PAC: Taming TCP Incast Congestion using Proactive ACK Control

[1]Wei Bai, [1]Kai Chen, [2]Haitao Wu, [3]Wuwei Lan, [1,4]Yangming Zhao
[1]The SING Lab, CSE Department, Hong Kong University of Science and Technology
[2]Microsoft [3]USTC [4]UESTC

***Abstract*—TCP incast congestion which can introduce hundreds of milliseconds delay and up to 90% throughput degradation, severely affecting application performance, has been a practical issue in high-bandwidth low-latency datacenter networks. Despite continuous efforts, prior solutions have significant drawbacks. They either only support quite a limited number of senders (*e.g.*, 40-60), which is not sufficient, or require non-trivial system modifications, which is impractical and not incrementally deployable.**

We present PAC, a simple yet very effective design to tame TCP incast congestion via Proactive ACK Control at the receiver. The key design principle behind PAC is that we treat ACK not only as the acknowledgement of received packets but also as the trigger for new packets. Leveraging datacenter network characteristics, PAC enforces a novel ACK control to release ACKs in such a way that the ACK-triggered in-flight data can fully utilize the bottleneck link without causing incast collapse even when faced with over a thousand senders.

We implement PAC on both Windows and Linux platforms, and extensively evaluate PAC using small-scale testbed experiments and large-scale ns-2 simulations. Our results show that PAC significantly outperforms the previous representative designs such as ICTCP and DCTCP by supporting 40X (*i.e.*, $40 \rightarrow 1600$) more senders; further, it does not introduce spurious timeout and retransmission even when the measured 99^{th} percentile RTT is only 3.6ms. Our implementation experiences show that PAC is readily deployable in production datacenters, while requiring minimal system modification compared to prior designs.

I. Introduction

Driven by economics and technology, datacenters are being built around the world to support various services and applications. TCP, as the *de facto* reliable transport layer protocol, plays an important role in datacenter communications.

However, under high-bandwidth low-latency and shallow-buffered datacenter environments, TCP is sensitive to incast congestion collapse when multiple senders send data to a receiver (*i.e.*, many-to-one communication) synchronously [4, 9, 25, 29]. As the number of senders increases, the shallow buffer on the ToR (Top-of-Rack) swtiches can easily become overfull [31], causing packet drops which lead to TCP timeouts. TCP timeouts would impose hundreds of milliseconds retransmission delay and up to 90% bottleneck link throughput reduction [29]. This would severely degrade the application performance especially those involving *barrier-synchronized* communications such as MapReduce [11], Spark [33], Dryad [17], and large-scale partition/aggregate web applications [4, 30].

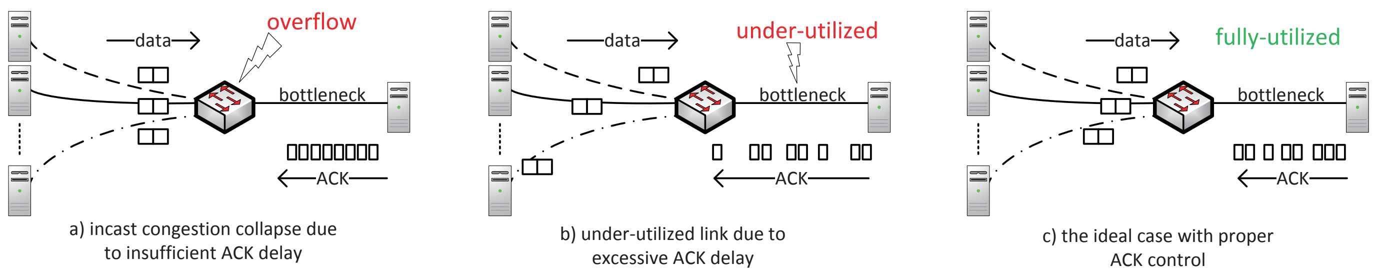
Due to the impact of the TCP incast problem, there have been continuous efforts in the community to address it. They can be generally divided into two categories: window-based solutions [4, 31] and recovery-based solutions [24, 29, 34]. While significant progress has been made, the prior work in both categories have important drawbacks.

Window-based solutions, such as ICTCP [31] and DCTCP [4], throttle aggregate throughput by decreasing the TCP window (congestion window or receiver window), in order to avoid overfilling the switch buffer and packet losses. However, these designs are fundamentally constrained by the number of senders they can support, *e.g.*, 40–60 [4, 31]. This is far from sufficient to sustain real datacenter communications. For example, a cluster with 1500 servers running data mining jobs sees over 80 concurrent flows per node [13]; A web server may access several hundred memcached servers for a single request in Facebook's memcached cluster [23]; Worse, a production datacenter with 6000 servers supporting web search application witnesses over 1000 concurrent flows for a worker node [4]. Under all these cases, even a minimal window of 1 MSS for each flow is sufficient to overwhelm the switch buffer on a synchronized burst.

Recovery-based solutions, such as reducing-RTO$_{min}$ [29], GIP [34] and CP [24], mainly focus on reducing the impact after packet losses. Reducing-RTO$_{min}$ [29] implements a fine-grained TCP RTO mechanism to reduce unnecessary long waiting time after packet loss. GIP [34] eliminates timeouts by modifying TCP stack to guarantee important packets. CP [24] achieves rapid packet loss notifications with modifications on both the TCP stack and switches. By design, these schemes can support many more senders and potentially solve the above problem. However, they entail non-trivial system modifications which are not readily deployable. Furthermore, there is still an open question of whether these modifications will affect normal behaviors of TCP and increase system overhead using the high resolution timer [29].

As a consequence, today's incast control is facing a dilemma: while the window-based solutions are incrementally deployable, they are not sufficient for real communications. On the other hand, while the recovery-based solutions are able to to handle a larger synchronized burst, unfortunately, they are not readily deployable.

The main contribution of this work is to solve the above dilemma by proposing PAC, a simple yet very effective and readily deployable incast control algorithm. To the best of our knowledge, PAC is the first design that achieves both goals simultaneously.



overflow
data
bottleneck
ACK
a) incast congestion collapse due to insufficient ACK delay
under-utilized
data
bottleneck
ACK
b) under-utilized link due to excessive ACK delay
fully-utilized
data
bottleneck
ACK
c) the ideal case with proper ACK control

Fig. 1. The general idea of PAC– an illustrative example.

The key design principle behind PAC is that we treat ACK not only as the acknowledgement of received packets but also as the trigger for new packets. Based on this, PAC proactively intercepts and releases ACKs in such a way that ACK-triggered in-flight traffic can fully utilize the bottleneck link without causing incast congestion collapse. This general idea is illustrated in Figure 1.

However, fully utilizing the bottleneck link without causing buffer overflow is a challenge. As shown in Figure 1, if PAC imposes excessive delay to ACKs, it can eliminate buffer overflow, but the link utilization and throughput may be degraded (Figure 1(b)). If the imposed delay is not sufficient, the buffer may still be overfilled by the ACK-triggered traffic and incast collapse can still happen (Figure 1(a)).

To solve this challenge, PAC uses the switch buffer size value as the initial threshold to modulate our ACK control rhythm. Then the core idea of the PAC algorithm is simply to wait to send back outstanding ACKs as long as the ACK-triggered in-flight traffic does not exceed the threshold. This simple threshold setting is inspired by the practical observations of production datacenter network characteristics, and is provably able to fully utilize the bottleneck link without overfilling the switch buffer (see Section III-B for details). Our experiments also validate that it works quite well in practice. In addition, PAC leverages ECN [26], a function widely available on commodity switches, to address the case when the network core experiences severe persistent congestion.

We have implemented PAC as a NDIS (Network Driver Interface Specification) filter driver for the Windows platform and as a kernel module for the Linux platform. Our implementation is located between TCP/IP stack and NIC (Network Interface Card) driver as a shim layer. It does not touch any TCP/IP implementation, naturally supporting various OS versions. In virtualized environments, PAC resides in hypervisor (or Dom 0), well supporting virtual machines.

We build a testbed with 38 Dell servers and 2 Broadcom Pronto-3295 Gigabit Ethernet switches. In our experiments, we rearrange the testbed topologies so that incast can happen both at the edge ToRs and in the network core. Our results show that PAC can easily handle many more senders than that of DCTCP and ICTCP, while maintaining around 900 Mbps goodput. Even in the case of severe network core congestion, PAC can still sustain over 700 Mbps goodput while DCTCP degrades to 100 Mbps goodput due to incast collapse (Note that ICTCP is designed only for the edge incast not the core). Furthermore, our measurements suggest that PAC's ACK control does not adversely prolong RTT and introduces little system overhead. For example, in our experiments, the measured RTT is only 1.3ms at the 50th percentile and 3.6ms at the 99th percentile with 100 concurrent connections. The additional CPU and memory overhead PAC introduced on the receiver are around 1%-1.6% and tens of KBs respectively.

To complement the small-scale testbed experiments, we also perform large-scale simulations with a large number of senders on simulated 1G and 10G networks using ns-2 [3]. Our results further demonstrate that PAC significantly outperforms both ICTCP and DCTCP by supporting 40X (*i.e.*, $40 \rightarrow 1600$) more senders.

The rest of the paper is organized as follows. Section II discusses the background and motivation. Section III presents the design. Section IV introduces the implementation and testbed setup. Section V evaluates PAC through testbed experiments and ns-2 simulations. Section VI discusses the related work. Section VII concludes the paper.

II. Background and Motivation

In this section, we first introduce the TCP incast problem. Then, we discuss two kinds of existing solutions to this problem: window-based solutions and recovery-based solutions respectively. Finally, we discuss some practical observations that motivate our ACK control design to the incast problem.

A. TCP Incast Problem

TCP incast congestion happens when multiple senders send data to a same receiver synchronously. The synchronized bursty TCP flows from many senders converge at the same port of the switch to the receiver and overfill its shallow buffer quickly, leading to TCP packet drops and timeouts. TCP timeouts impose hundreds of milliseconds delay and degrade TCP throughput greatly.

TCP incast problem was initially identified in cluster file systems [22] and has nowadays become a practical issue in datacenter networks. The special characteristics of datacenter networks naturally create three preconditions for TCP incast congestion [29]. First, datacenter networks are designed to achieve high-bandwidth and low-latency using shallow-buffered switches. Second, barrier-synchronized many-to-one traffic is quite common in datacenters, such as the shuffle phase of many cloud computing systems [11, 17, 33] and partion/aggregate design pattern of many large-scale web applications [4, 30]. Third, the traffic volume may be small.

B. *Window-based Solutions*

The window-based solutions such as DCTCP [4] and ICTCP [31] have been proposed to mitigate TCP incast congestion. The key idea of those window-based solutions is to adjust the congestion or receive window to control in-flight traffic, so that it will not overfill the switch buffer.

DCTCP [4] leverages ECN [26], which has been well supported in commodity switches, to deliver congestion information. DCTCP provides much better bursty tolerance than TCP for incast flows by marking packets to deliver congestion notification when instant switch buffer occupation exceeds a threshold. Further, Tuning ECN [32] accelerates the delivery of congestion notification using dequeue marking instead of traditional enqueue marking.

ICTCP [31], on the other hand, adaptively adjusts the receive window on the receiver side to throttle aggregate throughput. ICTCP measures available bandwidth and per-flow throughput in each control interval. It only increases the receive window when there is enough available bandwidth and the difference of measured throughput and expected throughput is small.

However, the window-based designs are fundamentally constrained because they cannot reduce the window size infinitely (*e.g.*, no less than 1 MSS (Maximum Segment Size)). Therefore, the number of senders they can support is limited. Suppose that there are N incast flows with the same round-trip times RTT, sharing a bottleneck link with the capacity of C and switch buffer size of B. The window size of flow i is $W_i(t)$. The queue size in switch at time t is given by:

$$Q(t) = \sum_{i=1}^{N} W_i(t) - C \times RTT \qquad (1)$$

where $W_i(t) \geq MSS$. Then we have $Q(t) \geq N \times MSS - C \times RTT$. If $Q(t)_{min} = N \times MSS - C \times RTT$ is still larger than the switch buffer size B, no window-based solution will help. Thus, the maximum number of senders that window-based solutions can support is roughly $(B + C \times RTT)/MSS$.

To validate this, we conducted a testbed experiment. In the experiment, we have each sender sending 64KB data to a receiver. The congestion/receive window is set to 1 MSS, and the switch buffer size set to 80K and 100K respectively. We increase the number of senders and depict the results in Figure 2. We can see that the maximum number of senders supported are around 50 and 62 respectively, which confirms our derivation above and also matches the results of recent works [4, 31, 32]. However, such a performance is far from the real requirements of communication as we showed previously.

C. *Recovery-based Solutions*

Unlike window-based solutions, recovery-based solutions address incast congestion via reducing the impact of timeouts. Vasudevan et al. [29] leverages high-resolution timers to implement microsecond-granularity TCP retransmission timeout

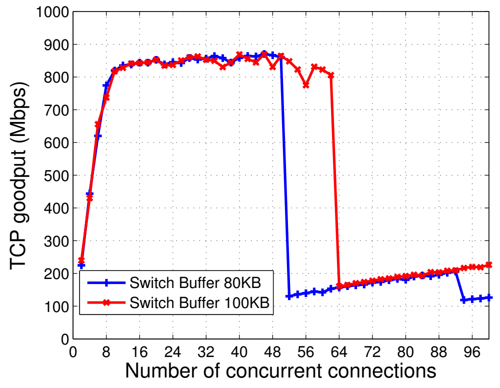


Fig. 2. The limitation of the window-based solutions: they only support tens of senders even with the minimal window size of 1 MSS.

mechanism. In this way, TCP can retransmit lost packets quickly without leaving the link idle for a long time. Another recent work GIP [34] is built on the observation that two kinds of timeouts, FLoss-TOs and LAck-TOs, should be avoided to improve goodput. Then, it proposed an enhanced mechanism to eliminate these two kinds of timeouts. TCP with GIP is less likely to suffer from timeouts even though packet losses still happen. CP [24] simply drops a packet's payload at an overloaded switch and uses a SACK-like ACK mechanism to achieve rapid and precise notification of lost packets.

By design, these recovery-based solutions can handle generic incast cases with a large number of senders. However, they require non-trivial modifications on existing TCP/IP stack, which makes them hard to incrementally deploy in production datacenters. CP [24] even requires changes to the switch hardware. Moreover, their modifications may trigger other problems for TCP. For example, reducing RTO_{min} [29] may lead to more spurious timeouts and retransmission in wide area networks with larger RTT and introduce extra system overheads due to the usage of high resolution timers.

D. *Why propose ACK control solution?*

Our overarching goal is to design a *readily deployable incast control mechanism to effectively deal with a large number of concurrent connections (e.g., over a thousand) that appeared in real production datacenter communications*. This is a challenge.

On the one hand, as discussed above, the window-based approach has its intrinsic limitations because the window size cannot be decreased infinitely (*e.g.*, 1 MSS at minimum). Thus, even tens of synchronized senders may be able to disable any window-based design. On the other hand, the essence of a recovery-based approach is to enable fast packet retransmission upon any packet loss, which is beyond the capability of legacy TCP or commodity switches. Thus, designing a new recovery-based method will very likely end up with non-trivial system modifications which may hurt the incremental deployability.

Therefore, we have to bypass the two existing directions by seeking a new solution space. This motivates us to explore the

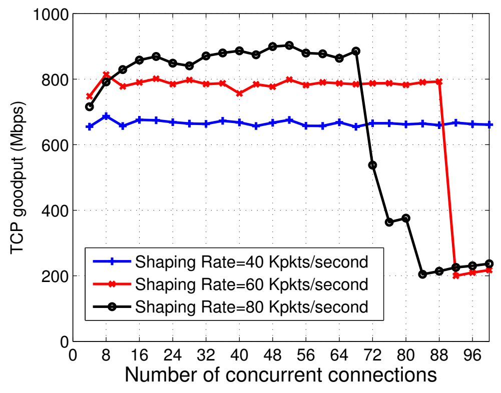


Fig. 3. TCP goodput with different shaping rate of ACK packets. Each connection generates 64KB data.

new avenue of proactive ACK control. The key factor inspiring the newly proposed ACK control approach is the viewpoint that we can treat ACK not only as the acknowledgement of received packets but also as the trigger for new packets. This is because, as a self-clocking protocol [18], TCP relies on the arrival of ACK packets to infer that the network can accept more packets, thus maintaining continuous and stable transmission to fully utilize link capacity.

As a matter of fact, the ACK control based solution naturally meets our requirements. As the trigger of new TCP traffic, we can leverage proactive ACK control to adjust the in-flight traffic without suffering from the limitation of minimal window size shared by the window-based solutions. In the meantime, from the implementation perspective, in order to enforce ACK control, we just need to properly regulate the sending rhythm of ACKs, requiring no modification to existing TCP/IP stack or switches. Thus, it is compatible to all TCP implementations and is readily deployable.

Given such benefits, the next question is how to properly hold and send back ACKs so that the ACK-triggered in-flight traffic can fill up the pipeline without overfilling the shallow switch buffer. The problem is that if we impose excessive delay to ACKs, we can eliminate incast congestion collapse, but the link utilization may be reduced. If we impose insufficient delay to ACKs, the switch buffer may still be overfilled by the ACK-triggered data, causing incast congestion collapse.

To demonstrate this conflict, we implemented a ACK shaper with different rates and show the behaviors in Figure 3. A low shaping rate (*e.g.*, 40K packets per-second) can mitigate incast while achieving low link utilization (no more than 70%). A high shaping rate (*e.g.*, 80K packets per-second) still suffers from incast congestion. A moderate shaping rate (*e.g.*, 60K packets per-second) cannot eliminate incast congestion even at the expense of wasting some bandwidth (nearly 200Mbps).

In the next section, we introduce how we solve this conflict by proposing PAC.

III. Design

In this section, we introduce our design for solving the TCP incast congestion problem. First, in Section III-A, we introduce

Algorithm 1: The Main PAC Algorithm

```
   Initialization:
1  $threshold$ = buffer_size;
2  $in\_flight$ = 0;
3  ACKqueues = empty;
   Main loop:
4  while true do
5      if packets received and ACK $p$ generated then
6          update in_flight;
7          update threshold;
8          ACKqueues.enqueue($p$);
9      if !is_empty(ACKqueues) then
10         $q$ = ACKqueues.dequeue();
11         while $in\_flight + q.trigger > threshold$ do
12             keep tracking received packets and updating
               $in\_flight$ and ACKqueues;
13         send back ACK $q$;
14         update in_flight;
```

our PAC main algorithm to control in-flight traffic via properly releasing ACK packets. In Section III-B, we analyze how to set threshold for the in-flight traffic in order to achieve high link utilization while eliminating incast collapse. Then, we discuss how to estimate the in-flight traffic in networks in Section III-C. Finally, we introduce how to schedule ACK packets among multiple flows in Section III-D.

A. The Main PAC Algorithm

Following the discussion in Section II-D, we design Algorithm 1 as the main procedure for our PAC algorithm. The key idea is straightforward—it proactively controls ACK sending-back rhythm to make sure the *in-flight* traffic does not exceed the *threshold*, thus avoiding incast congestion collapse while still maintaining high link utilization.

In the initialization (lines 1–3), we set the *threshold* to be available switch buffer size, *in-flight* traffic volume to be 0, and *ACKqueues* to be empty, as there is no traffic in the network and no ACK at the receiver at the very beginning.

In the main loop, once PAC receiver receives new packets from the network, it will update the in-flight traffic and the threshold, then put the newly generated ACKs into the ACKqueues (lines 5–8). When there is an outstanding ACK in the ACKqueues to be sent back, we will first check whether releasing this ACK will cause the in-flight traffic to exceed the threshold. If so, PAC will hold it until there is enough network space to absorb the potential traffic triggered by this ACK before sending it back safely (lines 9–14).

We find that our PAC algorithm, though relatively simple, achieves very good performance in both simulations and real implementation. For example, PAC can sustain over 1000 connections in our ns-2 simulations and achieve nearly 900Mbps goodput in our testbed experiments. We believe there are intrinsic reasons behind, and in what follows we delve into these reasons by answering the following three questions.

1) How to set a proper *threshold*?
2) How to estimate the *in-flight* traffic?
3) How to schedule ACK packets at the receiver side?

B. *How to set a proper threshold?*

We rely on the *threshold* to modulate in-flight traffic. A large threshold can lead to congestion, while a small one would degrade link utilization.

To decide the threshold, we made the following key observations over production datacenter network characteristics.

- First, in the absence of queueing delay, normally the RTT in production datacenter networks is 100us for intra-rack and 250us for inter-rack [4].
- Second, to understand the RTT with queueing delay, we made an analysis over a production datacenter with over 40,000 servers and found that 90% percentile of the RTT is less than 400us.
- Third, it has been an emerging trend that production datacenter networks, *e.g.*, Windows Azure storage, offer uniform high capacity between racks, and the over-subscription ratio, typically less than 3:1, exists only at the ToR switches [20]. Then, with ECMP [16] and packet-level load balancing mechanisms [7, 12], it turns out that the congestion usually occurs at the edge and the core is free of persistent congestion [6, 20].
- Fourth, today's commodity switches can have around 100KB buffer per port, for example, the Broadcom Pronto-3295 switch in our testbed has about 4MB buffer memory shared among 48 ports.

Based on the above observations, we set the *threshold* to be the switch buffer size. The reason is three-fold.

- First, because the base intra-rack RTT can be near 100us in a Gigabit network, the base BDP (Bandwidth Delay Product) without queue is less than $100us \times 1Gbps = 12.5KB$, which is a very small pipe with limited capacity to absorb in-flight traffic. Therefore, it is more likely that the majority of the in-flight traffic can stay in the switch buffer. Thus, setting the threshold as the switch buffer size is a safe and conservative way in order to prevent incast congestion collapse.
- Second, because the network core is less congested and inter-rack RTT is around 250us, the base BDP is around $250us \times 1Gbps = 31.25KB$ which is smaller than a typical buffer size (e.g., 100KB). Therefore, by setting the threshold as the switch buffer size, it is likely that we can sustain high bottleneck link utilization.
- Third, even with some queueing delay due to congestion, most of the RTT (90%) is still less than 400us and the BDP is around $400us \times 1Gbps = 50KB$, which is almost a half of the buffer size, indicating that it is still possible to fill up the bottleneck link with such a threshold.

Our evaluation results in Section V confirm that the above threshold setting works well in practice. Based on this we can achieve high throughput without causing incast collapse even with some congestion at the network core. However, we anticipate that, in case the network core experiences a high degree of persistent congestion (which is rare), such a threshold might be aggressive, possibly leading to incast collapse which can degrade the performance. To address this, we leverage ECN to estimate the congestion situation and let the threshold react to such a condition. Specifically, we compute the fraction of ECN marked incoming packets as α, while α indicates the level of congestion. Inspired by [4], we adjust the threshold as follows:

$$threshold \leftarrow \begin{cases} threshold \times (1 - \alpha/2) & \alpha > 0 \\ min\{threshold \times 2,\ buffer\_size\} & \alpha = 0 \end{cases} \tag{2}$$

When $\alpha > 0$, we reduce the threshold in proportion to the level of congestion. When $\alpha = 0$, it means that the network is free of congestion and we increase the threshold exponentially to the buffer size value in order to quickly fill up the pipeline. The experiment results show that this scheme can effectively deal with the incast happening at the network core.

C. *How to estimate in-flight traffic?*

To estimate the in-flight traffic, we consider three aspects: 1. when releasing an ACK packet, we should increase the in-flight traffic; 2. when receiving an incoming packet, we should reduce the in-flight traffic; 3. We must be able to detect the possible mis-estimation occurring in the first two steps and correct the deviation.

Update *in-flight* when releasing an ACK packet: When PAC sends an ACK packet back to the network, new TCP traffic is supposed to be generated, thus in-flight traffic should be increased correspondingly (lines 13–14). However, the problem is how to estimate the volume of in-flight traffic triggered by an ACK packet? Assuming the ACK packet PAC wants to release is q, the latest released ACK packet of the same flow is *prev*, and their acknowledgement numbers are ACK_q and ACK_{prev} respectively. Based on the window sliding and cumulative acknowledgement mechanism of TCP, taking TCP Reno [18] as an example, the traffic (*i.e.*, application payload) triggered by q is given by:

$$\begin{aligned} q.trigger &= ACK_q - ACK_{prev} + Increment \\ &= \begin{cases} ACK_q - ACK_{prev} + MSS & SS \\ ACK_q - ACK_{prev} + \dfrac{MSS \times MSS}{Window} & CA \end{cases} \end{aligned} \tag{3}$$

where *Increment* is the variation of the TCP congestion window, *SS* stands for slow start and *CA* congestion avoidance.

However, we note that, when ECN is enabled, *Increment* can be negative for an ACK packet marked with the ECE-Echo flag. In this case, we just conservatively set $q.trigger = ACK_q - ACK_{prev}$. We later show that PAC can correct the derivation of estimation noise. Based on the above derivation, when we send back q, we should update the volume of in-flight traffic (line 14) as follows:

$$in\_flight = in\_flight + q.trigger \tag{4}$$

Update *in-flight* when receiving an incoming packet: This case is easy. When the receiver receives an incoming packet with length L, we simply reduce the volume of in-flight traffic correspondingly:

$$in\_flight = in\_flight - L \tag{5}$$

Deviation Correction: The above estimation of the in-flight traffic might not always be accurate and can introduce deviations. Even though datacenters are under a single administrate control and we know the TCP congestion control algorithm (*e.g.*, Reno [18] or CUBIC [14]) that end hosts use, the *Increment* in equation (3) is still difficult to yield. This is because: 1) we have little or no knowledge of the TCP phase, slow start or congestion avoidance, of this flow; 2) it is possible that the last few ACKs may not trigger any data due to the end of the TCP connection; 3) The ECN or flow control may reduce the sending window. All above these may lead to over-estimation of the in-flight traffic, thus decreasing the throughput.

To address to problem, we leverage per-flow information to infer the TCP phase of a flow. For flow i, its per-flow control interval is RTT_i. We measure the incoming throughput of the flow i in the N-th interval as BW_N^i. If we observe that $BW_N^i \leq BW_{N-1}^i \times \lambda$ holds for two consecutive intervals, where λ is 0.8 in our implementation, we infer that the congestion window increasing pace slows down and flow i has transformed from slow start into congestion avoidance. In addition, if any incoming packet of flow i is marked with ECN, it also indicates that flow i has come into the congestion avoidance phase [26].

The above per-flow stage inference cannot directly correct the deviation of the previous over-estimation of the in-flight traffic. To this end, we treat the reduction of aggregate incoming throughput as the signal to adjust the in-flight traffic. When a receiver senses a suspicious throughput reduction (without ECN), it decreases the in-flight value. Specifically, we use the average RTT of all TCP connections as the control interval. In one control interval, we measure its actual incoming throughput as BW_T and maintain a smooth bandwidth BW_S and in-flight traffic in_flight_S using the exponential filter. When we observe that $BW_T \leq BW_S \times \lambda$, it indicates a suspicious link utilization reduction, owing to the reasons discussed above. Therefore, we adjust the in-flight traffic to correct the deviation as follows:

$$in\_flight = min\{in\_flight,\ \frac{in\_flight_S \times BW_T}{BW_S \times (1 - \alpha/2)}\} \tag{6}$$

where α is the fraction of the ECN marked incoming packets. When α is small, $\frac{in\_flight_S \times BW_T}{BW_S \times (1-\alpha/2)} \approx \frac{in\_flight_S \times BW_T}{BW_S}$ and we adjust in-flight traffic mainly based on the reduction of incoming throughput. When α is large, it indicates a high level of congestion and $\frac{BW_T}{BW_S \times (1-\alpha/2)}$ can be even larger than 1. In such a case, we cannot conclude whether the throughput reduction is due to an over-estimation or core congestion, thus we keep the current in-flight value. Experiments show that the above heuristic is effective in correcting the deviation and maintains a high throughput.

D. How to schedule ACK packets?

Throughput-sensitive large flows, latency-sensitive short flows and bursty incast flows coexist in current datacenters. The goal of ACK scheduling at the receiver side is to prioritize the short/bursty flows while not imposing excessive delay on any of them. As a practical incremental deployable transport-layer solution, PAC should not make any assumptions about the flow size information as opposed to prior works [6, 15, 28, 30], and should be self-adaptive without a priori knowledge of flow size.

To this end, we adopt the Multi-level Feedback Queue (MLFQ) [10, 27] to implement our ACK scheduling at the receiver side. To enforce MLFQ, PAC introduces N distinct queues to store ACK packets, which are assigned with N different priority levels. The ACK packets in different priority queues are scheduled strictly based on their priorities strictly (from high to low). For the ACK packets in the same priority queue, PAC performs per-flow Round Robin (RR) to schedule them. The ACK packets of a newly generated flow is initially put in the highest priority queue. Then, with the increase in flow size, its priority is reduced gradually and its ACK packets will be moved from higher level queues to lower level queues step by step. In this way, short flows would likely to be completed in the first few high priority queues, while long flows would eventually sink to the lowest priority queues, giving preference to the short ones. Therefore, the latency-sensitive short flows will receive prioritized services and experience relatively less delay.

Another concern of PAC is its interaction with TCP. In datacenters, RTO_{min} is usually set to be a relatively low value (*e.g.*, 10 microseconds [4]) compared to the Internet. As RTT, especially for those low-priority large flows, can be increased due to the PAC's ACK scheduling, it is not sure whether this will cause RTT to be prolonged beyond RTO_{min}, thus leading to TCP timeouts and spurious retransmissions. For this reason, we specifically measure the RTTs in our testbed experiments, and we find that our ACK control does not adversely prolong the RTTs (for example, with 100 connections, the 99^{th} percentile RTT is only 3.6ms) and we do not observe any spurious retransmission.

Even though this phenomenon is rare, we still take into account the possibility and design counter-measures. Suppose PAC observes the arrival of a stale ACK that are being queued in the ACK queue (which may be caused by TCP timeout and retransmission), it drops all the out-of-order ACK packets of this flow, moves the remaining in-order ACK packets to the highest priority queue, and marks this flow as slow start. The reason is that, by dropping the the out-of-order ACKs, PAC avoids disturbing the TCP at the sender. In addition, by increasing the priority, PAC reduces the delay and RTT so that the following unnecessary retransmissions can be mitigated.

IV. Experimental Setup

A. Implementation

As a prototype, we have implemented PAC both as a NDIS (Network Driver Interface Specification) filter driver for the

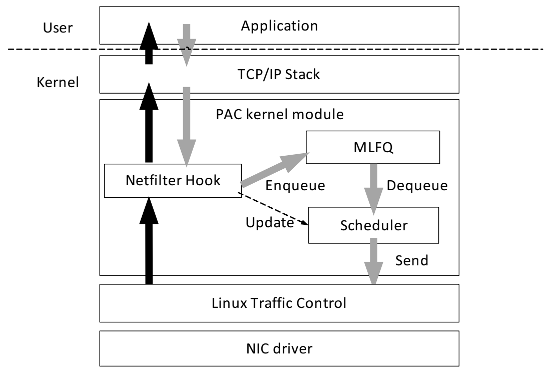


Fig. 4. Software stack of PAC implementation in Linux.

Windows platform and as a kernel module for the Linux platform. In a non-virtualized environment, PAC works as a shim layer above the NIC driver. This does not touch any TCP/IP implementation of OS, making it readily deployable in production datacenters. In a virtualized environment, PAC resides in hypervisor (or Dom 0), well supporting virtual machines in cloud datacenters [31]. In what follows, we introduce our Linux implementation which is used for all of the experiments in this paper.

As shown in Figure 4, PAC kernel module is located between the TCP/IP stack and the Linux Traffic Control (TC) module. PAC implementation consists of three key components: *Netfilter hook*, *Multi-level Feedback Queues (MLFQ)* and *Scheduler*. We insert Netfilter [2] hooks in *PRE_ROUTING* and *POST_ROUTING* to intercept all incoming and outgoing TCP packets. Outgoing TCP ACK packets are enqueued into MLFQ. Each TCP flow, identified by an 4-tuple: source/destination IP addresses and source/destination port numbers, maintains its own flow state. For a TCP flow, when its FIN/RST packets are observed or none of this flow are captured for five minutes, its flow state will be removed. The scheduler chooses ACK packets from MLFQ and sends them back to network based on the PAC algorithm.

The operations of the the PAC kernel module are as follows: 1) When an outgoing ACK packet is captured the by Netfilter hook, it will be directed to MLFQ. 2) The scheduler is responsible to dequeue an ACK packet from MLFQ and decide when to release it based on Algorithm 1. If the in-flight traffic does not exceed the threshold, this ACK packet will be sent immediately and the information in the scheduler will be updated (*e.g.*, in-flight traffic). Otherwise, the scheduler just holds this packet and keeps tracking incoming packets until it is safe to release it. 3) When incoming traffic is observed by the Netfilter hook, the information of the scheduler is also updated correspondingly as we discuss in Section III-C.

CPU Overhead: We measured the CPU overhead introduced by our PAC kernel module. We installed PAC kernel module on a Dell server PowerEdge R320 with a Intel E5-1410 2.8GHz CPU and 8GB memory. We enabled ECN in switches and used TCP New Reno with DCTCP as our congestion control algorithm. We started 10 concurrent long TCP connections and achieved more than 900Mbps throughput in total. The extra overhead introduced by PAC is around 1%-1.6% compared with the case PAC kernel module is not deployed.

Buffering Pressure: Since PAC holds ACKs at the receiver, we measured packet buffering overheads introduced by PAC. We constructed an incast congestion scenario with 100 connections where each connection generated 64KB data. We run this 10 times and obtained the maximum ACK packets queuing length for each time. The average value is 223. Considering the length of an ACK packet is just 66 bytes (Ethernet Header 14 + IP Header 20 + TCP Header 20 + TCP Options 12 = 66), the queueing space required by PAC is very small; only tens of KBs.

Get Fine-Grained RTT: As section III shows, PAC requires RTT for control interval calculation. At the sender side, live RTT can be easily obtained by the time elapsed between when a data packet was sent and the ACK for that data packet arrived. At the receiver side where PAC is deployed, if the traffic between the sender and receiver is bidirectional, we can also use the above solution to get RTT. Considering data traffic in the reverse direction may not be enough, we need generic and robust solutions to obtain RTT.

In our implementation, we use two methods to obtain reverse RTT at the receiver side: 1) Using the TCP timestamp option [19] 2) Using initial RTT measured in connection establishment (base RTT) plus extra delay added by PAC. The TCP timestamp is enabled by default in our Linux 2.6 kernel. However, the current timestamp value (TSval) in the TCP timestamp option is in millisecond granularity. Similar to ICTCP [31], we modify timestamp values to microsecond-granularity time. Note that this modification is only done by PAC at the receiver side and we do not touch the TCP stack on end host as [29]. Apart from a fine-grained TCP timestamp, we can estimate RTT as the sum of the initial RTT sampled in the connection establishment and extra delay added by PAC. Though not very accurate, this coarse-grained estimation also works well in practice. By default, we use the second coarse-grained method to minimize overhead and found it achieves good performance.

B. Testbed

Our testbed consists of 38 Dell servers and two Pronto 3295 48-port Gigabit Ethernet switches with 4MB shared memory. Our topologies are shown in Figure 5 and Figure 6. In Figure 5, all the servers are connected to a 48-port Gigabit Ethernet switch and incast congestion happens at the last hop. In Figure 6, incast congestion happens in the intermediate link (network core). Each server has a four-core Intel E5-1410 2.8GHz CPU, 8G memory, 500GB hard disk and one Broadcom BCM5719 NetXtreme Gigabit Ethernet NIC. The OS of each server is Debian 6.0 64-bit version with Linux 2.6.38.3 kernel. The servers have their own background TCP traffic for various services (*e.g.*, SSH) but the amount is small compared to our incast traffic. The base RTT in our testbed is around 100us. We allocate a static buffer size of 100 packets ($100 \times 1.5KB = 150KB$) to the port with congestion. In our experiments, each

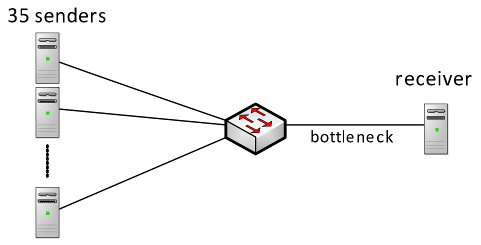


Fig. 5. The topology of incast at network edge, in our experiments each sender can setup multiple connections emulating more senders.

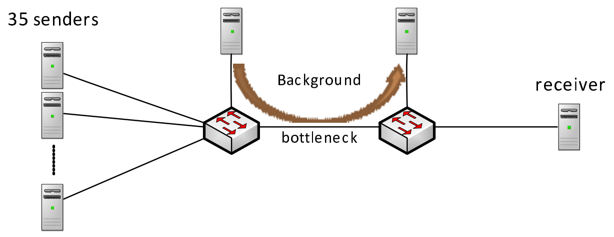


Fig. 6. The topology of incast in network core, in our experiments each sender can setup multiple connections emulating more senders.

sender can generate multiple connections (*e.g.*, 3) to emulate multiple senders to send data to the receiver.

We use TCP New Reno [18] as our congestion control algorithm and disable the delayed ACK. Advanced offload mechanisms (*e.g.*, Large Segmentation Offload) are enabled by default. For DCTCP implementation, we use public code from [1] and add ECN capability to SYN packets [21]. We implement ICTCP as a Linux kernel module following the description in [31]. The default RTO_{min} on Linux is 200 milliseconds.

V. Evaluation

The goals of the evaluation are to: 1) show that PAC can support many more connections than window-based solutions and maintain high link utilization, 2) show that PAC can mitigate incast congestion in the network core, 3) quantify PAC's impact on latency, and 4) explore PAC's performance in non-incast scenarios.

Summary of main results is as follows:

1) For typical edge incast, PAC can easily handle 100 connections while achieving around 900Mbps goodput in our small-scale testbed experiments, and furthermore, it supports over 1600 connections (40X that of ICTCP and DCTCP) while achieving over 800Mbps goodput in our large-scale ns-2 simulations.
2) In the presence of network core congestion, PAC can support over 100 concurrent connections while maintaining over 700Mbps goodput in our testbed experiments. However, DCTCP starts to degrade when the number of connections exceeds 24.
3) PAC imposes little influence on latency: in our experiments with 100 concurrent connections, the measured 50th percentile RTT is 1.3ms and the 99th percentile is 3.6ms, which will not incur spurious timeout even the $RTOmin$ is as 10ms in production datacenters.
4) For non-incast cases, PAC demonstrates good fairness on multiple connections and achieves high throughput for long-term TCP traffic.

Parameters: Table I gives the parameters used in the testbed experiments. We set the ECN marking threshold in the switch to be 10 packets (15KB) because the base RTT in our testbed is around 100us ($BaseBDP = 100us \times 1G = 12.5KB < 15KB$). While we allocated a 150KB fixed buffer to switch incast port, we intentionally set the initial threshold for the PAC algorithm to be 80KB which is the average buffer size for a port ($4MB/48$). This is because PAC should not make any assumptions about buffer management in switches and it should also be adaptive. We implement a two-level MLFQ and use 20KB as the single threshold.

PAC	ECN Marking Thresh.=10 packets Initial threshold=80 KB, λ=0.8 2 priorities, Flow Size Thresh.=20KB
DCTCP	ECN Marking Thresh.=10packets, g=0.0625
ICTCP	Minimal Window= 2MSS, γ_1=0.1, γ_2=0.5

TABLE I
Parameter Settings

Benchmark Traffic: We write a simple client-server application to construct incast congestion. We use two types of traffic benchmarks used by previous works [4, 9, 25, 29].

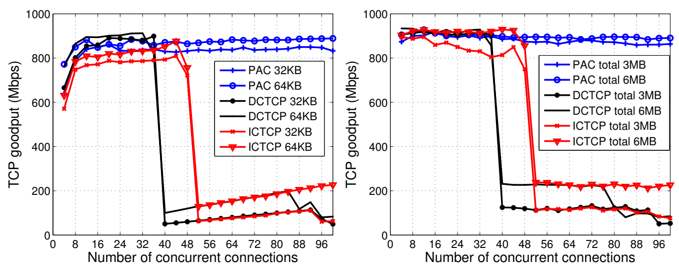
- *Fixed volume per sender*: In this scenario, we fix the traffic amount generated by each connection and the total amount of traffic is increased with the number of connections. [9, 25]
- *Fixed volume in total*: In this scenario, we fix the total traffic amount shared by all connections and the number of connections varies. The more connections, the lower volume per connection. [4, 29]

The TCP connections are barrier-synchronized per round and our result is the average value of 20 rounds.

A. Incast Congestion at Network Edge

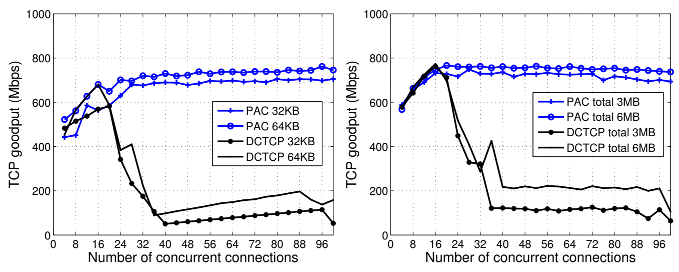
We use the topology in Figure 5 to reproduce incast congestion at network edges. Figure 7 shows the goodput of PAC, ICTCP, and DCTCP. In general, PAC can easily handle 100 concurrent connections without seeing any trend in performance degradation, while ICTCP and DCTCP begin to downgrade when the numbers of connections exceed 48 and 36 respectively.

When the number of connections is small, DCTCP shows little advantage than PAC in throughput (tens of Mbps). For example, PAC achieves 807 Mbps while DCTCP achieves 829 Mbps with 10 connections at 32KB per server. We attribute this to PAC's conservative parameter settings and estimation deviation. However, PAC can continue to achieve near 900Mbps goodput with an increasing number of connections, while the other two suffer incast collapse. We also observe that ICTCP can support more connections than DCTCP. That is because, 1) DCTCP relies on ECN to detect congestion, thus requiring a larger buffer to avoid congestion during control



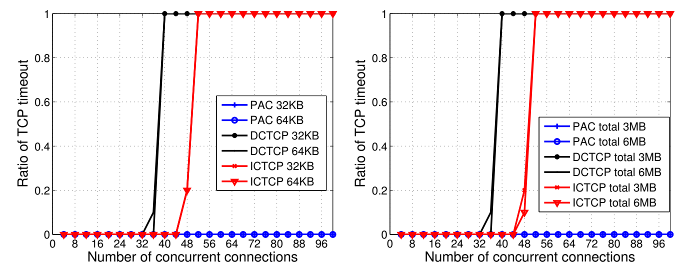
(a) Fixed volume per sender (b) Fixed volume in total

Fig. 7. The goodput of PAC, DCTCP, and ICTCP in network edge incast congestion.



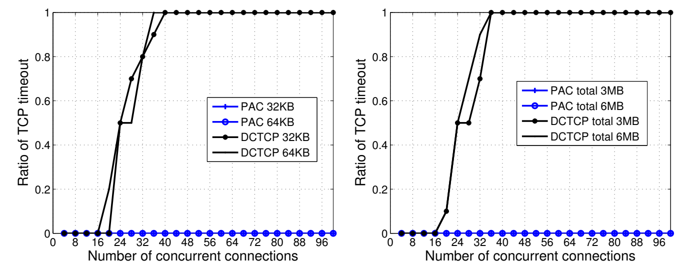
(a) Fixed volume per sender (b) Fixed volume in total

Fig. 9. The goodput of PAC and DCTCP in network core incast congestion.



(a) Fixed volume per sender (b) Fixed volume in total

Fig. 8. The timeout ratio in network edge incast congestion.



(a) Fixed volume per sender (b) Fixed volume in total

Fig. 10. The timeout ratio in network core incast congestion.

latency; 2) The default initial congestion window in Linux is 3MSS, larger than the minimal window of ICTCP which is 2MSS.

To evaluate the effectiveness of PAC, we also measure the TCP timeout events in our experiments. We compute the timeout ratio as among the total 20 rounds for each experiment, how many rounds we see at least one TCP timeout event. The results are shown in Figure 8. We observe that the timeout ratio of PAC remains at 0 with the increasing number of connections.

We further explore the reasons behind our results. As we have analyzed in section II-B, the maximum number of connections supported by DCTCP and ICTCP is given by:

$$N_{max} = (B + C \times RTT)/Window_{min} \tag{7}$$

In our testbed, the switch buffer size B is around 150KB and $C \times RTT$ is smaller than 12.5KB ($100us \times 1Gbps = 12.5KB$). The $Window_{min}$ for ICTCP and DCTCP are 3KB ($2 \times 1.5KB$) and 4.5KB ($3 \times 1.5KB$) respectively. In theory, ICTCP and DCTCP can handle around 54 and 36 connections at most in our experiment setup. Our experiments confirm these theoretical values, and the little deviation is mainly due to the servers' own background traffic. For PAC, since we increase RTT by delaying ACK packets at the receiver side, we can *decouple the maximum number of connections from switch buffer size*, thus supporting many more connections.

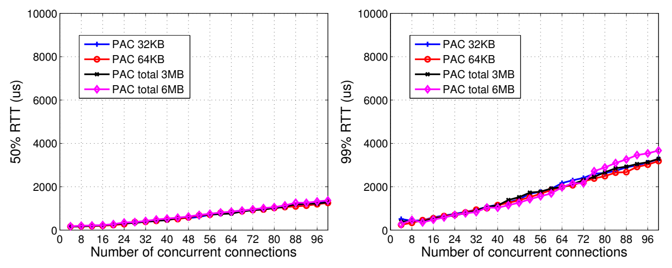
B. Incast Congestion in Network Core

We use the topology in Figure 6 to reproduce incast congestion in the network core. We generate a long-term TCP connection as background traffic which occupies 750Mbps bandwidth before the incast traffic starts. In this scenario, because the intermediate link is the bottleneck and ICTCP does not work (it was designed for edge incast [31]), we only compare PAC with DCTCP. The goodput of PAC and DCTCP is shown in Figure 9. Due to the influence of the background traffic, the goodput of both PAC and DCTCP is decreased compared with edge congestion; almost 500-700Mbps when the number of connections is within 20. Unlike DCTCP, PAC achieves better goodput as the number of connections increases, finally reaching over 700Mbps goodput with 100 connections.

Figure 10 shows the TCP timeout ratio in network core incast congestion. Even under the pressure of background traffic, PAC still maintains zero timeout. We attribute this to PAC's robust adjustment mechanism, which adapts to various situations in the network core. In contrast, DCTCP suffers from more severe timeout in core congestion than that at the edge congestion. For example, DCTCP begins to experience timeout when the number of connections reaches 20 in core congestion and 36 at edge congestion. This is because incast flows see a smaller available buffer size at the core due to the pressure of background traffic. As we have analyzed before, the window-based solutions are sensitive to available switch buffer size. Smaller available buffer size leads to worse performance.

C. Impact on Latency

PAC avoids incast congestion by proactively intercepting and controlling ACKs at the receiver, which might increase the



(a) The 50th percentile (b) The 99th percentile

Fig. 11. The 50th percentile and 99th percentile RTTs of PAC in our testbed experiments.

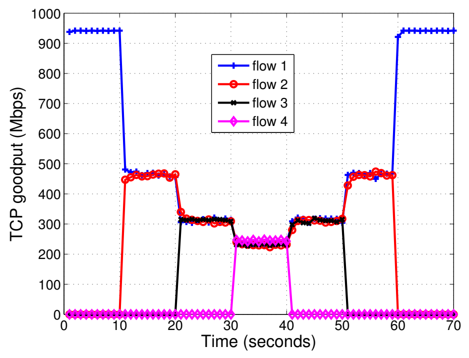


Fig. 12. The goodput of 4 PAC flows from 4 servers to the same receiver under the same switch, these 4 flows are gradually added and then removed every 10 seconds.

RTT of TCP connection. To measure how much influence PAC can impose on the RTT, we used the microsecond-granularity TCP timestamp to measure RTT at the receiver side. Figure 11 shows the results of the 50th percentile RTT and the 99th percentile RTT respectively.

We find that PAC delivers little impact on RTT. For example, when there are 100 concurrent connections, half of the RTTs are less than 1.3ms, while 99% of them are less than 3.6ms. Currently, many production datacenters have reduced RTO_{min} to a low value (*e.g.*, 10ms [4]). In Linux, the lowest possible RTO value is 5 jiffies (5ms) [29]. This suggests that PAC can work smoothly and will not result in issues like spurious timeouts and retransmissions in production datacenters with low RTO_{min}.

To analyze the PAC's influence on RTT, we assume there are N synchronized TCP incast connections with identical window W and RTT, sharing a bottleneck link with capacity C. To avoid incast congestion, the aggregate incoming throughput should be no larger than link capacity:

$$C \geq \frac{N \times W}{RTT} \tag{8}$$

Therefore, to avoid incast collapse, RTT should satisfy the following requirement:

$$RTT \geq \frac{W \times N}{C} \tag{9}$$

As we see from above formula, RTT is closely related to both the window and the number of connections. Considering ECN is enabled, W should be close to minimal congestion window which is 3 MSS in Linux. With 100 connections, RTT should be enlarged to $3 \times 1.5KB \times 100/1Gbps \approx 3.6ms$, matching our measurement results.

D. Long-Term Performance

To evaluate the performance of PAC in non-incast scenarios, we generate 4 TCP flows from 4 servers to the same receiver under the same switch. These 4 flows are gradually added at time 1, 10, 20 30, and then removed at time 40, 50, 60, 70. We depict the throughput dynamics in Figure 12. We find that with the addition and removal of flows, the active flows can always quickly and fairly share the network bandwidth, and their aggregate throughput is more than 900Mbps, close to link capacity. PAC achieves good fairness because we used the Round Robin to schedule ACK packets among the flows in the same priority level of our MLFQ, and it maintains high link utilization because we used an appropriate threshold to regulate the in-flight traffic.

E. Large-Scale Simulation

Due to the size and hardware limitations of our testbed, we cannot test PAC's performance in 10G network. Moreover, we also want to explore the maximum number of connections PAC can handle. Therefore, we complement our testbed experiments with large-scale ns-2 simulations. Our simulations have three parts: 1) we evaluate PAC in 1G network to confirm with our textbed experiments; 2) we evaluate PAC in 10G network; and 3) we explore the maximum number of concurrent connections PAC can support simultaneously.

Simulation topology: We simulate a topology similar to Figure 5. All the servers are connected under the same ToR switch. The only difference is that we have *infinite* senders and each sender has one connection with the receiver. To simulate the system overhead, we add random delay to the RTT.

Parameters	1G Network	10G Network
MSS	1460 Bytes	1460 Bytes
Switch	96KB	375KB
Base RTT	120us	80us
TCP	NewReno	NewReno
RTOmin	200ms	200ms
PAC	Initial threshold=96KB	Initial threshold=375KB
	K=10pkts	K=65pkts
	α=0.8	α=0.8
ICTCP	Minimal window=2MSS	Minimal window=2MSS
	γ_1=0.1	γ_1=0.1
	γ_2=0.15	γ_2=0.15
DCTCP	Initial window=2MSS	Initial window=2MSS
	K=10pkts	K=65pkts
	g=0.0625	g=0.0625

TABLE II
SIMULATION SETTINGS

Simulation settings: Our simulations are performed in both 1G network and 10G network. We summarize our parameters in Table II. For PAC, we set the threshold to be equal to the switch buffer size, 96KB and 375KB for 1G network and 10G network respectively. For DCTCP, we set the marking

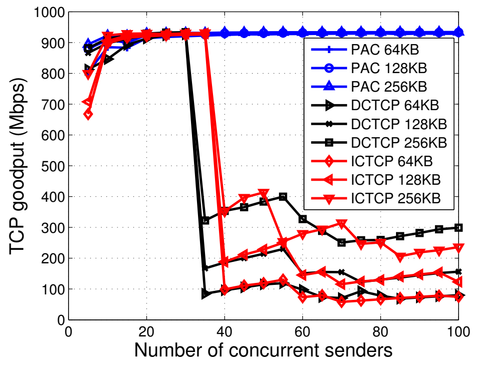


Fig. 13. The goodput of PAC, DCTCP and ICTCP in 1G network.

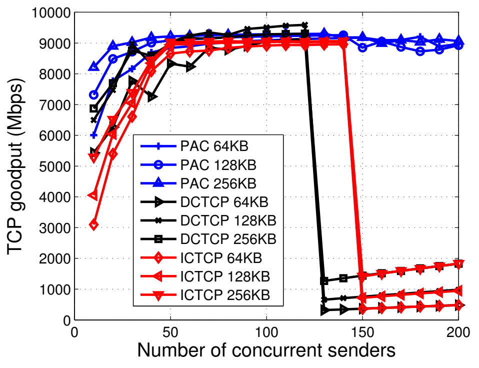


Fig. 14. The goodput of PAC, DCTCP and ICTCP in 10G network.

threshold of DCTCP roughly as the base BDP, 10 packets for 1G network and 65 packets for 10G network. For ICTCP, we determine the best settings for its parameters according to the simulations. We use NewReno as our TCP implementation and disable delayed ACK. The RTO_{min} is 200ms.

Workload: We also use two types of workloads as in testbed: fixed volume per server and fixed volume in total. We use the first type when we compare the performance of PAC with DCTCP and ICTCP. We use the second one when we explore the maximum number of concurrent connections PAC can handle.

1) 1G Network: The goodput of PAC, DCTCP and ICTCP in 1G network is shown in Figure 13. From the figure, we find that by setting a proper threshold to modulate the ACK sending-back rate, PAC avoids incast congestion collapse while still achieving high link utilization. On the contrary, ICTCP and DCTCP achieve high throughput at the beginning, but eventually downgrade when the number of senders becomes larger, around 40 in our results.

We also find that the number of connections that DCTCP and ICTCP can handle is smaller that those of our testbed experiments. This is because we have reduced switch buffer size from 150KB to 96KB. This verifies that the window-based solutions are sensitive to switch buffer management as we analyzed in Section II-B.

2) 10G Network: Figure 14 shows the goodput of PAC, DCTCP and ICTCP in 10G network. Given that both the base BDP and switch buffer size become larger in 10G network and the minimal window remains the same, DCTCP and ICTCP can generally handle more concurrent connections than they do in 1G network, but finally experience packet losses and thus incast collapse when the number of senders exceeds 130 and 150, respectively. In contrast, PAC can still avoid incast collapse and maintain high throughput with an increasing number of connections.

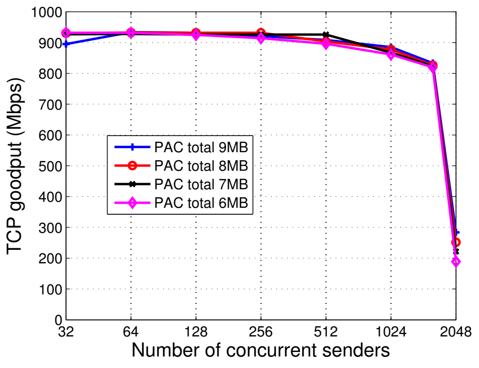


Fig. 15. The maximum number of connections PAC can support.

3) The Maximum Number of Connections: To explore the maximum number of connections that PAC can handle, we fix the total traffic volume and gradually increase the number of senders. We use 1G network environment for this experiment. The result is shown in Figure 15.

From the figure, we find that PAC can easily support more than 1000 concurrent connections and sustain over 800Mbps throughput when facing 1600 senders. It begins to experience packet losses and TCP timeouts when the number of connections reaches 2048.

When we update the in-flight traffic, we do not take into account the control messages of connection establishment. Compared with the size of MSS, the size of the SYN packets, which is just tens of bytes, can be ignored. However, when the number of senders is significantly large, these control signals can be a threat. Suppose there are 2000 senders and each sends a SYN packet of 66 bytes, the total amount of traffic is more than 132KB. When these SYN packets come synchronously, they will overfill the switch buffer and PAC fails to work.

VI. Related Works

The most related works to PAC include DCTCP [4], ICTCP [31], Tuing ECN [32], GIP [34], reducing-RTO_{min} [29] and CP [24]. As we have discussed previously, window-based solutions [4, 31, 32] only support quite a limited number of senders while recovery-based solutions [24, 29, 34] are not incrementally deployable. We have done extensive experiments and simulations to compare with them. Apart from the above works, there have also recently been some other datacenter transport designs, though their main design goals are not mitigating incast congestion.

HULL [5] tries to keep the switch buffer empty to achieve ultra-low latency. HULL uses phantom queue to simulate a network at less than 100% utilization and relies on ECN to deliver congestion information. Moreover, HULL deploys hardware packet pacer on end hosts to throttle bursty traffic.

D2TCP [28] adds deadline-awareness on the top of DCTCP. It adjusts the congestion window based on both the congestion situation and deadline information to meet deadlines.

MCP [8] establishes a theoretical foundation on how to achieve a minimal-delay deadline-guaranteed datacenter transport layer protocol, and then it leverages ECN and commodity switches to implement and approximate this optimal solution.

D3 [30] achieves explicit rate control based on deadline information to guarantee deadlines. It maintains the state for each flow in switches, which requires hardware changes. Moreover, its first-come-first-served nature may lead to sub-optimal scheduling.

PDQ [15] uses preemptive flow scheduling to approximate ideal SRPT (shortest remaining processing time) scheduling, thus minimizing average flow completion time. Like D3, its flow scheduling is also based on explicit rate control assigned by switches. Therefore, it requires non-trivial switch and end host modifications, making it hard to implement in practice.

Similar to PDQ [15], pFabric [6] also tries to approximate optimal flow scheduling. Instead of using explicit rate control as [15, 30], it decouples flow scheduling from rate control and achieves near-optimal flow completion times.

VII. Conclusion

In this paper, we have designed, implemented, and evaluated PAC, a simple yet very effective design to tame TCP incast congestion via Proactive ACK Control. The key design principle behind PAC is that we treat ACK not only as the acknowledgement of received packets but also as the trigger for new packets. Leveraging practical datacenter network characteristics, PAC enforces a novel ACK control mechanism to release ACKs in such a way that the ACK-triggered in-flight traffic can fully utilize the bottleneck link without running into incast congestion collapse even when faced with over a thousand senders. Our extensive experiments and simulations show that PAC significantly outperforms the existing window-based solutions. Our implementation experiences show that PAC requires minimal system modification, making it readily deployable in production datacenters.

Acknowledgements This work is supported by HKRGC under ECS 26200014, and National Basic Research Program of China (973) under Grant No.2014CB340303 and 2013CB329103, and the Fundamental Research Funds for the Central Universities.

References

[1] "DCTCP Patch," http://simula.stanford.edu/~alizade/Site/DCTCP.html.

[2] "Linux netfilter," http://www.netfilter.org.

[3] "The Network Simulator NS-2," http://www.isi.edu/nsnam/ns/.

[4] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center TCP (DCTCP)," in *SIGCOMM 2010*.

[5] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda, "Less is more: trading a little bandwidth for ultra-low latency in the data center," in *NSDI 2012*.

[6] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, "pfabric: Minimal near-optimal datacenter transport," in *SIGCOMM 2013*.

[7] J. Cao, R. Xia, P. Yang, C. Guo, G. Lu, L. Yuan, Y. Zheng, H. Wu, Y. Xiong, and D. Maltz, "Per-packet load-balanced, low-latency routing for clos-based data center networks," in *CoNEXT 2013*.

[8] L. Chen, S. Hu, K. Chen, H. Wu, and D. Tsang, "Towards Minimal-Delay Deadline-Driven Data Center TCP," in *HotNets 2013*.

[9] Y. Chen, R. Griffith, J. Liu, R. H. Katz, and A. D. Joseph, "Understanding TCP incast throughput collapse in datacenter networks," in *WREN 2009*.

[10] F. J. Corbató, M. Merwin-Daggett, and R. C. Daley, "An experimental time-sharing system," in *Proceedings of the May 1-3, 1962, spring joint computer conference*, pp. 335–344.

[11] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, pp. 107–113, 2008.

[12] A. Dixit, P. Prakash, Y. C. Hu, and R. R. Kompella, "On the impact of packet spraying in data center networks," in *INFOCOM 2013*.

[13] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "VL2: a scalable and flexible data center network," in *SIGCOMM 2009*.

[14] S. Ha, I. Rhee, and L. Xu, "CUBIC: a new TCP-friendly high-speed TCP variant," *ACM SIGOPS Operating Systems Review*, 2008.

[15] C.-Y. Hong, M. Caesar, and P. Godfrey, "Finishing flows quickly with preemptive scheduling," in *SIGCOMM 2012*.

[16] C. Hopps, "RFC 2992: Analysis of an Equal-Cost Multi-Path Algorithm," 2000.

[17] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," *ACM SIGOPS Operating Systems Review*, pp. 59–72, 2007.

[18] V. Jacobson, "Congestion avoidance and control," in *SIGCOMM 1988*.

[19] V. Jacobson, R. Braden, and D. Borman, "RFC 1323: TCP extensions for high performance," 1992.

[20] V. Jeyakumar, M. Alizadeh, D. Mazieres, B. Prabhakar, C. Kim, and A. Greenberg, "EyeQ: practical network performance isolation at the edge," in *NSDI 2013*.

[21] A. Kuzmanovic, A. Mondal, S. Floyd, and K. Ramakrishnan, "Adding Explicit Congestion Notification (ECN) Capability to TCP's SYN/ACK Packets," *draft-ietf-tcpm-ecnsyn-03 (work in progress)*, 2007.

[22] D. Nagle, D. Serenyi, and A. Matthews, "The Panasas ActiveScale Storage Cluster: Delivering Scalable High Bandwidth Storage," in *Supercomputing 2004*.

[23] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, "Scaling Memcache at Facebook," in *NSDI 2013*.

[24] R. S. C. L. Peng Cheng, Fengyuan Ren, "Catch the Whole Lot in an Action: Rapid Precise Packet Loss Notification in Data Centers," in *NSDI 2014*.

[25] 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 2008*.

[26] K. Ramakrishnan, S. Floyd, D. Black *et al.*, "RFC 3168: The addition of explicit congestion notification (ECN) to IP," 2001.

[27] A. Silberschatz, P. B. Galvin, G. Gagne, and A. Silberschatz, *Operating system concepts*, 1998.

[28] B. Vamanan, J. Hasan, and T. Vijaykumar, "Deadline-aware datacenter tcp (d2tcp)," in *SIGCOMM 2012*.

[29] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. G. Andersen, G. R. Ganger, G. A. Gibson, and B. Mueller, "Safe and effective fine-grained TCP retransmissions for datacenter communication," in *SIGCOMM 2009*.

[30] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron, "Better never than late: Meeting deadlines in datacenter networks," in *SIGCOMM 2011*.

[31] H. Wu, Z. Feng, C. Guo, and Y. Zhang, "ICTCP: Incast Congestion Control for TCP in data center networks," in *CoNEXT 2010*.

[32] H. Wu, J. Ju, G. Lu, C. Guo, Y. Xiong, and Y. Zhang, "Tuning ECN for data center networks," in *CoNEXT 2012*.

[33] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing," in *NSDI 2012*.

[34] J. Zhang, F. Ren, L. Tang, and C. Lin, "Taming TCP Incast Throughput Collapse in Data Center Networks," in *ICNP 2013*.