HPCC: High Precision Congestion Control

Yuliang Li[♠♡], Rui Miao[♠], Hongqiang Harry Liu[♠], Yan Zhuang[♠], Fei Feng[♠], Lingbo Tang[♠], Zheng Cao[♠], Ming Zhang[♠], Frank Kelly[♢], Mohammad Alizadeh[♣], Minlan Yu[♡]

Alibaba Group[♠], Harvard University[♡], University of Cambridge[♢], Massachusetts Institute of Technology[♣]

ABSTRACT

Congestion control (CC) is the key to achieving ultra-low latency, high bandwidth and network stability in high-speed networks. From years of experience operating large-scale and high-speed RDMA networks, we find the existing high-speed CC schemes have inherent limitations for reaching these goals. In this paper, we present HPCC (High Precision Congestion Control), a new high-speed CC mechanism which achieves the three goals simultaneously. HPCC leverages in-network telemetry (INT) to obtain precise link load information and controls traffic precisely. By addressing challenges such as delayed INT information during congestion and overreaction to INT information, HPCC can quickly converge to utilize free bandwidth while avoiding congestion, and can maintain near-zero in-network queues for ultra-low latency. HPCC is also fair and easy to deploy in hardware. We implement HPCC with commodity programmable NICs and switches. In our evaluation, compared to DCQCN and TIMELY, HPCC shortens flow completion times by up to 95%, causing little congestion even under large-scale incasts.

CCS CONCEPTS

• **Networks → Transport protocols**; **Data center networks**;

KEYWORDS

Congestion Control; RDMA; Programmable Switch; Smart NIC

ACM Reference Format:

Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, Minlan Yu. 2019. HPCC: High Precision Congestion Control. In *SIGCOMM '19: 2019 Conference of the ACM Special Interest Group on Data Communication, August 19–23, 2019, Beijing, China.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3341302.3342085

1 INTRODUCTION

The link speed in data center networks has grown from 1Gbps to 100Gbps in the past decade, and this growth is continuing. Ultra-low latency and high bandwidth, which are demanded by more and more applications, are two critical requirements in today's and future high-speed networks.

Specifically, as one of the largest cloud providers in the world, we observe two critical trends in our data centers that drive the demand on high-speed networks. The first trend is new data center architectures like resource disaggregation and heterogeneous computing. In resource disaggregation, CPUs need high-speed networking with remote resources like GPU, memory and disk. According to a recent study [17], resource disaggregation requires 3-5μs network latency and 40-100Gbps network bandwidth to maintain good application-level performance. In heterogeneous computing environments, different computing chips, *e.g.* CPU, FPGA, and GPU, also need high-speed interconnections, and the lower the latency, the better. The second trend is new applications like storage on high I/O speed media, *e.g.* NVMe (non-volatile memory express) and large-scale machine learning training on high computation speed devices, *e.g.* GPU and ASIC. These applications periodically transfer large volume data, and their performance bottleneck is usually in the network since their storage and computation speeds are very fast.

Given that traditional software-based network stacks in hosts can no longer sustain the critical latency and bandwidth requirements [43], offloading network stacks into hardware is an inevitable direction in high-speed networks. In recent years, we deployed large-scale networks with RDMA (remote direct memory access) over Converged Ethernet Version 2 (RoCEv2) in our data centers as our current hardware-offloading solution.

Unfortunately, after operating large-scale RoCEv2 networks for years, we find that RDMA networks face fundamental challenges to reconcile low latency, high bandwidth utilization, and high stability. This is because high speed implies that flows start at line rate and aggressively grab available network capacity, which can easily cause severe congestion in large-scale networks. In addition, high throughput usually results in deep packet queueing, which undermines the performance of latency-sensitive flows and the ability of the network to handle unexpected congestion. We highlight two representative cases among the many we encountered in practice to demonstrate the difficulty:

Case-1: PFC (priority flow control) storms. A cloud storage (test) cluster with RDMA once encountered a network-wide, large-amplitude traffic drop due to a long-lasting PFC storm. This was triggered by a large incast event together with a vendor bug which caused the switch to keep sending PFC pause frames indefinitely. Because incast events and congestion are the norm in this type of cluster, and we are not sure whether there will be other vendor bugs that create PFC storms, we decided to try our best to prevent any PFC pauses. Therefore, we tuned the CC algorithm to reduce rates quickly and increase rates conservatively to avoid triggering PFC pauses. We did get fewer PFC pauses (lower risk), but the average link utilization in the network was very low (higher cost).

Case-2: Surprisingly long latency. A machine learning (ML) application complained about > 100us average latency for short messages; its expectation was a tail latency of < 50μs with RDMA.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SIGCOMM '19, August 19–23, 2019, Beijing, China

© 2019 Association for Computing Machinery.

ACM ISBN 978-1-4503-5956-6/19/08…$15.00
https://doi.org/10.1145/3341302.3342085

The reason for the long latency, which we finally dug out, was the in-network queues occupied majorly by a cloud storage system that is bandwidth intensive in the same cluster. As a result, we have to separate the two applications by deploying the ML application to a new cluster. The new cluster had low utilization (higher cost) given that the ML application is not very bandwidth hungry.

To address the difficulty to reconcile latency, bandwidth/utilization, and stability, we believe a good design of CC is the key. This is because CC is the primary mechanism to avoid packet buffering or loss under high traffic loads. If CC fails frequently, backup methods like PFC or packet retransmissions can either introduce stability concerns or suffer a large performance penalty. Unfortunately, we found state-of-art CC mechanisms in RDMA networks, such as DCQCN [43] and TIMELY [31], have some essential limitations:

Slow convergence. With coarse-grained feedback signals, such as ECN or RTT, current CC schemes do not know exactly how much to increase or decrease sending rates. Therefore, they use heuristics to guess the rate updates and try to iteratively converge to a stable rate distribution. Such iterative methods are slow for handling large-scale congestion events[25], as we can see in Case-1.

Unavoidable packet queueing. A DCQCN sender leverages the one-bit ECN mark to judge the risk of congestion, and a TIMELY sender uses the increase of RTT to detect congestion. Therefore, the sender starts to reduce flow rates only after a queue builds up. These built-up queues can significantly increase the network latency, and this is exactly the issue met by the ML application at the beginning in Case-2.

Complicated parameter tuning. The heuristics used by current CC algorithms to adjust sending rates have many parameters to tune for a specific network environment. For instance, DCQCN has 15 knobs to set up. As a result, operators usually face a complex and time-consuming parameter tuning stage in daily RDMA network operations, which significantly increases the risk of incorrect settings that cause instability or poor performance.

The fundamental cause of the preceding three limitations is the lack of fine-grained network load information in legacy networks – ECN is the only feedback an end host can get from switches, and RTT is a pure end-to-end measurement without switches' involvement. However, this situation has recently changed. With In-network telemetry (INT) features that have become available in new switching ASICs [2–4], obtaining fine-grained network load information and using it to improve CC has become possible in production networks.

In this paper, we propose a new CC mechanism, HPCC (High Precision Congestion Control), for large-scale, high-speed networks. The key idea behind HPCC is to leverage the precise link load information from INT to compute accurate flow rate updates. Unlike existing approaches that often require a large number of iterations to find the proper flow rates, HPCC requires only one rate update step in most cases. Using precise information from INT enables HPCC to address the three limitations in current CC schemes. First, HPCC senders can quickly ramp up flow rates for high utilization or ramp down flow rates for congestion avoidance. Second, HPCC senders can quickly adjust the flow rates to keep each link's input rate slightly lower than the link's capacity, preventing queues from being built-up as well as preserving high link utilization. Finally, since sending rates are computed precisely based on direct measurements at switches, HPCC requires merely 3 independent parameters that are used to tune fairness and efficiency.

On the flip side, leveraging INT information in CC is not straightforward. There are two main challenges to design HPCC. First, INT information piggybacked on packets can be delayed by link congestion, which can defer the flow rate reduction for resolving the congestion. In HPCC, our CC algorithm aims to limit and control the total inflight bytes to busy links, preventing senders from sending extra traffic even if the feedback gets delayed. Second, despite that INT information is in all the ACK packets, there can be destructive overreactions if a sender blindly reacts to all the information for fast reaction (§3.2). Our CC algorithm selectively uses INT information by combining per-ACK and per-RTT reactions, achieving fast reaction without overreaction.

HPCC meets our goals of achieving ultra-low latency, high bandwidth, and high stability simultaneously in large-scale high-speed networks. In addition, HPCC also has the following essential properties for being practical: ***(i)*** *Deployment ready*: It merely requires standard INT features (with a trivial and optional extension for efficiency) in switches and is friendly to implementation in NIC hardware. ***(ii)*** *Fairness*: It separates efficiency and fairness control. It uses multiplicative increase and decrease to converge quickly to the proper rate on each link, ensuring efficiency and stability, while it uses additive increase to move towards fairness for long flows.

HPCC's stability and fairness are guaranteed in theory (Appendix A). We implement HPCC on commodity NIC with FPGA and commodity switching ASIC with P4 programmability. With testbed experiments and large-scale simulations, we show that compared with DCQCN, TIMELY and other alternatives, HPCC reacts faster to available bandwidth and congestion and maintains close-to-zero queues. In our 32-server testbed, even under 50% traffic load, HPCC keeps the queue size zero at the median and 22.9KB (only 7.3μs queueing delay) at the 99th-percentile , which results in a 95% reduction in the 99th-percentile latency compared to DCQCN without sacrificing throughput. In our 320-server simulation, even under incast events where PFC storms happen frequently with DCQCN and TIMELY, PFC pauses are not triggered with HPCC.

Note that despite HPCC having been designed from our experiences with RDMA networks, we believe its insights and designs are also suitable for other high-speed networking solutions in general.

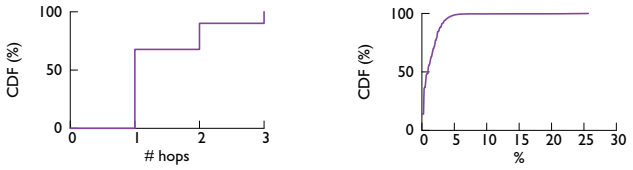
2 EXPERIENCE AND MOTIVATION

In this section, we present our production data and experiences that demonstrate the difficulty to operate large-scale, high-speed RDMA networks due to current CC schemes' limitations. We also propose some key directions and requirements for the next generation CC of high-speed networks.

2.1 Our large RDMA deployments

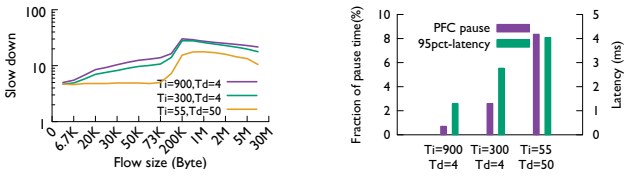
We adopt RDMA in our data centers for ultra-low latency and large bandwidth demanded by multiple critical applications, such as distributed storage, database, and deep learning training frameworks.

Our data center network is a Clos topology with three layers – ToR, Agg, and Core switches. A PoD (point-of-delivery), which



(a) Propagation depth (b) Suppressed bandwidth

Figure 1: The impacts of PFC pauses in production.



(a) 95-percentile FCT under normal case with 30% network load

(b) PFC duration and latency with 30% network load + incast

Figure 2: FCT slowdown and PFC pauses with different rate increasing timers in DCQCN, using *WebSearch*.

consists of tens of ToR switches that are interconnected by a number of Agg switches, is a basic deployment unit. Different PoDs are interconnected by Core switches. Each server has two uplinks connected with two ToR switches for high availability of servers, as required by our customers. In the current RDMA deployment, each PoD is an independent RDMA domain, which means that only servers within the same PoD can communicate with RDMA.

We use the latest production-ready version of RoCEv2: DCQCN is used as the congestion control (CC) solution which is integrated into hardware by RDMA NIC vendors. PFC [1] is enabled in NICs and switches for lossless network requirements. The strategy to recover from packet loss is "go-back-N", which means a NACK will be sent from receiver to sender if the former finds a lost packet, and the sender will resend all packets starting from the lost packet.

There have been tens of thousands of servers supporting RDMA, carrying our databases, cloud storage, data analysis systems, HPC and machine learning applications in production. Applications have reported impressive improvements by adopting RDMA. For instance, distributed machine learning training has been accelerated by 100+ times compared with the TCP/IP version, and the I/O speed of SSD-based cloud storage has been boosted by about 50 times compared to the TCP/IP version. These improvements majorly stem from the hardware offloading characteristic of RDMA.

2.2 Our goals for RDMA

Besides ultra-low latency and high bandwidth, network stability and operational complexity are also critical in RDMA networks, because RDMA networks face more risks and tighter performance requirements than TCP/IP networks.

First of all, RDMA hosts are aggressive for resources. They start sending at line rate, which makes common problems like incast much more severe than TCP/IP. The high risk of congestion also means a high risk to trigger PFC pauses.

Second, PFC has the potential for large and destructive impact on networks. PFC pauses all upstream interfaces once it detects a risk of packet loss, and the pauses can propagate via a tree-like graph to multiple hops away. Such spreading of congestion can possibly trigger PFC deadlocks [21, 23, 38] and PFC storms (Case-1 in §1) that can silence a lot of senders even if the network has free capacity. Despite the probability of PFC deadlocks and storms being fairly small, they are still big threats to operators and applications, since currently we have no methods to guarantee they won't occur [23].

Third, even in normal cases, PFC can still suppress a large number of innocent senders. For instance, by monitoring the propagation graph of each PFC pause in a PoD, we can see that about 10% of PFC events propagate three hops (Figure 1a), which means the whole PoD is impacted due to a single or a small number of senders. Figure 1b shows that more than 10% PFC pauses suppress more than 3% of the total network capacity of a data center, and in the worst case the capacity loss can be 25%! Again, we can see that the 25% capacity loss is rare, but it is still a threat which operators have to plan for.

Finally, operational complexity is an important factor that is previously neglected. Because of the high performance requirement and stability risks, it often takes months to tune the parameters for RDMA before actual deployment, in order to find a good balance. Moreover, because different applications have different traffic patterns, and different environments have different topologies, link speeds, and switch buffer sizes, operators have to tune parameters for the deployment of each new application and new environment.

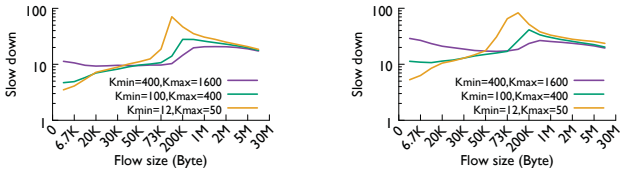
Therefore, we have four essential goals for our RDMA networks: (i) latency should be as low as possible; (ii) bandwidth/utilization should be as high as possible; (iii) congestion and PFC pauses should be as few as possible; (iv) the operational complexity should be as low as possible. Achieving the four goals will provide huge value to our customers and ourselves, and we believe the key to achieving them is a proper CC mechanism.

2.3 Trade-offs in current RDMA CC

DCQCN is the default CC in our RDMA networks. It leverages ECN to discover congestion risk and reacts quickly. It also allows hosts to begin transmitting aggressively at line rate and increase their rates quickly after transient congestion (*e.g.* FastRecovery [43]). Nonetheless, its effectiveness depends on whether its parameters are suitable to specific network environments.

In our practice, operators always struggle to balance two trade-offs in DCQCN configurations: throughput v.s. stability, *e.g.* Case-1 in §1, and bandwidth v.s. latency, *e.g.* Case-2 in §1. To make it concrete, since we cannot directly change configurations in production, we highlight the two trade-offs with experiments on a testbed that has similar hardware/software environments but a smaller topology compared to our production networks. The testbed is a PoD with 230 servers (each has two 25Gbps uplinks), 16 ToR switches and 8 Agg switches connected by 100Gbps links. We intentionally use public traffic workloads, *e.g. WebSearch* [8] and *FB_Hadoop* [37], instead of our own traffic traces for reproductivity.

Throughput vs. Stability It is hard to achieve high throughput without harming the network's stability in one DCQCN configuration. To quickly utilize free capacity, senders must have high sensitivity to available bandwidth and increase flow rates fast, while such aggressive behavior can easily trigger buffer overflows and traffic oscillations in the network, resulting in large scale PFC pauses. For



(a) 30% network load. (b) 50% network load.

Figure 3: 95-percentile FCT slowdown distribution with different ECN thresholds, using *WebSearch*.

example, Figure 2 approximately shows the issue in Case-1 of §1. Figure 2a shows the FCT (flow completion time) slowdown[1] at 95-percentile under different DCQCN rate-increasing timers (T_i) and rate-decreasing timers (T_d) with 30% average network load from *WebSearch*. Setting $T_i = 55\mu s$, $T_d = 50\mu s$ is from DCQCN's original paper; $T_i = 300\mu s$, $T_d = 4\mu s$ is a vendor's default; and $T_i = 900\mu s$, $T_d = 4\mu s$ is a more conservative version from us. Figure 2a shows that smaller (T_i) and larger (T_d) reduce the FCT slowdown because they make senders more aggressive to detect and utilize available bandwidth. However, smaller T_i and larger T_d is more likely to have more and longer PFC pauses compared to more conservative timer settings during incast events. Figure 2b shows the PFC pause duration and 95-percentile latency of short flows when there are incast events whose total load is 2% of the network's total capacity. Each incast event is from 60 senders to 1 receiver. We can see that smaller (T_i) and larger (T_d) suffers from longer PFC pause durations and larger tail latencies of flows. We also have tried out different DCQCN parameters, different average link loads and different traffic traces, and the trade-off between throughput and stability remains.

Bandwidth vs. Latency Though "high bandwidth and low latency" has become a "catchphrase" of RDMA, we find it is practically hard to achieve them simultaneously in one DCQCN configuration. This is because for consistently low latency the network needs to maintain steadily small queues in buffers (which means low ECN marking thresholds), while senders will be too conservative to increase flow rates if ECN marking thresholds are low. For example, Figure 3 approximately shows the issue in Case-2 of §1. It shows the FCT slowdown with different ECN marking thresholds (K_{min}, K_{max}) in switches and *WebSearch* as input traffic loads. Figure 3a shows that when we use low ECN thresholds, small flows which are latency-sensitive have lower FCT, while big flows which are bandwidth-sensitive suffer from larger FCT. The trend is more obvious when the network load is higher (Figure 3b when the average link load is 50%). For instance, the 95th-percentile RTT is about $150\mu s - 30$ (slowdown) $\times 5\mu s$ (baseline RTT) — when K_{min} = 400KB, K_{max} = 1600KB, which is a lot worse than the ML application's requirement (<50μs) in Case-2. We have tried out different DCQCN parameters, different average link loads and different traffic traces, and the trade-off between bandwidth and latency remains.

As mentioned in §1, we are usually forced to sacrifice utilization (or money) to achieve latency and stability. The unsatisfactory outcome made us rethink about the fundamental reasons for the tight tensions among latency, bandwidth, and stability. Essentially, as the first generation of CC for RDMA designed more than 5 years ago, DCQCN has several design issues due to the limitations of hardware when it was proposed, which results in the challenges to network operations. For instance, DCQCN's timer-based scheduling inherently creates the tradeoff between throughput (more aggressive timers) and stability (less aggressive timers), while its ECN (queue) based congestion signaling directly results in the trade-off between latency (lower ECN thresholds) and bandwidth (higher ECN thresholds). Other than the preceding two trade-offs, the timer-based scheduling can also trigger traffic oscillations during link failures; the queue-based feedback also creates a new trade-off between ECN threshold and PFC threshold. We omit the details due to space limit.

Further, though we have less production experience with TIMELY, Microsoft reports that TIMELY's performance is comparable to or worse than DCQCN [44], which is also validated in §5.3.

[1] "FCT slowdown" means a flow's actual FCT normalized by its ideal FCT when the network only has this flow.

2.4 Next generation of high-speed CC

We advocate that the next generation of CC for RDMA or other types of high-speed networks should have the following properties *simultaneously* to make a significant improvement on both application performance and network stability:

(i) Fast converge. The network can quickly converge to high utilization or congestion avoidance. The timing of traffic adjustments should be adaptive to specific network environments rather than manually configured.

(ii) Close-to-empty queue. The queue sizes of in-network buffers are maintained steadily low, close-to-zero.

(iii) Few parameters. The new CC should not rely on lots of parameters that require the operators to tune. Instead, it should adapt to the environment and traffic pattern itself, so that it can reduce the operational complexity.

(iv) Fairness. The new CC ensures fairness among flows.

(v) Easy to deploy on hardware. The new CC algorithm is simple enough to be implemented on commodity NIC hardware and commodity switch hardware.

Nowadays, we have seen two critical trends that have the potential to realize a CC which satisfies all of the preceding requirements. The first trend is that switches are more open and flexible in the data plane. Especially, in-network telemetry (INT) is being popularized quickly. Almost all the switch vendors we know have INT feature enabled already in their new products (e.g., Barefoot Tofino [2], Broadcom Tomahawk3 [3], Broadcom Trident3 [4], *etc.*). With INT, a sender can know exactly the loads of the links along a flow's path from an ACK packet, which facilitates the sender to make accurate flow rate adjustments. The second trend is that NIC hardware is becoming more capable and programmable. They have faster speed and more resources to expose packet level events and processing. With these new hardware features, we design and implement HPCC, which achieves the desired CC properties simultaneously.

3 DESIGN

The key design choice of HPCC is to rely on switches to provide fine-grained load information, such as queue size and accumulated tx/rx traffic to compute precise flow rates. This has two major benefits: (i) HPCC can quickly converge to proper flow rates to

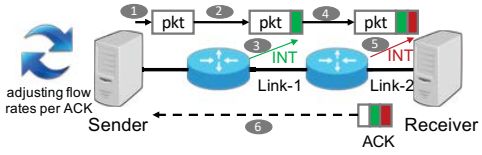


Figure 4: The overview of HPCC framework.

highly utilize bandwidth while avoiding congestion; and (ii) HPCC can consistently maintain a close-to-zero queue for low latency.

Nonetheless, there are two major challenges to realize the design choice. First, during congestion, feedback signals can be delayed, causing a high rate to persist for a long time. This results in much more inflight data from each sender than needed to sustain high utilization; as our experiment in §2.3 shows, each sender can have significantly more inflight data than the BDP (Bandwidth-delay product)[2]. To avoid this problem, HPCC directly controls the number of inflight bytes, in contrast to DCQCN and TIMELY that only control the sending rate. In this way, even if feedback signals are delayed, the senders do not send excessive packets because the total inflight bytes are limited. Second, while a HPCC sender can react to network load information in each ACK, it must carefully navigate the tension between reacting quickly and overreacting to congestion feedback. We combine RTT-based and ACK-based reactions to overcome this tension.

3.1 HPCC framework

HPCC is a sender-driven CC framework. As shown in Figure 4, each packet a sender sends will be acknowledged by the receiver. During the propagation of the packet from the sender to the receiver, each switch along the path leverages the INT feature of its switching ASIC to insert some meta-data that reports the current load of the packet's egress port, including timestamp (ts), queue length ($qLen$), transmitted bytes ($txBytes$), and the link bandwidth capacity (B).

When the receiver gets the packet, it copies all the meta-data recorded by the switches to the ACK message it sends back to the sender. The sender decides how to adjust its flow rate each time it receives an ACK with network load information.

3.2 CC based on inflight bytes

HPCC is a window-based CC scheme that controls the number of inflight bytes. The inflight bytes mean the amount of data that have been sent, but not acknowledged at the sender yet.

Controlling inflight bytes has an important advantage compared to controlling rates. In the absence of congestion, the inflight bytes and rate are interchangeable with equation $inflight = rate \times T$ where T is the base propagation RTT. However, controlling inflight bytes greatly improves the tolerance to delayed feedback during congestion. Compared to a pure rate-based CC scheme which continuously sends packets before feedback comes, the control on the inflight bytes ensures the number of inflight bytes is within a limit, making senders immediately stop sending when the limit is reached, no matter how long the feedback gets delayed. As a result, the whole network is greatly stabilized.

Senders limit inflight bytes with sending windows. Each sender maintains a sending window, which limits the inflight bytes it can send. Using a window is a standard idea in TCP, but the benefits for tolerance to feedback delays are substantial in data centers, because the queueing delay (hence the feedback delay) can be orders of magnitude higher than the ultra-low base RTT [26]. The initial sending window size should be set so that flows can start at line rate, so we use $W_{init} = B_{NIC} \times T$, where B_{NIC} is the NIC bandwidth.

In addition to the window, we also pace the packet sending rate to avoid bursty traffic. Packet pacer is generally available in NICs [43]. The pacing rate is $R = \frac{W}{T}$, which is the rate that a window size W can achieve in a network with base RTT T.

Congestion signal and control law based on inflight bytes. In addition to the sending window, HPCC's congestion signal and control law are also based on the inflight bytes.

The inflight bytes directly corresponds to the link utilization. Specifically, for a link, the inflight bytes is the total inflight bytes of all flows traversing it. Assume a link's bandwidth is B, and the i-th flow traversing it has a window size W_i. The inflight bytes for this link is $I = \sum W_i$.

If $I < B \times T$, we have $\sum \frac{W_i}{T} < B$. $\frac{W_i}{T}$ is the throughput that flow i achieves if there is no congestion. So in this case, the total throughput of all these flows is lower than the link bandwidth.

If $I \geq B \times T$, we have $\sum \frac{W_i}{T} \geq B$. In this case, there must be congestion (otherwise, the total throughput would exceed the link capacity which is impossible), and queues form. The congestion can be on this link, or somewhere else if there are multiple bottlenecks.

So our goal is to control I to be slightly smaller than $B \times T$ for every link, such that there is no congestion and no queues.

Estimating the inflight bytes for each link. The first question is how a sender uses INT information to estimate I_j for each link j on its path. Specifically, the inflight bytes consist of data packets in the queues and in the pipeline. So for each link j, we estimate I_j using its queue length ($qlen$) and its output rate ($txRate$), as in Eqn (1):

$$I_j = qlen + txRate \times T \tag{1}$$

where $qlen$ is directly from INT. $txRate$ is calcualted with $txBytes$ and ts: $txRate = \frac{ack_1.txBytes - ack_0.txBytes}{ack_1.ts - ack_0.ts}$ where ack_1 and ack_0 are two ACKs. $txRate \times T$ estimates the number of bytes in the pipeline. Eqn (1) assumes all flows have the same known base RTT. This is possible in data centers, where the RTT between most server pairs are very close due to the regularity of the topology.

In the most common congestion scenario where there is one bottleneck j, I_j is the estimation of the total inflight bytes of all flows traversing link j. In the case where some flows traverse multiple bottlenecks, I_j is the lower bound of total inflight bytes.

Reacting to the signals. Each sender should adjust its window so that I_j for each link j on its flow's path is slightly lower than $B_j \times T$– specifically, to be $\eta \times B_j \times T$ (η is a constant close to 1, e.g., 95%). Thus, for link j, each sender can multiplicatively reduce its window by a factor of $k_j = \frac{I_j}{\eta \times B_j \times T} = U_j/\eta$, where U_j is the normalized inflight bytes of link j:

$$U_j = \frac{I_j}{B_j \times T} = \frac{qlen_j + txRate_j \times T}{B_j \times T} = \frac{qlen_j}{B_j \times T} + \frac{txRate_j}{B_j} \tag{2}$$

[2]In Figure 2b, the PFC being propagated to hosts means at least 3 switches (intra-PoD) has reached the PFC threshold. So the inflight bytes is at least 11×BDP per flow on average, calculated based on our data center spec and the incast ratio.

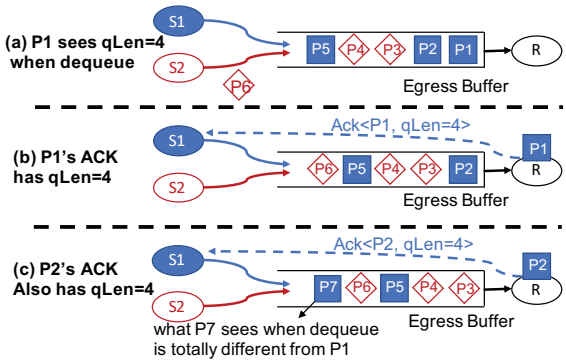


Figure 5: Overreaction to two sequential ACKs.

Sender i should react to the most congested link:

$$W_i = \frac{W_i}{max_j(k_j)} + W_{AI} = \frac{W_i}{max_j(U_j)/\eta} + W_{AI} \quad (3)$$

where W_{AI} is an additive increase (AI) part to ensure fairness, which is very small. Note that the first term in Eqn (3) is an MIMD term. This decouples the utilization control from the fairness control to ensure senders quickly grab free bandwidth or avoid congestion, which is inspired by XCP [27].

Note that if there are multiple bottlenecks, Eqn (1) under-estimates the number of inflight bytes, so HPCC needs multiple rounds of adjustments to resolve congestion. However, during incast cases, which are the most common congestion cases in data center [39], there is only one bottleneck, so HPCC can resolve the congestion with only one round of adjustment (Appendix A).

Fast reaction without overreaction. With Eqn (2) and Eqn (3), an HPCC sender can react with every ACK. Reacting to each ACK enables fast congestion avoidance, but it reacts multiple times to ACKs describing the same packets and queues. Figure 5 shows an example of how reacting to each ACK causes overreaction. At the beginning, in Figure 5(a), there are three packets (P_1, P_2 and P_5) from sender S_1 and two packets (P_3 and P_4) from sender S_2 queued in an egress buffer of a link. When P_1 is dequeued, the buffer has 4 packets. So its ACK to S_1 indicates $qLen = 4$ (Figure 5(b)). Suppose S_1 uses Eqn (2) and Eqn (3) to decide $W(1) = W(0)/2$, where $W(0)$ is the window size of S_1 in Figure 5(a). However, P_2 also sees 4 packets when it is dequeued in Figure 5(b) and its ACK to S_1 has $qLen = 4$ in Figure 5(c). If S_1 blindly updates its window size based on P_2's ACK, it ends up with $W(2) = W(1)/2 = W(0)/4$. This conservative window size is an overreaction because P_1 and P_2's ACKs report the link conditions for almost the same set of packets.

One way to prevent overreaction is to make sure that the window is only adjusted when an ACK that describes a brand new set of packets is received. For instance, in Figure 5(c), P_7 is a packet which is sent out from S_1 after S_1 gets the ACK of P_1. Therefore, the packets P_7 sees in any queue are totally different from what P_1 sees. In other words, the network status reported in P_1's and P_7's ACKs have no overlap. Therefore, for avoiding overreaction, we always remember the first packet (Q) sent right after the window is adjusted and only adjust the window again when the sender gets Q's ACK. However, the drawback of this strategy is that merely updating window each RTT might be too slow for handling urgent cases like failures and incasts (§5.4).

HPCC combines the per-ACK and per-RTT strategies to achieve fast reaction without overreaction. The key idea is to introduce a reference window size W_i^c, a runtime state updated in per-RTT basis. Hence, only when receiving the ACK of the first packet sent with the current W_i^c, we update it with $W_i^c = W_i$, i.e., assigning the current window size W_i to the reference window size W_i^c. With W_i^c, the sender can safely update its window size using Eqn (4):

$$W_i = \frac{W_i^c}{\max_j(U_j)/\eta} + W_{AI} \quad (4)$$

Since W_i is computed from W_i^c which is fixed in a RTT, the sender does not overreact to the same network loads. For example, in the case of Figure 5, we have $W(2) = W(1) = W(0)/2 = W^c/2$ even if S_1 recomputes window sizes on ACKs of both P_1 and P_2. Meanwhile, if the inflight bytes dramatically change within an RTT, the window size is still adjusted by Eqn (4) because U_j is updated.

Algorithm 1 Sender algorithm. $ack.L$ is an array of link feedbacks in the ACK; each link $ack.L[i]$ has four fields: $qlen$, $txBytes$, ts, and B. L is the sender's record of link feedbacks at the last update.

```
1:  function MeasureInflight(ack)
2:      u = 0;
3:      for each link i on the path do
4:          txRate = (ack.L[i].txBytes − L[i].txBytes) / (ack.L[i].ts − L[i].ts);
5:          u' = min(ack.L[i].qlen, L[i].qlen) / (ack.L[i].B·T) + txRate / ack.L[i].B;
6:          if u' > u then
7:              u = u'; τ = ack.L[i].ts − L[i].ts;
8:      τ = min(τ, T);
9:      U = (1 − τ/T) · U + τ/T · u;
10:     return U;
11: function ComputeWind(U, updateWc)
12:     if U >= η or incStage >= maxStage then
13:         W = W^c/(U/η) + W_AI;
14:         if updateWc then
15:             incStage = 0; W^c = W;
16:     else
17:         W = W^c + W_AI;
18:         if updateWc then
19:             incStage++; W^c = W;
20:     return W;
21: procedure NewAck(ack)
22:     if ack.seq>lastUpdateSeq then
23:         W = ComputeWind(MeasureInflight(ack), True);
24:         lastUpdateSeq = snd_nxt;
25:     else
26:         W = ComputeWind(MeasureInflight(ack), False);
27:     R = W/T; L = ack.L;
```

The overall workflow of the sender side CC algorithm. Algorithm 1 illustrates the overall process of CC at the sender side for a single flow. Each newly received ACK message triggers the procedure NewACK at Line 21. At Line 22, the variable $lastUpdateSeq$ is used to remember the first packet sent with a new W^c, and the sequence number in the incoming ACK should be larger than $lastUpdateSeq$ to trigger a new sync between W^c and W (Line 14-15 and 18-19). The sender also remembers the pacing rate and current

INT information at Line 27. The sender computes a new window size W at Line 23 or Line 26, depending on whether to update W^c, with function MeasureInflight and ComputeWind.

Function MeasureInflight estimates normalized inflight bytes with Eqn (2) at Line 5. First, it computes $txRate$ of each link from the current and last accumulated transferred bytes $txBytes$ and timestamp ts (Line 4). It also uses the minimum of the current and last $qlen$ to filter out noises in $qlen$ (Line 5). The loop from Line 3 to 7 selects $\max_i(U_i)$ in Eqn. (3). Instead of directly using $\max_i(U_i)$, we use an EWMA (Exponentially Weighted Moving Average) to filter the noises from timer inaccuracy and transient queues. (Line 9).

Function ComputeWind combines multiplicative increase/decrease (MI/MD) and additive increase (AI) to balance the reaction speed and fairness. If a sender finds it should increase the window size, it first tries AI for $maxStage$ times with the step W_{AI} (Line 17). If it still finds room to increase after $maxStage$ times of AI or the normalized inflight bytes is above η, it calls Eqn (4) once to quickly ramp up or ramp down the window size (Line 12-13).

3.3 Parameters of HPCC

HPCC has three easy-to-set parameters: η, $maxStage$, and W_{AI}. η controls a simple tradeoff between utilization and transient queue length (due to the temporary collision of packets caused by their random arrivals. See Appendix A.1), so we set it to 95% by default, which only loses 5% bandwidth but achieves almost zero queue. $maxStage$ controls a simple tradeoff between steady state stability and the speed to reclaim free bandwidth. We find $maxStage = 5$ is conservatively large for stability, while the speed of reclaiming free bandwidth is still much faster than traditional additive increase, especially in high bandwidth networks. W_{AI} controls the tradeoff between the maximum number of concurrent flows on a link that can sustain near-zero queues and the speed of convergence to fairness (Appendix A.3). Normally we set a very small W_{AI} to support a large number of concurrent flows on a link, because slower fairness is not critical. A rule of thumb is to set $W_{AI} = \frac{W_{init} \times (1-\eta)}{N}$, where N is the expected maximum number of concurrent flows on a link. The intuition is that the total additive increase every round ($N \times W_{AI}$) should not exceed the bandwidth headroom, and thus no queue forms. Even if the actual number of concurrent flows on a link exceeds N, the CC is still stable and achieves full utilization, but just cannot maintain zero queues. Note that none of the three parameters are reliability-critical.

3.4 Properties of HPCC

HPCC has fewer parameters and the tuning is simpler than previous CC schemes. Most previous CC schemes, such as DCQCN[43], TIMELY[31] and DCTCP[8] which are productionized, do not have precise feedback, so they have to use heuristics to infer the current network state. These heuristics work differently in different environments, so they have parameters for operators to tune for the environments. On the other hand, HPCC uses precise feedback to know the exact network state, so HPCC does not need heuristics, and thus no need for the associated parameters.

For example, they heuristically maintain the equilibrium during steady states (i.e., there are a fixed number of flows). Specifically, the AI step and MD factor should be in a dynamic equilibrium.

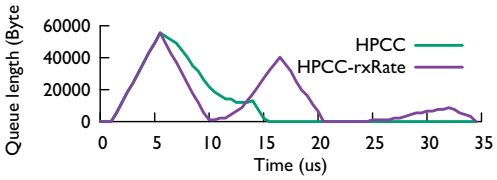


Figure 6: Comparison of txRate and rxRate.

The smaller AI step, the stabler equilibrium and slower fairness convergence, but once some flow finishes, it takes longer time to reclaim the free bandwidth. So operators have to carefully tune the AI step to achieve a good tradeoff; DCQCN and TIMELY even have two parameters for AI steps. In contrast, HPCC reclaims bandwidth through MI, enabled by the precise link load information, so the AI step can be small and is easy to set.

For the MD factor, they use EWMA to gradually find out the right value. Average over longer terms give more precise MD factor and thus stabler equilibrium, but once more flows join, it takes longer time to resolve the congestion. So operators have to also carefully tune α (the parameter that controls the weight of new feedback). In contrast, HPCC directly knows the MD factor (Eqn (3)), so it does not have this parameter. Note that the EWMA in HPCC is parameterless: the weights of new ACKs are automatically scaled with inter-packet time gaps.

There are also CC schemes that use explicit feedback, such as XCP[27] and RCP[16]. However, their congestion signals are heuristic combinations of different types of feedback, so they have scaling parameters to tune their relative significance. In contrast, HPCC's congestion signal has a concrete physical meaning–the inflight bytes, so it does not need parameters.

Key insight on using txRate. Using $txRate$ in Eqn (3) allows HPCC to accurately estimate the amount of inflight bytes, whereas the same equation with $rxRate$ has no concrete physical meaning. In fact, $rxRate$ and $qlen$ overlap in terms of the congestion that they measure: a high $rxRate$ increases the queue occupancy, which implies a large $qlen$. Therefore, schemes like XCP[27] and RCP[16] that combine terms based on $rxRate$ and $qlen$ to measure the extent of congestion require scaling parameters to tune the relative importance of these terms. Further, as first observed in ABC [20], using $txRate$ improves the accuracy of the feedback signal in a window-based CC scheme like HPCC. The reason is that the $txRate$ at a switch queue reflects the $rxRate$ at that queue *one RTT in the future*, since packet transmissions from senders are clocked by acknowledgements in a window-based scheme. Therefore, by adjusting the window size based on $txRate$, the senders can anticipate what the extent of congestion will be one RTT after the measurement is taken at the switch, and react more accurately compared to when using $rxRate$.

We perform a simple experiment to compare the use of $txRate$ and $rxRate$. We use HPCC and HPCC-rxRate (replacing $txRate$ with $rxRate$ in all calculations) in a simple 2-to-1 congestion scenario. Figure 6 shows the queue length over time. We can see that using $rxRate$ has oscillation before it finally converges, which is a result of the aforementioned problems. On the other hand, using $txRate$ gracefully converges without oscillation.

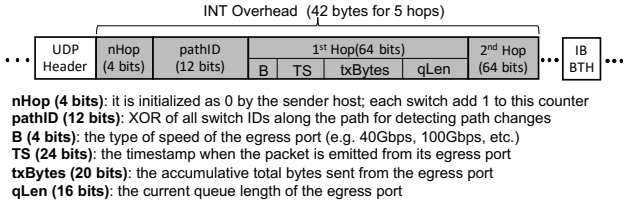


Figure 7: The packet format of HPCC.

Theoretical properties of HPCC. For simple models of a system with arbitrary network topology and with multiple bottleneck links, we prove that HPCC has rapid convergence to a Pareto optimal rate allocation, followed by a slower convergence to fairness. Appendix A has a detailed analysis of HPCC's properties.

4 IMPLEMENTATION

We implement a prototype of HPCC in commodity NICs with FPGA programmability to realize the CC algorithm (§3.2) and commodity switching ASICs with P4 programmability to realize a standard INT feature[3] (§3.1). We also implemented DCQCN on the same hardware platform for fair comparisons.

4.1 INT padding at switches

HPCC only relies on packets to share information across senders, receivers, and switches. Figure 7 shows the packet format of the INT padding after UDP header and before IB BTH (Base Transport Header) as in RoCEv2 standard. The field *nHop* is the hop count of the packet's path. The field *pathID* is the XOR of all the switch IDs (which are 12 bits) along the path. The sender sets *nHop* and *pathID* to 0. Each switch along the path adds *nHop* by 1, and XORs its own switch ID to the *pathID*. The sender uses *pathID* to judge whether the path of the flow has been changed. If so, it throws away the existing status records of the flow and builds up new records.

Each switch has an 8-byte field to record the status of the egress port of the packet when the packet is emitted. *B* is a *enum* type which indicates the speed type of the port. The timestamp (*TS*), total bytes sent so far (*txBytes*, in units of 128 Bytes) and the queue length (*qLen*, in units of 80 Bytes) are all standard INT information.

The overhead of the INT padding for HPCC is low. Inside a data center, the path length is often no more than 5 hops, so the total padding is at most 42 bytes, which is only 4.2% in a 1KB packet.

Our switch also has modules such as destination-based ECMP routing, QoS, WRED, PFC, *etc.*.

4.2 Congestion control at NICs

Figure 8 shows HPCC implementation on a commodity programmable NIC. The NIC provides an FPGA chip which is connected to the main memory with a vendor-provided PCIe module and the Ethernet adapter with a vendor-provided MAC module. Sitting between the PCIe and MAC modules, HPCC's modules realize both sender and receiver roles.

The Congestion Control (CC) module implements the sender side CC algorithm. It receives `ACK` events which are generated from the RX pipeline, adjusts the sending window and rate, and stores the new sending window and rate for the flow of the current ACK in the flow scheduler via an `Update` event.

[3]INT only requires a tiny subset of P4 programmability. Our other non-P4 ASIC vendors also provide the same INT features in their new releases.

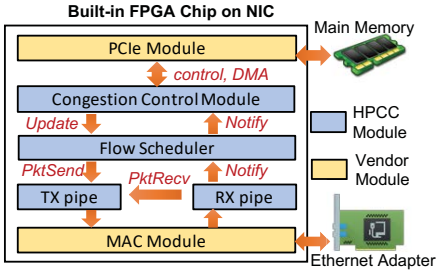


Figure 8: Overview of HPCC's NIC implementation.

The flow scheduler paces flow rates with a credit-based mechanism. Specifically, it scans through all the flows in a round-robin manner and assigns credit to each flow proportional to its current pacing rate. It also maintains the current sending window size and unacknowledged packets for active flows. If a flow has accumulated sufficient credits to send one packet and the flow's sending window permits, the flow scheduler invokes a `PktSend` event to TX pipe.

The TX pipe implements IB/UDP/IP stacks for running in RoCEv2. It maintains the flow context for each of concurrent flows, including 5-tuples, the packet sequence number (PSN), destination QP (queue pair), etc. Once it receives the `PktSend` event with QP ID from the flow scheduler, it generates the corresponding packet and delivers to the MAC module.

The RX pipe parses the incoming packets from the MAC module and generates multiple events to other HPCC modules. (1) On receiving a data packet, the RX pipe extracts its flow context and invokes a `PktRecv` event to the TX pipe to formulate a corresponding ACK packet. If the packet is out-of-sequence (OOS), the TX pipe sends a NAK instead. (2) On receiving an ACK packet, the RX pipe extracts the network status from the packet and passes it to the CC module via the flow scheduler. (3) On receiving a NAK, the RX pipe notifies the TX pipe to start go-back-to-N retransmission. (4) On receiving a control packet with an RDMA operation, the RX pipe notifies the flow scheduler to create a flow with a new QP ID, or remove an existing flow. Currently, HPCC supports two operations: RDMA WRITE and RDMA READ. We leave the full support of IB verbs as future work.

4.3 Performance optimization

We did many performance optimizations in our hardware implementation. Here are two examples:

Accelerating divisions in hardware. After receiving an ACK, the CC module needs to recompute the window, which requires divisions in Eqn (4). However, divisions are expensive operations especially in FPGA. We design a lookup table to replace division operations by applying the multiplication operation on the value of $\frac{1}{n}$, where n is an integer. To reduce the table size while constraining the estimation error, we choose store n values whose difference with the previous stored one is larger than ϵ (i.e., $\frac{1}{n_{k+1}} - \frac{1}{n_k} \geq \epsilon \times \frac{1}{n_k}$). As a result, we can speed up the division option in Eqn (4) by about 8 times. The hardware memory overhead is negligible. In our

current implementation, we merely use about 10KB to represent $\{\frac{1}{n}|1 \le n \le 2^{22}, n \in \mathbb{N}\}$.

Supporting many concurrent flows. The bottleneck to supporting a large number of concurrent flows is the speed of the clock engine in the hardware. Because the flow scheduler uses round robin over a fixed size array to schedule different flows, it can only support up to 50 concurrent flows at line rate with a single engine. To support more concurrent flows, we use multiple independent engines to schedule multiple independent arrays of flows. The FPGA we use in our prototype has six engines, which means we can support 300 concurrent flows per 25GE interface. We expect to be able to support 9K flows in ASIC implementations given that the ASIC's clock is much faster (*e.g.* 0.2ns per tick) than our FPGA's clock (*e.g.* 5ns), which is sufficient in data centers.

4.4 Complexity and overhead

Our NIC implementation takes about 12000 lines of Verilog code for the flow scheduler, RX/TX pipes, register profiles, and top flow controls. The CC modules for HPCC and DCQCN have about 2000 and 800 lines of Verilog code respectively. The hardware resources (*e.g.* CLB LUTs, CLB register, Block RAM, DSP, *etc.*) used by both HPCC and DCQCN are less than 2% of the total in the FPGA.

We believe HPCC can be easily implemented in the next-generation RoCE NICs. This is because HPCC conforms to the paradigm of existing RoCE NICs, so it just needs simple logic changes, rather than architectural changes. Specifically, it has a simple receiver and three components at the sender: measurement, calculation, and traffic enforcement, which are already in RoCE NICs, unlike some other paradigms that are architecturally different, such as receiver-driven or credit-based CC[14, 18, 22, 33]. The major changes are incremental. We just need an INT parser at the measurement part which is just another type of header parsing, and changes to the calculation part are also simple as demonstrated in our prototype.

Our switch side implementation consists of about 300 lines of P4 code and 700 lines of configurations via Program-Dependent (PD) APIs in the control plane. The regular modules (*e.g.* QoS, WRED, *etc.*) are all standard modules used in today's commodity switches. The additional resources used to support INT function are small over baseline *switch.p4* [6] with 25% more stateful ALU usage and a 5% increase in memory and Packet Header Vector resources.

Difficulty to implement TCP-like CC in hardware. From our experience in implementation of HPCC and DCQCN and our conversations with NIC vendors, we found it is hard to implement TCP-like CC algorithms which use sliding windows. The primary reason is sliding windows should support retransmissions of arbitrary packet losses, so they needs random access to memory which is complex to implement in the hardware even for a single flow. It is even harder when the number of flows goes up to hundreds or thousands. However, implementing a sending window as we did in HPCC, which is just a sequential array per flow without random memory access, is straightforward and effective.

5 PERFORMANCE EVALUATION

In this section, we use testbed experiments with our prototype and large scale NS3 simulations [5] to evaluate the performance of HPCC and compare to existing alternatives.

5.1 Evaluation setup

Network topologies The testbed topology mimics a small scale RDMA PoD in our production. The testbed includes one Agg switch and four ToRs (ToR_1-ToR_4) connected via four 100Gbps links. There are 32 servers in total and each server has two 25Gbps NICs. 16 servers are connected to ToR_1 and ToR_2 via two uplinks, and the other 16 servers are connected to ToR_3 and ToR_4. The base RTT is 5.4μs within a rack and 8.5μs cross racks.

The topology in the NS3 simulations is a FatTree [7]. There are 16 Core switches, 20 Agg switches, 20 ToRs and 320 servers (16 in each rack), and each server has a single 100Gbps NIC connected to a single ToR. The capacity of each link between Core and Agg switches, Agg switches and ToRs are all 400Gbps. All links have a 1μs propagation delay, which gives a 12μs maximum base RTT. The switch buffer size is 32MB which is derived from real device configurations. The whole network is a single RDMA domain.

Traffic loads We use widely accepted and public available data center traffic traces, *WebSearch* [43] and *FB_Hadoop* [37] in both testbed experiments and simulations. We adjust the flow generation rates to set the average link loads to 30% and 50% respectively. We also create some simple artificial traffic loads to evaluate the micro-benchmarks of HPCC.

Alternative approaches We compare HPCC with DCQCN and TIMELY [31], which are CC schemes designed for RDMA. Since neither of them limits inflight bytes, we also try to improve them by adding a sending window (same as we use for HPCC), and we call the improved version "DCQCN+win" and "TIMELY+win". We also compare with DCTCP [8] which is a host-based TCP-like CC for high throughput and low latency in data center networks. We remove the slow start phase in DCTCP for fair comparisons.

Parameters For HPCC, we set W_{AI} = 80bytes[4], $maxStage$ = 5, and η = 95%[5] in Algorithm 1. We set T to 9μs for testbed and 13μs for simulations, which are slightly greater than the maximum RTT of the networks. For DCQCN, we use the parameters suggested by a major NIC vendor; For TIMELY, we use the parameters suggested in [31]. For DCQCN and DCTCP, we scale the ECN marking threshold proportional to the link bandwidth (Bw). For DCQCN, we set $K_{min} = 100KB \times \frac{Bw}{25Gbps}$ and $K_{max} = 400KB \times \frac{Bw}{25Gbps}$ according to our experiences (no vendor suggestion available). For DCTCP, we set $K_{min} = K_{max} = 30KB \times \frac{Bw}{10Gbps}$ according to [8]. We set the dynamic PFC threshold so that the PFC is triggered when an ingress queue consumes more than 11% of the free buffer.

Performance metrics We have five performance metrics. (i) FCT slow down; (ii) Real-time bandwidth of individual flows; (iii) network latency; (iv) PFC pause duration; (v) Size of in-network queues.

INT overhead For considering the impact of INT overhead on the performance, we assume each packet in HPCC has an additional 42 bytes in the header. This is a worst-case assumption because a data packet merely has 42 bytes INT meta-data at the last hop.

[4]Calculated based on 100 concurrent flows under 100Gbps according to §3.3.
[5]We tried $maxStage$ from 0 to 5, and η from 95% to 98%, all of which give similar results. Here we show the most conservative setting.

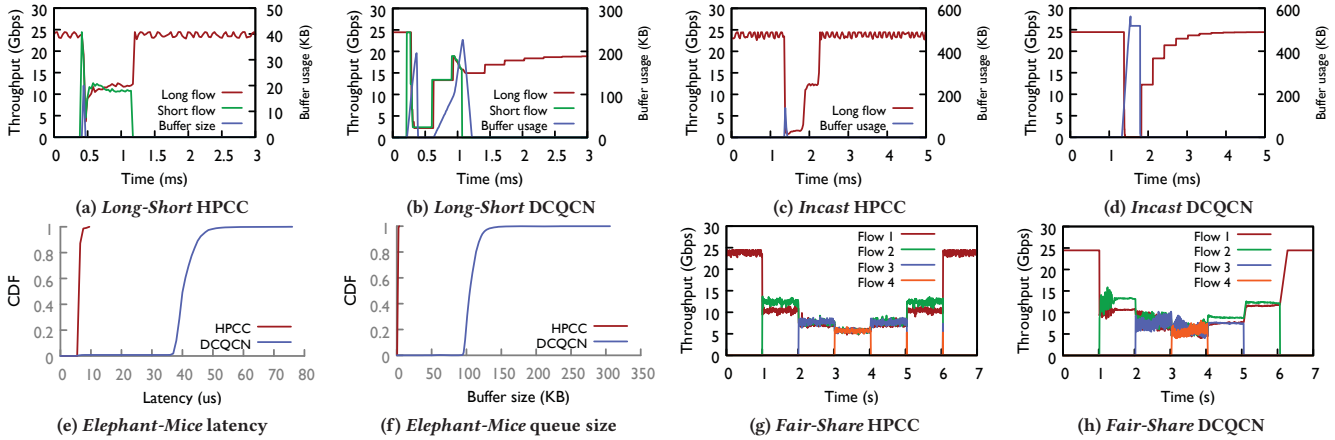


Figure 9: Comparing HPCC and DCQCN on testbed with four micro-benchmark traffic loads.

5.2 Testbed experiments

We run our prototype (§4) on the testbed. We compare HPCC with DCQCN with both micro-benchmarks and realistic traffic loads.

Micro-benchmarks

HPCC has faster and better rate recovery Figure 9a and 9b illustrate the behaviors of HPCC and DCQCN with *Long-Short* traffic. A long flow sends at full line rate, and later a short flow with 1MB size joins sharing the same links as the long flow and leaves after a period of time. HPCC recovers the rate of the long flow right after the short flow ends, while DCQCN cannot recover to line rate even after 2 ms (>350 RTTs). HPCC ramps up quickly because its feedback does not rely on the queue.

HPCC has faster and better congestion avoidance Figure 9c and 9d show how HPCC and DCQCN react to congestion caused by *Incast*. Seven senders start to send flows at the same time towards the receiver of a long-running flow. HPCC quickly reacts after just one round trip, so the queue drains quickly. With DCQCN, the queue builds up to 550 KB due to two reasons: (1) it waits for the queue build up for ECN, and (2) it does not limit the inflight bytes.

HPCC has lower network latency We keep sending mice flows (1KB each) through a link that is saturated by two elephant flows, and measure the mice flow latency and the buffer size. Figure 9e and 9f show HPCC keeps a near-zero queue and therefore the latency of mice flows is close to 5.4μs, the base RTT. DCQCN keeps a standing queue around the ECN marking threshold, so the latency is consistently higher than 35μs.

HPCC has fairness Figure 9g and 9h show the fairness of HPCC and DCQCN. 4 flows join a link one by one every second and leave afterwards. HPCC provides good fairness even in a short time scale.

End-to-end performance We evaluate HPCC and DCQCN under *WebSearch*, at 30% and 50% loads. We also run similar experiments under *FB_Hadoop*, which show similar trends.

HPCC significantly reduces FCT for short flows HPCC and DCQCN achieve similar FCT slowdowns in the median, but at 95th and 99th percentile, HPCC achieves a much better FCT slowdown especially for short flows (Figure 10a and 10c). For example, at 30% load, HPCC reduces the 99th-percentile FCT slowdown from 11.2 down to 2.38 for flows shorter than 3KB, which is only 16.9μs. The gap is larger with higher loads. For example, at 50% load, HPCC achieves a 95% reduction on the 99th-percentile FCT slowdown, from 53.9 down to 2.70, for the flows shorter than 3KB, which is only 19.2μs.

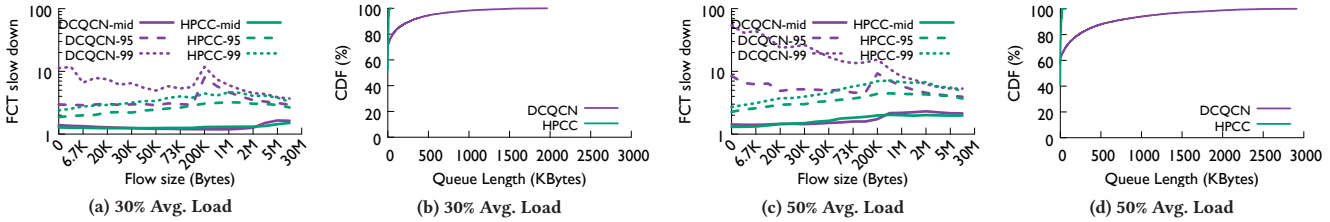
HPCC has steadily close-to-zero queues Figure 10b and 10d show the CDF of queue lengths at switches, which provides more insight into the achieved performance. In both cases, the median queue size is 0, which explains the closeness of median FCT slow down. However, HPCC keeps the ultra-low queue size even at the very tail, thus achieving much lower FCTs for short flows. For example, at 50% load, HPCC's 95 and 99 percentile queue sizes are 19.7KB and 22.9KB, whereas DCQCN's sizes are 1.1MB and 2.1 MB. These experiments confirm that HPCC is very effective at keeping the queue near zero under realistic traffic patterns. As a result, there is no packet loss and PFC is not necessary because the queue size never reaches the PFC threshold.

5.3 Large-scale event-driven simulations

We verified the fidelity of simulation by performing the same experiment as the testbed, which matches testbed results well. We then use simulations to evaluate HPCC on a larger network topology and higher line rates. Figure 11 shows the comparison of HPCC and other CC schemes for *FB_Hadoop* traffic. To stress test with diverse traffic patterns, we either add incast traffic to 30% load traffic or run 50% load traffic. We generate the incast traffic by randomly selecting 60 senders and one receiver, each sending 500KB. The incast traffic load is 2% of the network capacity. We also tried various levels of traffic load, incast sizes and incast ratios, as well as with *WebSearch*, all of which show similar trends. Here are the key observations:

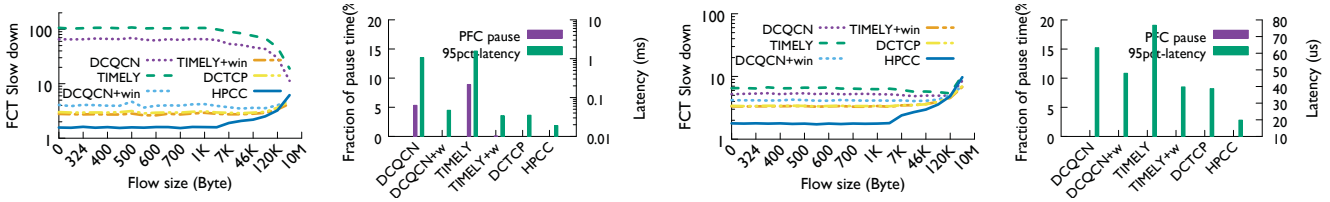
HPCC is beneficial to short flows. Since HPCC keeps near-zero queues and resolves congestion quickly, it is beneficial to short flows. All other CC schemes maintain standing queues, so they cannot keep the latency low. Figure 11a and 11c show that for the flows shorter than 120KB, HPCC achieves much lower FCTs than all the other schemes at 95th percentile. This is beneficial to applications with many short flows. This is the case for *FB_Hadoop*, where 90% of the flows are shorter than 120KB.

Figure 11b and 11d show that on the tail, HPCC still achieves very low round trip latency, under 20μs. For example, the 95-percentile



(a) 30% Avg. Load (b) 30% Avg. Load (c) 50% Avg. Load (d) 50% Avg. Load

Figure 10: FCT slow down and queue size of HPCC and DCQCN in testbed with *WebSearch* (30% and 50% avg. load).



(a) 30% Avg. Load + incast (b) 30% Avg. Load + incast (c) 50% Avg. Load (d) 50% Avg. Load

Figure 11: FCT slow down at 95-percentile, PFC and latency with *FB_Hadoop* (30% avg. load + 60-to-1 incast and 50% avg. load).

latency at 50% load is 19.8μs, which is less than 8μs extra latency compared to the 12μs base RTT.

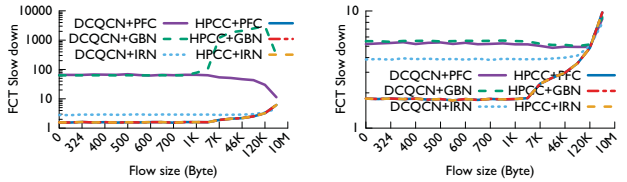
DCTCP outperforms DCQCN and TIMELY because it controls the queue better as its window limits the inflight bytes. This may seem to contradict previous statements about their relative performance[31, 43]. However, this discrepancy is mainly because they compare hardware DCQCN/kernel-bypassing TIMELY with DCTCP based on kernel[31, 43], which is known to have huge performance cost and requires higher ECN threshold, while we only simulate the effect of CC, excluding the cost introduced by the software nature for a fair comparison. That said, HPCC reduces DCTCP's latency by more than 2 times, which is significant in data centers.

Throughput for long flows. Since HPCC explicitly controls the bottleneck links to have a 5% bandwidth headroom, and the INT header consumes extra bandwidth, the long flow has a higher slowdown as shown in Figure 11a and 11c. The slowdown increases with a higher load as the theory in [9] shows. The reason is that the long flow slowdown is inversely proportional to the residual capacity of the network. Other CC schemes aim to fully utilize the bandwidth, so their residual capacity is $(100\% - load)$ which is 44.6% at 50% load (including header and ACK); for HPCC, the residual capacity is $(95\% - load \times (1 + INT_overhead))$ which is 36.1%. So at 50% load the long flows are 1.24 times slower with HPCC than with other schemes, which matches the FCT quite well. This is a fundamental tradeoff we have to make in favor of short flows.

CC is the key to achieve stability and high performance. As Figure 11b shows, large scale PFC pauses only appear when using DCQCN and TIMELY, which confirms our insight that CC is the key to the stability problem. Specifically, controlling the inflight bytes is the key: just adding a sending window to DCQCN and TIMELY reduces PFCs to almost zero.

We further show that a good CC scheme lessens the importance of the flow control choices. We use PFC, go-back-N retransmission, and IRN [32][6], in combination with DCQCN and HPCC, and

[6]When using go-back-N or IRN, where packet losses are not prevented by PFC, we set the dynamic threshold for the egress queues with $\alpha = 1$, which allows a single congested egress port to consume up to half of the buffer.



(a) 30% Avg. Load + incast (b) 50% Avg. Load

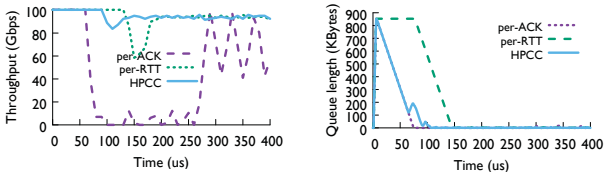
Figure 12: FCT slow down at 95-percentile, with different flow control choices. GBN stands for go-back-N.

perform the same experiment. Figure 12 shows that with HPCC, different flow control schemes do not affect the performance. On the other hand, since DCQCN controls the queue poorly, a better flow control does improve its performance (it is worth noting that, IRN adds a fixed sized window which also limits the inflight bytes, and thus has improvement over the other two schemes). But even with IRN, DCQCN still cannot match HPCC's performance, which confirms that CC is the key problem.

5.4 Design choices

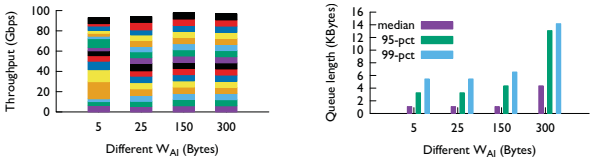
We use a simple 16 to 1 incast scenario to show the design choices of HPCC. The 16 senders and 1 receiver are connected via 100Gbps links through a single switch, with 1μs link propagation delay.

HPCC achieves fast reaction without overreaction. We illustrate the benefit of HPCC's strategy of combining per-ACK and per-RTT reactions. Figure 13 shows the time series of aggregate throughput and queue lengths. The queue builds up at the beginning of all flows. Since the first few ACKs of each flow already see the long queue, per-ACK reaction reacts to the queue quickly. However, it incurs a significant overreaction, so the aggregate throughput soon drops to almost 0 and then oscillates. Per-RTT reaction reacts to the queue slowly (only after all the ACKs in the first round are received) and wastes the information brought by the first few ACKs. As a result, the long queue persists for a long time. HPCC introduces a reference rate that is updated every RTT and reacts to every ACK based on the reference rate, so HPCC achieves fast reaction without overreaction.



(a) Total throughput. (b) Queue length.

Figure 13: Different ways of reacting to ACK.



(a) Total throughput. (b) Queue length.

Figure 14: Fairness and queue size with W_{AI}.

Tuning W_{AI} for HPCC Figure 14a shows the first 10ms of the throughput of different flows. Figure 14b shows the queue length distribution, sampled every 1μs.

With 16 concurrent flows, W_{AI} should not exceed 100 Gbps×4μs $\times(1-0.95)/16 \approx$ 150bytes (4μs is the base RTT), which is confirmed by the result. Specifically, all W_{AI} within 150bytes achieve a queue length within 4 KB at 95-th percentile, while W_{AI}=300bytes has a queue length of 13 KB at 95-th percentile (Figure 14b). Within the 150 bytes range, a higher W_{AI} has better fairness. That said, since we need to prepare for the worst case, we set W_{AI} to sustain 100 concurrent flows (25 bytes in this case) that has good fairness.

It is worth noting that, HPCC's performance degrades gracefully with a high W_{AI}. In the 300 bytes case, the queue length is still very low in general: a 13KB queue just means 1 μs queueing delay.

6 RELATED WORK

CC is an enduring research topic. Here we try to cover several related works which are closely related.

RDMA CC TIMELY [31] is a pioneer in RDMA CC, which uses RTT as a congestion signal. §5 shows that TIMELY suffers from incast congestion because it gradually adjusts its rate. Furthermore, it can converge to much longer queues than DCQCN [44]. iWarp [35] is an alternative to RoCEv2. It puts the full TCP stack into hardware NIC. As a result, iWarp suffers from well-known TCP problems in data centers [40], such as high latency and vulnerability to incast. Furthermore, due to the complexity to implement TCP stack in hardware, iWarp NICs in general have a higher cost [19].

General CC for data center networks DCTCP [8] and TCP Bolt [8] are two solutions implemented in host software which suffers from high CPU overhead and high latency, while implementing them in hardware raise problems similar to iWarp. In addition, because they both use ECN similar to DCQCN, they can hardly achieve small queues.

There are several proposals aiming to reduce latency with changes in both host software and switch hardware. pFabric [10] needs to run sophisticated priority scheduling logics in switches and to correctly prioritize traffic in hosts, which are hard to deploy [22]; HULL [9] advocates leaving a bandwidth headroom for ultra-low latency in data centers, which is similar to HPCC. However, it requires non-trivial implementations on switches for Phantom queues to get feedback before link saturation; DeTail [42] needs a new switch architecture for lossless fabric and performs per-packet adaptive load balancing of packet routes in switches; HOMA [33] and NDP [22] are receiver-driven, credit-based solutions, which is a big shift from the state-of-the-art in practice, since they have complex receivers. They, together with PIAS [11] also require priority queues, while the number of priority queues is limited in switches, and production networks often have to reserve them for application QoS. Different from these solutions, one HPCC's design goal is ease of implementation in hardware for offloading and deployability with the start-of-the-art commodity NICs and switches. In addition, HPCC works with a single priority queue.

There are also explicit CCs such as XCP [27] and RCP [16]. Besides the key difference discussed in § 3.4, both of them require switches to perform computation which is not widely available in most commodity switches, but INT is widely available (§ 2.4). Moreover, because HPCC allows line rate start, new flows can finish faster than with slow start (XCP) and processor sharing (RCP). The line rate start is allowed by our control on inflight bytes, which drains queues rapidly. HPCC's decoupling of utilization and fairness is inspired by XCP.

The stability of CC algorithms has been investigated by several authors using a variety of simplified models [12, 27, 28, 41]. We draw two main insights from previous work. Firstly the speed of adaptation to new observations of congestion should be scaled to round-trip times in order to avoid destabilizing oscillations. Secondly, while feedback based on queue size is important for dealing with sudden overloads, it is not particularly helpful for steady state stability when queueing delays are short compared with round-trip times (Appendix A.1). So we aim to keep link utilizations less than 100% to keep steady state queueing delays very short.

Flow controls for RDMA IRN [32] and MELO [30] are recent proposals to reduce hardware-based selective packet re-transmissions to prevent PFC pauses or even replace PFC. These efforts are orthogonal and complementary with HPCC. Different from the fixed window used in IRN, the sending window in HPCC is proportional to flow's sending rate with better network stability.

7 CONCLUSION

We share our production experiences on the difficulties to operate RDMA networks with the state-of-the-art high-speed CC. We propose HPCC as a next-generation CC for high-speed networks to achieve ultra-low latency, high bandwidth, and stability simultaneously. HPCC achieves fast convergence, small queues, and fairness by leveraging precise load information from INT. It has been implemented with commodity programmable NICs and switches and shows remarkable gains. We believe HPCC is a start towards CC for future hyper-speed networks.

This work does not raise any ethical issues.

8 ACKNOWLEDGEMENT

We thank our shepherd Nandita Dukkipati and SIGCOMM reviewers for their helpful feedback. Yuliang Li and Minlan Yu are supported in part by the NSF grant CNS-1618138. Mohammad Alizadeh is supported by NSF grants CNS-1751009 and CNS-1617702.

REFERENCES

[1] 2011. IEEE. 802.11Qbb. Priority based flow control. (2011).
[2] 2019. In-band Network Telemetry in Barefoot Tofino. https://www.opencompute.org/files/INT-In-Band-Network-Telemetry-A-Powerful-Analytics-Framework-for-your-Data-Center-OCP-Final3.pdf. (2019).
[3] 2019. In-band Network Telemetry in Broadcom Tomahawk 3. https://www.broadcom.com/company/news/product-releases/2372840. (2019).
[4] 2019. In-band Network Telemetry in Broadcom Trident3. https://www.broadcom.com/blog/new-trident-3-switch-delivers-smarter-programmability-for-enterprise-and-service-provider-datacenters. (2019).
[5] 2019. Network Simulator 3. (2019). https://www.nsnam.org/.
[6] 2019. Switch Implementation with P4. (2019). https://github.com/p4lang/switch/blob/master/p4src/switch.p4.
[7] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM, 2008*.
[8] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data Center TCP (DCTCP). In *SIGCOMM, 2010*.
[9] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less is more: trading a little bandwidth for ultra-low latency in the data center. In *NSDI, 2012*.
[10] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pFabric: Minimal near-optimal datacenter transport. In *SIGCOMM, 2013*.
[11] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. Information-agnostic flow scheduling for commodity data centers. In *NSDI, 2015*.
[12] Hamsa Balakrishnan, Nandita Dukkipati, Nick McKeown, and Claire J Tomlin. 2007. Stability analysis of explicit congestion control protocols. *IEEE Communications Letters* 11, 10 (2007), 823–825.
[13] Thomas Bonald, Laurent Massoulié, Alexandre Proutière, and Jorma Virtamo. 2006. A queueing analysis of max-min fairness, proportional fairness and balanced fairness. *Queueing Systems* 53, 1-2 (2006), 65–84.
[14] Inho Cho, Keon Jang, and Dongsu Han. Credit-Scheduled Delay-Bounded Congestion Control for Datacenters. In *SIGCOMM, 2017*.
[15] Donald Davies. 1972. The control of congestion in packet-switching networks. *IEEE Transactions on Communications* 20, 3 (1972), 546–550.
[16] Nandita Dukkipati. 2007. *Rate Control Protocol (RCP): Congestion control to make flows complete quickly*. Stanford University.
[17] Peter Xiang Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network Requirements for Resource Disaggregation. In *OSDI, 2016*.
[18] Peter Xiang Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. pHost: Distributed Near-optimal Datacenter Transport over Commodity Network Fabric. In *CoNEXT, 2015*.
[19] Dániel Géhberger, Dávid Balla, Markosz Maliosz, and Csaba Simon. Performance evaluation of low latency communication alternatives in a containerized cloud environment. In *IEEE CLOUD, 2018*.
[20] Prateesh Goyal, Anup Agarwal, Ravi Netravali, Mohammad Alizadeh, and Hari Balakrishnan. 2019. ABC: A Simple Explicit Congestion Control Protocol for Wireless Networks. *arXiv:1905.03429* (2019).
[21] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over commodity ethernet at scale. In *SIGCOMM, 2016*.
[22] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *SIGCOMM, 2017*.
[23] Shuihai Hu, Yibo Zhu, Peng Cheng, Chuanxiong Guo, Kun Tan, Jitendra Padhye, and Kai Chen. Tagger: Practical PFC Deadlock Prevention in Data Center Networks. In *CoNEXT, 2017*.
[24] Van Jacobson. Congestion avoidance and control. In *SIGCOMM, 1988*.
[25] Lavanya Jose, Stephen Ibanez, Mohammad Alizadeh, and Nick McKeown. A Distributed Algorithm to Calculate Max-Min Fair Rates Without Per-Flow State. In *SIGMETRICS, 2019*.
[26] Glenn Judd. Attaining the Promise and Avoiding the Pitfalls of TCP in the Datacenter. In *NSDI, 2015*.
[27] Dina Katabi, Mark Handley, and Charlie Rohrs. Congestion control for high bandwidth-delay product networks. In *SIGCOMM, 2002*.
[28] Frank Kelly, Gaurav Raina, and Thomas Voice. 2008. Stability and fairness of explicit congestion control with small buffers. *ACM SIGCOMM Computer Communication Review* 38, 3 (2008), 51–62.
[29] Frank Kelly and Elena Yudovina. 2014. *Stochastic networks*. Vol. 2. Cambridge University Press.
[30] Yuanwei Lu, Guo Chen, Zhenyuan Ruan, Wencong Xiao, Bojie Li, Jiansong Zhang, Yongqiang Xiong, Peng Cheng, and Enhong Chen. Memory efficient loss recovery for hardware-based transport in datacenter. In *APNET, 2017*.
[31] Radhika Mittal, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, David Zats, et al. TIMELY: RTT-based Congestion Control for the Datacenter. In *SIGCOMM, 2015*.
[32] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. Revisiting Network Support for RDMA. In *SIGCOMM, 2018*.
[33] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A Receiver-driven Low-latency Transport Protocol Using Network Priorities. In *SIGCOMM, 2018*.
[34] Ilkka Norros, James W. Roberts, Alain Simonian, and Jorma T. Virtamo. 1991. The superposition of variable bit rate sources in an ATM multiplexer. *IEEE Journal on Selected Areas in Communications* 9, 3 (1991), 378–387.
[35] Renato Recio, Bernard Metzler, Paul Culley, Jeff Hilland, and Dave Garcia. 2007. *A remote direct memory access protocol specification (RFC5040)*. Technical Report.
[36] James W. Roberts (Ed.). 1992. *Performance Evaluation and Design of Multiservice Networks*. Commission of the European Communities.
[37] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the Social Network's (Datacenter) Network. In *SIGCOMM, 2015*.
[38] Alex Shpiner, Eitan Zahavi, Vladimir Zdornov, Tal Anker, and Matty Kadosh. Unlocking credit loop deadlocks. In *HotNets, 2016*.
[39] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *SIGCOMM, 2015*.
[40] Vijay Vasudevan, Amar Phanishayee, Hiral Shah, Elie Krevat, David G Andersen, Gregory R Ganger, Garth A Gibson, and Brian Mueller. Safe and effective fine-grained TCP retransmissions for datacenter communication. In *SIGCOMM, 2009*.
[41] Thomas Voice and Gaurav Raina. 2009. Stability analysis of a max-min fair Rate Control Protocol (RCP) in a small buffer regime. *IEEE Trans. Automat. Control* 54, 8 (2009), 1908–1913.
[42] David Zats, Tathagata Das, Prashanth Mohan, Dhruba Borthakur, and Randy Katz. DeTail: reducing the flow completion time tail in datacenter networks. In *SIGCOMM, 2012*.
[43] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion Control for Large-Scale RDMA Deployments. In *SIGCOMM, 2015*.
[44] Yibo Zhu, Monia Ghobadi, Vishal Misra, and Jitendra Padhye. ECN or Delay: Lessons Learnt from Analysis of DCQCN and TIMELY. In *CoNEXT, 2016*.

APPENDIX

Appendices are supporting material that has not been peer reviewed.

A THE ANALYSIS OF HPCC'S THEORETICAL PROPERTIES

A.1 Bounding queueing delays

To achieve ultra-low latency at the queues within the network we control the windows of sources to keep the utilization of resources around a target utilization of less than 100%, and each source paces packet transmissions so that the time between two packets entering the network is the reciprocal of the source's pacing rate.

With a fixed number of long flows the natural model for a queue at a resource is then a $\sum D_i/D/1$ queue, a deterministic server with an arrival process that is a heterogeneous mix of periodic sources. This model has been analyzed extensively [34, 36]. A superposition of homogeneous streams yields the greatest buffer requirement for a given load, and a Brownian bridge approximation is accurate in heavy traffic. As an example of the numerical results, if the load is 95% of capacity and there are 50 sources then the mean number of packets in the queues is about 3 and the probability there are more than 20 packets in the queue is about 10^{-9}. Even if the load is 100% the mean number of packets in the queue is only about $(\pi N/8)^{0.5}$ where N is the number of streams (and thus less than 5 with 50 sources; note that since sources are periodic, the queue remains stable even at 100% load).

In practice the arrival process at a resource is likely to be more variable than a heterogeneous mix of periodic sources, even with just a fixed number of long flows. Nonetheless we expect little queueing unless the load on the queue exceeds 100%. Conversely, if more than a small number of packets are queued then it is almost certainly because the load on the resource exceeds its capacity.

The stability analysis of XCP and RCP [12] aims for 100% utilization, and requires feedback based on queue size, as well as rate mismatch, to shrink an otherwise persistent queue. This complicates the stability analysis, which is available for only a single congested resource. When the target utilization is less than 100%, feedback based on queue size is not particularly helpful for steady state stability; and the stability analysis for systems using just feedback based on rate mismatch can be simplified and extended to a network with multiple resources and RTTs [28].

In HPCC, measurement of queue size is essential for dealing rapidly with transient overloads. New sources are allowed to start transmitting at line rate (to allow short flows to finish quickly), and this may cause queue lengths to grow rapidly especially for incasts. The interaction of window limits with the queueing term *qlen* in the algorithm is designed to drain queues rapidly (Appendix A.4).

A.2 Fast convergence of utilization

Next we consider how quickly loads on the resources can be brought back under control following a perturbation (perhaps caused by a new source or sources starting).

We begin with a very simple discrete time model, where sources all share the same RTT and rates at sources are updated synchronously once per RTT. First note that if there is a single bottleneck resource then we could achieve the target utilization (η) in one RTT with the update $R(t+RTT) = R(t)(U_{target}/U)$ where U is the observed utilization (it is simpler to work with rates rather than windows in the analysis since rates and capacity constraints have the same units and fairness is traditionally defined with respect to rates).

Now suppose there are resources $i = 1, 2, \cdots, I$ and paths $j = 1, 2, \cdots, J$. Let A be the incidence matrix defined by $A_{ij} = 1$ if resource i is used by path j and $A_{ij} = 0$ otherwise; assume each path uses at least one resource, so that no column of A is identically zero. Let $C_i > 0$ be the (target) capacity of resource i, for $i = 1, 2, \cdots, I$, and define the vector $C = (C_i, i = 1, 2, \cdots, I)$. A rate allocation is a vector $R = (R_j, j = 1, 2, \cdots, J)$. Let Y_i be the load on resource i and let $Y = (Y_i, i = 1, 2, \cdots, I)$. From the definition of the matrix A we have that $Y = AR$. Say that R is *feasible* if the vector inequality $Y \le C$ is satisfied, so that the load on each resource is not greater than the (target) capacity of the resource.

Suppose the initial state $R(0)$ has $R_j(0) > 0$ for $j = 1, 2, \cdots, J$ and suppose rates are updated in discrete time by the recursions

$$Y(n) = AR(n) \tag{5}$$

$$R_j(n+1) = \frac{R_j(n)}{\max_i\{Y_i(n)A_{ij}/C_i\}}. \tag{6}$$

Lemma

(i) $Y(n) \le C$ for $n = 1, 2, \cdots$; hence after one step rates are all feasible.

(ii) $R(n+1) \ge R(n)$ for $n = 1, 2, \cdots$; hence after the first step rates are either constant or increase.

(iii) $R(n) = R$ for $n = I, I+1, \cdots$ where R is Pareto optimal; hence after at most I steps $R(n)$ is constant and is then Pareto optimal.

Proof

$$Y_i(n+1) = \sum_j A_{ij}\frac{R_j(n)}{\max_k\{Y_k(n)A_{kj}/C_k\}} \le \sum_j A_{ij}\frac{R_j(n)}{\{Y_i(n)/C_i\}} = C_i,$$

and so rates are feasible for $n \ge 1$. Also $\max\{Y_i(n)/C_i\} \le 1$ for $n = 1, 2, \cdots$, and hence after the first step rates are non-decreasing. Furthermore if $k = \text{argmax}\{Y_i(0)/C_i\}$ then after one time step $Y_k(1) = C_k$ - the resource k is *bottlenecked*. Thereafter the rates on paths through the bottlenecked resource k remain unchanged. We can remove resource k from the network, together with all paths through it. At each subsequent step at least one more resource becomes bottlenecked and can be removed. After at most I steps either all resources are bottlenecked or all paths have been removed. At the resulting rate allocation R all paths pass through at least one bottleneck, and so no path can have its rate increased without decreasing the rate of another path: hence the rate allocation R is Pareto optimal.

The recursions (5)-(6) thus give convergence to feasibility after just one RTT and fast convergence to a Pareto optimal allocation $R(n) = R$. However the allocation will not in general be fair, and indeed R will in general depend on the initial state $R(0)$. Next we consider how an additive increase term encourages convergence to a form of fairness.

A.3 Additive increase and fairness

Consider a network with multiple resources where RTTs vary and updates are asynchronous. For a given source let U_i be the utilization at resource i observed by the source and let $U = \max_i\{U_i\}$ where the maximum is over the resources on the path associated with the source. Suppose the rate at the source is updated once per RTT by

$$R(t+RTT) = R(t)\frac{U_{target}}{U(t+RTT)} + a$$

where $a > 0$ is a small additive increase. (In this sub-section we use R for the rate from a typical source, rather than the vector giving the rates over all sources.) Then at an equilibrium point (where $U(t) = U$ and $R(t) = R$ do not vary with time for any of the sources) we have that

$$U = \max_i\{U_i\}, \quad R = a\left(1 - \frac{U_{target}}{U}\right)^{-1}.$$

Let $U_{(1)}$ be the equilibrium utilization at the most congested bottleneck (i.e. the resource for which U_i is highest), and let $R_{(1)}$ be the equilibrium rate on paths through this bottleneck. Let $U_{(2)}$ be the equilibrium utilization at the next most congested bottleneck, and let $R_{(2)}$ be the equilibrium rate on paths which pass through this as their most congested bottleneck. Similarly define $U_{(i)}, R_{(i)}$ for $i = 3, 4, \cdots$. Thus $R_{(1)} \le R_{(2)} \le \cdots$. Then, by recursion of the above analysis,

$$R_{(i)} = a\left(1 - \frac{U_{target}}{U_{(i)}}\right)^{-1}.$$

Thus

$$U_{(i)} = U_{target}\left(1 - \frac{a}{R_{(i)}}\right)^{-1}$$

confirming that $U_{(1)} \geq U_{(2)} \geq \cdots$. Observe that the equilibrium utilizations are above U_{target} by an amount that increases with a. The highest utilization $U_{(1)}$ will be less than 100% if $a < R_{(1)}(1 - U_{target})$. For example, if $U_{target} = 95\%$ then a should be less than 5% of the flow rate $R_{(1)}$.

There is a trade-off in the choice of the additive increase a: smaller values of a will produce smaller fluctuations about the equilibrium, but at the cost of slower convergence to the equilbrium. If a is small enough the rates will be approximately max-min fair.

Next consider the impact of stochastic fluctuations in the observed utilizations and hence in the rates. The distribution of $\max_i\{U_i\}$ will depend mainly on the most congested link but will be biased upwards by other congested links on the path. The rates achieved along different paths will be biased away from max-min fairness towards proportional fairness. This is not of itself a major problem: relative to max-min fairness, rate allocations under proportional fairness give an improvement of utilization across multiple resources (since the absolute priority max-min fairness gives to smaller flows can cause starvation at some resources [13]).

With more registers at sources we can exercise more control over the form of fairness achieved, as we now briefly describe. Suppose a source maintains a distinct register R_i for each resource on its path, updated by

$$R_i(t + RTT) = R_i(t)\frac{U_{target}}{U_i(t + RTT)} + a$$

where $U_i(t + RTT)$ is the utilization at resource i observed by the source over the preceding RTT and again $a > 0$ is a small additive increase. Update the rate R of the source by

$$R = \left(\sum_i R_i^{-\alpha}\right)^{-1/\alpha} \tag{7}$$

where $\alpha \in (0, \infty)$ is a fixed parameter.

At an equilibrium point

$$R_i = a\left(1 - \frac{U_{target}}{U_i}\right)^{-1}$$

and R is given by equation (7). Here we interpret R_i as the rate which would be allocated to a source whose path went through just one resource, resource i, and then rate R is the α-fair rate allocation. Note that as $\alpha \to \infty$ the expression (7) approaches $\min_i\{R_i\}$ corresponding to max-min fairness. The case $\alpha = 1$ corresponds to proportional fairness. The case $\alpha \to 0$ approaches the rate allocation which maximizes the sum of the rates over all sources [29].

A.4 Window limits

Since the earliest days of packet-switching the importance of controlling the number of inflight packets traversing the network has been understood [15] and congestion control in TCP makes explicit use of window flow control [24]. In our algorithm we also limit the number of inflight packets a source has. We illustrate the benefit in this sub-section.

We suppose that a source limits its transmissions so that at any time the source has no more packets unacknowledged than its window limit. If an update indicates congestion, then the window limit decreases, so the source may be restricted from transmitting if acknowledgements are slow in returning to the source.

As an example, suppose a new source starts transmitting at line rate. Then it can continue to do so for the first RTT: if after this acknowledgements start returning at line rate then the window limit will not restrict the source. So if a new source transmitting at line rate does not observe congestion, then it can continue at line rate.

As a second example, suppose a set of 64 new sources begin transmitting together, each at line rate, and that the paths used by the sources converge on an intree to a single root queue which was already busy at its target utilization. Further suppose the 64 new sources continue to transmit at line rate for one base RTT - their initial burst. This is a very stressful case: the queueing time at the root of the intree will build up to nearly 64 times the base RTT of the new sources by the time the last packets of the initial bursts arrive at the root queue. But after the first acknowledgements start arriving at the new sources their windows will decrease rapidly, since these acknowledgements carry early news of the queue building up at the root queue. Consequently very few packets will be sent by the source following its initial burst until the receipt of the last packet from its initial burst, by which time it will decrease its window to about 1/65 of its initial window (note that the RTT time of the last packet from the initial burst is about 65 times the base RTT, and that this packet has observed approximately the peak queue at the root of the intree). Thus the window limits on sources allows the queue at the root of the intree to empty as fast as is possible, and the queueing term *qlen* in the algorithm forces new sources to moderate their windows following receipt of the delayed acknowledgements from the initial burst. The rates of the new sources and the rates of the existing flows through the root queue will not yet be fair - that will take longer, as a consequence of the additive increase term in the algorithm. But utilization has been brought under control as quickly as possible in a very stressful case without triggering PFC.