to obtain incast degree for these applications. For instance, incasts in Web Search application is caused when responses to a query collide at a switch's output port. Because the (receiving) server already knows the incast degree when it issued the query, it is only a matter of passing this information to the senders along with the query so that senders can start at the correct sending rate rather than having to learn the correct rate iteratively over several RTTs.

III. ICON

In this section, we present the technical details of *ICON*, which specifically targets incast problem in datacenter networks. Incast is predominately induced by a set of servers (e.g., 32) that respond to a server's query. For example, in distributed storage clusters, a file may be divided into multiple chunks that are stored on different servers across the datacenter. In such a case, a single server, which is responsible for retrieving and reassembling the whole file, queries *all* the servers to retrieve the file. Such distributed lookup results in incast. Incast causes many packet drops at the edge switch connected to the retrieving (sink) server. As we discussed in section I, incast is an important and critical issue in datacenter networks, and incasts degrade the performance of both foreground and background applications.

ICON's incast control algorithm consists of two parts: (1) end-to-end application level knowledge sharing about the number of concurrent senders (i.e., incast degree), and (2) pacing TCP window over RTT proportional to the shared number. We will discuss the application-level information sharing in section III-A and the traffic pacing in section III-B.

A. Application-level knowledge sharing

We make the key observation that incast, which bottlenecks the receiver (or the last switch's output port that connects to the receiver), it is inherently application-depended, and, therefore, the incast problem could be solved at end hosts rather than involving switches and routers.

The application layer of the sink server, which is responsible for distributed lookup, knows the number of the servers that it is going to the search query. Therefore, simply communicating this *incast degree* to all those servers that are part of the distributed lookup could go a long way in addressing incasts. At the very beginning of the communication between all senders and the sink server, this number will be piggybacked by TCP packets from sink server to sender servers (during TCP 3-way handshaking). At this point, all senders know how many different servers will share a same route through the sink server. Giving this prior knowledge to servers is key as it can serve to right the correct sending rate at the sender. However, there are a few challenges that we must solve to realize the goal:

Communicating incast degree to senders. We can use unused fields in IP or TCP header for this purpose. For example, we can use "options" field in TCP header or "Type-of-Service" (ToS) field in IP header but we chose the first one as it does not affect the network functionality.

Handling persistent connections. We know the calculated number of servers will be reported to sender servers only during the TCP handshake. If the TCP connection between end hosts is persistent, sender servers save the reported number in their memory and use it unless a new information arrives from the sink server. In the other words, the very first reported number will be valid until a change happens in the number of active servers. This number rarely changes because the number of senders is usually fixed as they are chosen to hold a particular piece of data. However, if this number changes, the sink server notifies other servers through a small update message. These update messages are designed to only convey the number of parallel senders. As long as only *one* server

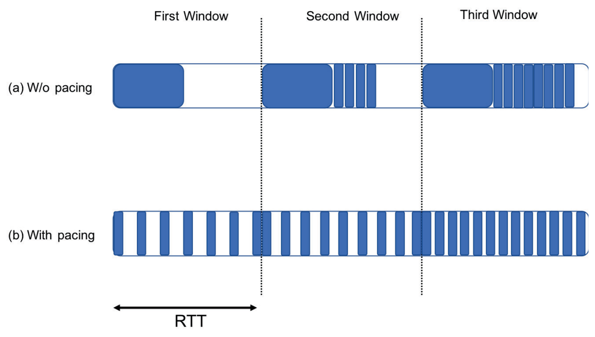

Fig. 2: (a) Normal TCP vs (b) TCP with Pacing

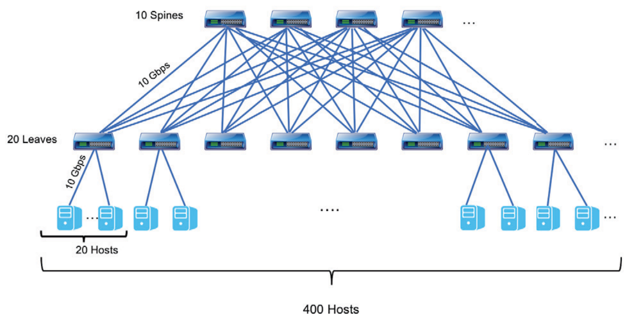

Fig. 3: Topology used in simulations

will send these small update packets, bandwidth consumption is almost negligible.

B. Traffic pacing

In this section, we cover details of traffic pacing at sender NICs. Pacing and rate control are two *different* aspects of congestion control. While rate control adjusts the sending rate to avoid any long-term over-subscription of the bottleneck link capacity, rate control alone is not sufficient to avoid congestion. Pacing determines controls the rate at a much more finer grain (e.g., how the packets are spaced within an RTT). Two schemes with the same sending rate but different packet pacing can lead to different queuing delays in the network. In general, uniformly distributing the packets over the sending "time period" can smooth out queue build-ups and reduce congestion. Figure 2 shows the difference between Normal TCP and the one which uses pacing.

ICON paces TCP window over RTT by providing a gap (pause time) between packets. As we can see in the figure 2, as TCP window grows, this gap becomes smaller because of more packets that it needs to send. Since TCP normally sends a window of packets as a bulk, it is more prone to cause packet drops as more packets are likely to collide on the bottleneck link. However, ICON paces packets over RTT to provide more time for the switch port at the bottleneck link to drain packets.

C. Combination of pacing and application knowledge sharing:

ICON uses a combination of application knowledge sharing and traffic pacing in order to improve tail flow completion time and throughput. Once the sender servers receive the number of active senders from the sink server, they start to calculate the pacing rate (i.e., the gap between each packet) using this number. Equation 1 shows how ICON calculates the new sending rate.

$$Sending\_rate = \frac{TCP\_window\_size}{RTT \times number\_of\_parallel\_senders} \tag{1}$$

As equation 1 implies, if TCP window grows up, the sending rate will increase which might be problematic if the flow size is very large. However, this is not a big issue in incast as incast is mostly caused by short flows. The newly calculated sending rate ensures that the sending rate of all senders is proportional to the number of competing flows on the bottleneck link. This technique guarantees that the bottlenecked switch port buffer has enough space to absorb all packets heading to the sink server. As an example, if TCP window is 10 MSS, RTT is 120 microseconds and we have only 1 sender, the new sending rate requires 10.8 microseconds gap (pause time) between sending each packet (assuming link speed is 10 Gbps). As you can see, our new congestion control scheme requires only a tiny modification over current transport protocols and is implementation-friendly.

In the absence of application knowledge, we assume that the "$number\_of\_parallel\_senders = 1$" in the previous equation. We show in section V that ICON outperforms other schemes, even without application knowledge.

IV. Experimental Methodology

We use ns-3 [16] simulator to simulate a datacenter network and evaluate ICON's performance. In this section, we present the details of our evaluation methodology including topology, workload, simulation parameters, etc.

1) Topology: We use ns-3 to simulate a *leaf-spine* datacenter topology as shown in Figure 3. Leaf-spine is a commonly used topology in many datacenters [3]. In our topology, the fabric interconnects 400 servers through 20 leaf switches and 10 spine switches. Our topology uses an *over-subscription factor* of 2, which is typical. So, each of those leaf switches has 20 ports connected to the servers and 10 ports connected to upper spine switches. All the links from servers to leaf switches, and from leaf switches to spine switches are 10 *Gbps* links. The maximum unloaded round trip time (RTT) of our network is 80 μs.

2) Workload: We model our workloads based on the results presented in [15]. Our workload has a mix of short and long flows. The flow arrivals are based on a Poisson process and source and destination servers are chosen uniformly randomly for both short and long flows. We define incast flows as a subset of short flows; the total load produced by short flows is equally divided among incast flows and other normal short flows (one-to-one short flows). Our workload model matches the one used in [25]. Our short flow sizes are randomly selected from a range of 8 KB to 32 KB and we set long flow size at 1 MB. Our long flows contribute to 70% of the overall network load, which matches previous work (e.g., pFabric [26]). We use a default *incast degree* of 26, but we vary the incast degree and study our sensitivity in section V. We evaluate ICON's performance using both tail and median flow completion times for short flows, and throughput for long flows.

3) Compared schemes: We compare ICON to the following schemes:

- **DCTCP:** We implemented DCTCP, matching all the details from the original paper [5]. We set DCTCP to be our baseline, as it is commonly used in today's datacenters.
- **ICTCP:** We implemented ICTCP based on their paper and we set the parameters as suggested in the paper [12]. Our ICTCP implementation measures available bandwidth at end hosts and sets the next receive window according to the available bandwidth. In the other words, the end host predicts the throughput and sets the receiver window based on the difference between expected throughput and the observed throughput. The receive window will be set to a smaller value if the difference is large. However, if the measured difference is small, TCP receive window is allowed to increase normally. ICTCP is able to set the fair share rate when a lot of packets arrive at a same time. ICTCP specifically targets incast.
- **ICON:** While ICON would work on top of any transport protocol, we implement it on top of DCTCP [5] as it is widely deployed in modern datacenters. Our proposed scheme consists of two parts. First, each server which is a part of TCP incast, paces its TCP window over RTT by setting appropriate pause times between sending each packet. Many modern NICs already support this feature [14]. Second, to find the best sending rate and pause duration for pacing, senders divide their sending rate by the number of parallel senders which is provided by the receiver (we describe the design in section III).
- **Information-agnostic ICON (IA-ICON):** We implement another version of ICON which does not rely on information provided by receiver application (i.e., number of parallel senders). Instead, *IA-ICON* paces the TCP window over RTT. While this scheme will pace sender data regardless of the number of senders, we include this method for two reasons: (1) quantify the utility of application knowledge and (2) isolate the effect of pacing from rate control (i.e., IA-ICON does not start with the correct sending rate but ICON does).

Finally, we would like to emphasize that we use identical values for parameters that are common to DCTCP, ICTCP, and ICON (e.g., ECN threshold, buffer size, RTT). While there are a number of packet scheduling and load balancing schemes in the last few years such as pFabric [26], PIAS [22], CONGA [3], HULA [4], etc., we only compare ICON to DCTCP and ICTCP as they target incasts. As such, ICON can be implemented on top of most load balancing schemes.

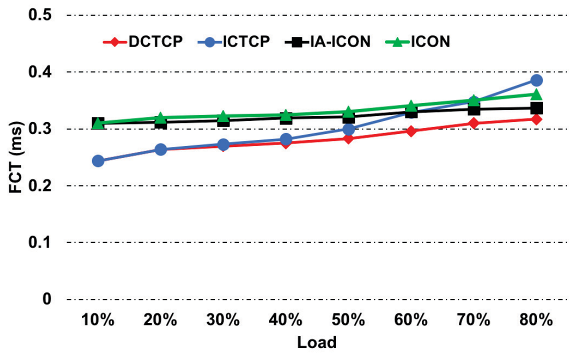

Fig. 4: Median flow completion time

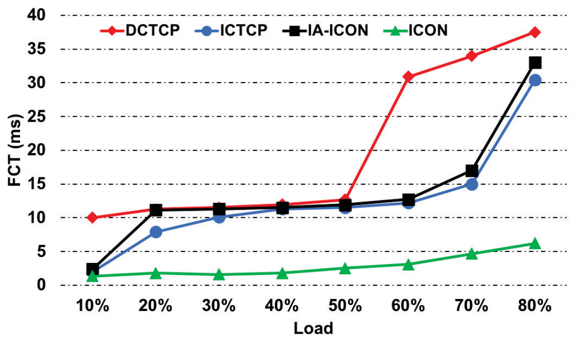

Fig. 5: Tail flow completion time

V. Results

In this section, we evaluate performance of ICON and compare its performance to other competing schemes. We present the following comparisons:

- Tail (99^{th} percentile) and median (50^{th} percentile) flow completion times
- Average throughput of long flows
- Number of dropped packets
- Sensitivity to different incast degrees
- Distribution of queue lengths in switches

A. Flow completion time

In this section, we show how ICON performs in terms of median and tail flow completion times compared to other schemes. The results for median and tail flow completion times are shown in figure 4 and figure 5, respectively. In both of these figures, X-axis shows the network load and Y-axis shows flow completion times in milliseconds. As expected, both tail and median flow completion times of all schemes increase with increasing load. We also see 1-2 orders of magnitude difference between median and tail flow completion times, matching several reported results [5, 10, 12]. Also, we see that tail rapidly increases at higher load values, which is also expected.

While ICON slightly under-performs other schemes (i.e., by about 13 %) in median flow completion times, it *drastically*

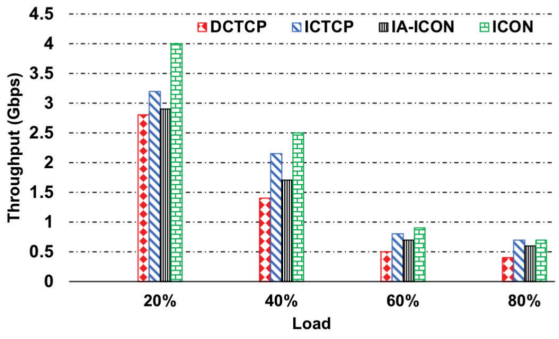

Fig. 6: Average throughput of long flows

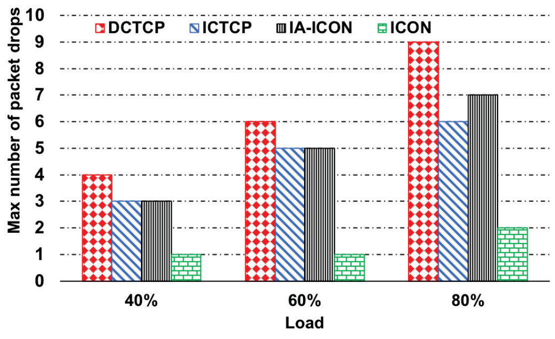

Fig. 7: Maximum number of packet drops

outperforms (i.e., by about a factor of $7-8$ or by about 80%) all other schemes in tail flow completion times. Because most foreground applications are sensitive to tail, not median flow completion times, ICON's contribution is well-suited to today's datacenters. In other words, ICON closes the gap between median and tail latencies, which reduces performance variability in datacenters. Also, it is interesting to note that IA-ICON performs quite close to ICTCP in tail flow completion times. Because IA-ICON isolates pacing from rate control, the fact that IA-ICON performs close to other complicated rate control schemes shows that pacing is as important as rate control in datacenter networks. The slightly worse median flow completion times, however, do not affect ICON's throughput, as we show next.

B. Average Throughput

In this section, we compare the average throughput of long flows for DCTCP, ICTCP, Information-Agnostic ICON, and ICON. As we can see in figure 6, as the load increases, long flow throughput decreases for all schemes due to congestion in the network created by short, bursty flows; the queuing in the network increases with load. ICTCP outperforms DCTCP, specifically at higher loads. ICON *significantly* outperforms all schemes across a range of loads. Because ICON leverages application knowledge, ICON instantly converges to the correct sending rate; the other schemes require several RTTs to converge to the correct fair-share sending rate. On average, ICON achieves 18% higher throughput compared to ICTCP. IA-ICON performs similar to ICTCP, albeit with a much lesser complexity. Therefore, our throughput results also highlight the importance of pacing.

C. Number of dropped packets

We dissect the bottomline performance of ICON by comparing the number of dropped packets in all the schemes. Because re-transmission of dropped packets directly influences tail flow completion times, we compare the maximum number of packet drops for different schemes. As the load increases, packet drop rate for all schemes increases due to increased congestion. As we see in figure 7, ICON substantially cuts the number of packet drops. Since ICON sets the right sending rate and paces the packets, ICON nearly eliminates packet drops.

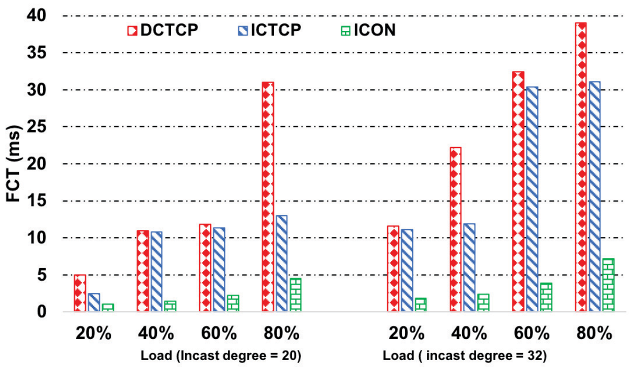

Fig. 8: Sensitivity to different incast degrees

D. Sensitivity to different incast degrees

We now study the sensitivity of ICON and other schemes as we vary the incast degree of applications. While our default incast degree value of 26 is typical for many datacenter applications, there is, indeed, a wide range in the incast degrees of various applications. In this experiment, we compare the tail flow completion times of foreground flows at an incast degree of 20 and 32 to those at an incast degree of 26, for varying loads. As shown in figure 8, the tail increases with higher incast degrees. Nevertheless, we see that ICON substantially outperforms the competition, even for lower incast degrees. Because the "real" incast degrees of commercial applications are likely higher (e.g., NDP [21] uses incast degrees in the order of a few hundred), ICON's potential improvements are likely to be even more impressive in production environments.

E. Queue length

Finally, we analyze the distribution of receiver switch's queue length for the three schemes. We sampled the queue lengths in one of the leaf switches connected to a sink server at 30% load, and we analyzed its cumulative distribution.

Figure 9 shows the CDF of DCTCP, ICTCP, and ICON. We see a *drastic* difference between the CDFs of these three schemes. While both DCTCP and ICTCP suffer from long tails in queue lengths, which corresponds to their long tail flow completion times, ICON's CDF is much narrower than

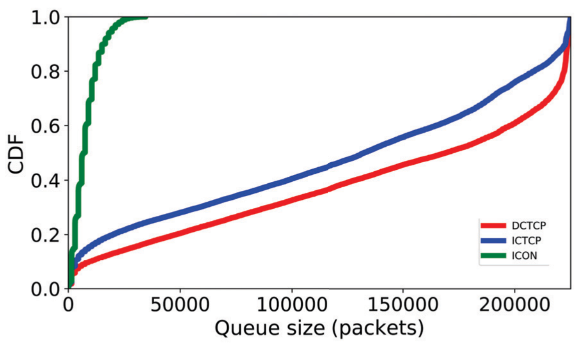

Fig. 9: Distribution of queue lengths in switches

the other schemes. ICON reduces 99th percentile queue length by a factor of 7.5 compared to ICTCP, *closely* matching our improvement in tail flow completion times. This analysis shows that both rate control and pacing play a key role in reducing queuing in the network.

VI. Related Work

Internet congestion control is a well-studied area for more than two decades. Over the last several years, datacenter congestion control has become a hot topic as datacenters are more amenable for adding new functionality in the network and datacenter workloads significantly differ from Internet client-server applications. Datacenter traffic is more bursty and less predictable than wide area network (WAN) and Internet traffic.

We have already discussed DCTCP and ICTCP in previous sections. Now we summarize other related work in this section. [27] focus on reducing timeouts during incast and addresses incast via techniques such as selective ACK (SACK) and disabling TCP slow start. However, incasts last less than one RTT and optimization do not effectively address congestion during incast. Similarly, [25] proposes to decreases TCP's retransmission timeout interval (RTO_{min}) at the end hosts, to reduce the duration of timeouts. While the proposal reduces the cost of timeouts, it does not fundamentally reduce the number of TCP timeouts, which happen due to incast congestion. In contrast, ICON reduces the number of such timeouts.

There are other proposals which argue for better adjusting TCP's congestion window, to reduce congestion during incasts. While smaller congestion windows reduce incast congestion, they also degrade throughput of long flows. In rate control protocol (RCP) [19], the switches directly inform the senders about their fair share rate by observing flows on the ingress link. However, RCP requires expensive switch support. Further, RCP also requires several RTTs to arrive at the right sending rate. TIMELY [7] monitors RTT, similar to TCP Vegas. However, unlike Vegas, TIMELY employs a gradient-based approach to quickly arrive at the right sending rate. In contrast to TIMELY, which also requires several rounds, ICON uses application knowledge and employs pacing to reduce incast congestion.

Similar to DCTCP, DCQCN [6] is a congestion control method which leverages ECN for Remote Direct Memory Access (RDMA) networks and performs rate-based congestion control. Also, QCN [28] provides congestion control based on network feedback (e.g., ECN) but operates at the layer-2. Express-Pass [29] is one of the most recent proposals on datacenter congestion control, which distributes end-to-end credit packets to simulate a token bucket scheme. Express-Pass improves tail flow completion times but requires support at the switches and the implementation is significantly more complex than ICON. A few other proposals [8, 23, 26, 30] focus on datacenter flow scheduling, whereas our main goal is to alleviate incast congestion.

VII. Conclusion

In this paper, we proposed ICON with a goal of improving tail latency for foreground applications and throughput for background applications, by avoiding queue buildups that happen due to incast at edge switches. While there has been a number of papers that show incast-induced congestion is common in datacenters, the problem remains largely unsolved. Incast is also inherently tied to the nature of many foreground applications that perform distributed search of small objects in large datasets. Incast cannot be fixed by having large buffers as large buffers cause increased queuing delays and exacerbate the problem even further.

We introduced ICON, which employs fine-grained traffic pacing at the end host's NIC. Fortunately, many vendors are starting to introduce support for packet pacing in NICs, which augurs well for our proposal. In addition to pacing, ICON also leverages application knowledge to arrive at the right sending rate, without having to continuously adjust the sending rates over several RTTs. ICON reduces the tail percentile flow completion times of short flows by *up to* 89% compared to DCTCP. It also improves the throughput of large flows by up to 30% compared to DCTCP. As incast degrees of foreground applications increase to cope up with the exponential growth of data, schemes such as ICON would become attractive.

References

[1] J. Dean and L. A. Barroso, "The tail at scale," *Commun. ACM*, vol. 56, no. 2, pp. 74–80, Feb. 2013.

[2] K. He, E. Rozner, K. Agarwal, W. Felter, J. Carter, and A. Akella, "Presto: Edge-based load balancing for fast datacenter networks," ser. SIGCOMM '15, 2015, pp. 465–478.

[3] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese, "Conga: Distributed congestion-aware load balancing for datacenters," in *Proceedings of the 2014 ACM Conference on SIGCOMM*, ser. SIGCOMM '14, 2014.

[4] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford, "Hula: Scalable load balancing using programmable data planes," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '16. ACM, 2016, pp. 10:1–10:12.

[5] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, “Data center tcp (dctcp),” in *Proceedings of the ACM SIGCOMM 2010 conference*, ser. SIGCOMM ’10. New York, NY, USA: ACM, 2010, pp. 63–74.

[6] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang, “Congestion control for large-scale rdma deployments,” in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM ’15. ACM, 2015, pp. 523–536.

[7] R. Mittal, V. T. Lam, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, and D. Zats, “Timely: Rtt-based congestion control for the datacenter,” in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM ’15. ACM, 2015, pp. 537–550.

[8] B. Vamanan, J. Hasan, and T. Vijaykumar, “Deadline-aware datacenter tcp (d2tcp),” in *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM ’12, 2012.

[9] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron, “Better never than late: meeting deadlines in datacenter networks,” in *Proceedings of the ACM SIGCOMM 2011 conference*, ser. SIGCOMM ’11. New York, NY, USA: ACM, 2011, pp. 50–61.

[10] Q. Zhang, V. Liu, H. Zeng, and A. Krishnamurthy, “High-resolution measurement of data center microbursts,” in *Proceedings of the 2017 Internet Measurement Conference*. ACM, 2017, pp. 78–85.

[11] W. Bai, L. Chen, K. Chen, and H. Wu, “Enabling ecn in multi-service multi-queue data centers,” in *Proceedings of NSDI*, 2016, pp. 537–549.

[12] H. Wu, Z. Feng, C. Guo, and Y. Zhang, “Ictcp: Incast congestion control for tcp in data center networks,” in *Proceedings of CoNEXT*, 2010, pp. 13:1–13:12.

[13] S. Radhakrishnan, Y. Geng, V. Jeyakumar, A. Kabbani, G. Porter, and A. Vahdat, “Senic: Scalable nic for end-host rate limiting.” in *NSDI*, vol. 14, 2014, pp. 475–488.

[14] “Intel ethernet controller x710.” [Online]. Available: www.intel.com/content/dam/www/public/us/en/documents/datasheets/xl710-10-40-controller-datasheet.pdf

[15] T. Benson, A. Akella, and D. A. Maltz, “Network traffic characteristics of data centers in the wild,” in *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, ser. IMC ’10. New York, NY, USA: ACM, 2010, pp. 267–280.

[16] “NS-3 network simulator,” http://www.nsnam.org/.

[17] D. Nagle, D. Serenyi, and A. Matthews, “The panasas activescale storage cluster: Delivering scalable high bandwidth storage,” in *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, 2004, p. 53.

[18] E. Krevat, V. Vasudevan, A. Phanishayee, D. G. Andersen, G. R. Ganger, G. A. Gibson, and S. Seshan, “On application-level approaches to avoiding tcp throughput collapse in cluster-based storage systems,” in *Proceedings of the 2nd international workshop on Petascale data storage: held in conjunction with Supercomputing’07*. ACM, 2007, pp. 1–4.

[19] N. Dukkipati, M. Kobayashi, R. Zhang-Shen, and N. McKeown, “Processor sharing flows in the internet,” in *Proceedings of the 13th International Conference on Quality of Service*, ser. IWQoS’05. Springer-Verlag, 2005, pp. 271–285.

[20] I. Cho, K. Jang, and D. Han, “Credit-scheduled delay-bounded congestion control for datacenters,” in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 2017, pp. 239–252.

[21] M. Handley, C. Raiciu, A. Agache, A. Voinescu, A. W. Moore, G. Antichi, and M. Wójcik, “Re-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*. ACM, 2017, pp. 29–42.

[22] W. Bai, K. Chen, H. Wang, L. Chen, D. Han, and C. Tian, “Information-agnostic flow scheduling for commodity data centers.” in *NSDI*, 2015, pp. 455–468.

[23] L. Chen, K. Chen, W. Bai, and M. Alizadeh, “Scheduling mix-flows in commodity datacenters with karuna,” ser. SIGCOMM ’16, 2016.

[24] H. Rezaei, M. Malekpourshahraki, and B. Vamanan, “Slytherin: Dynamic, network-assisted prioritization of tail packets in datacenter networks,” in *2018 27th International Conference on Computer Communication and Networks (ICCCN)*. IEEE, 2018, pp. 1–9.

[25] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. G. Andersen, G. R. Ganger, G. A. Gibson, and B. Mueller, “Safe and effective fine-grained tcp retransmissions for datacenter communication.”

[26] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, “pfabric: Minimal near-optimal datacenter transport,” in *SIGCOMM*, ser. SIGCOMM ’13. New York, NY, USA: ACM, 2013, pp. 435–446.

[27] A. Phanishayee, E. Krevat, V. Vasudevan, D. G. Andersen, G. R. Ganger, G. A. Gibson, and S. Seshan, “Measurement and analysis of tcp throughput collapse in cluster-based storage systems.” in *FAST*, vol. 8, 2008, pp. 1–14.

[28] R. Pan, B. Prabhakar, and A. Laxmikantha, “Qcn: Quantized congestion notification an overview,” 2007.

[29] I. Cho, K. Jang, and D. Han, “Credit-scheduled delay-bounded congestion control for datacenters,” in *Proceedings of SIGCOMM*, 2017, pp. 239–252.

[30] C.-Y. Hong, M. Caesar, and P. B. Godfrey, “Finishing flows quickly with preemptive scheduling,” in *SIGCOMM*, ser. SIGCOMM ’12. New York, NY, USA: ACM, 2012, pp. 127–138.