Congestion Control for Large-Scale RDMA Deployments

Yibo Zhu[1,3] Haggai Eran[2] Daniel Firestone[1] Chuanxiong Guo[1] Marina Lipshteyn[1]
Yehonatan Liron[2] Jitendra Padhye[1] Shachar Raindel[2] Mohamad Haj Yahia[2] Ming Zhang[1]

[1]Microsoft [2]Mellanox [3]U. C. Santa Barbara

ABSTRACT

Modern datacenter applications demand high throughput (40Gbps) and ultra-low latency (< 10 μs per hop) from the network, with low CPU overhead. Standard TCP/IP stacks cannot meet these requirements, but Remote Direct Memory Access (RDMA) can. On IP-routed datacenter networks, RDMA is deployed using RoCEv2 protocol, which relies on Priority-based Flow Control (PFC) to enable a drop-free network. However, PFC can lead to poor application performance due to problems like head-of-line blocking and unfairness. To alleviates these problems, we introduce DCQCN, an end-to-end congestion control scheme for RoCEv2. To optimize DCQCN performance, we build a fluid model, and provide guidelines for tuning switch buffer thresholds, and other protocol parameters. Using a 3-tier Clos network testbed, we show that DCQCN dramatically improves throughput and fairness of RoCEv2 RDMA traffic. DCQCN is implemented in Mellanox NICs, and is being deployed in Microsoft's datacenters.

CCS Concepts

•**Networks** → **Transport protocols;**

Keywords

Datacenter transport; RDMA; PFC; ECN; congestion control

1. INTRODUCTION

Datacenter applications like cloud storage [16] need high bandwidth (40Gbps or more) to meet rising customer demand. Traditional TCP/IP stacks cannot be used at such speeds, since they have very high CPU overhead [29]. The brutal economics of cloud services business dictates that CPU usage that cannot be monetized should be minimized: a core spent on supporting high TCP throughput is a core that cannot be sold as a VM. Other applications such as distributed memory caches [10, 30] and large-scale machine learning demand ultra-low latency (less than 10 μs per hop) message transfers. Traditional TCP/IP stacks have far higher latency [10].

We are deploying Remote Direct Memory Access (RDMA) technology in Microsoft's datacenters to provide ultra-low latency and high throughput to applications, with very low CPU overhead. With RDMA, network interface cards (NICs) transfer data in and out of pre-registered memory buffers at both end hosts. The networking protocol is implemented entirely on the NICs, bypassing the host networking stack. The bypass significantly reduces CPU overhead and overall latency. To simplify design and implementation, the protocol assumes a lossless networking fabric.

While the HPC community has long used RDMA in special-purpose clusters [11, 24, 26, 32, 38], deploying RDMA on a large scale in modern, IP-routed datacenter networks presents a number of challenges. One key challenge is the need for a congestion control protocol that can operate efficiently in a high-speed, lossless environment, and that can be implemented on the NIC.

We have developed a protocol, called Datacenter QCN (DCQCN) for this purpose. DCQCN builds upon the congestion control components defined in the RoCEv2 standard. DCQCN is implemented in Mellanox NICs, and is currently being deployed in Microsoft's datacenters.

To understand the need for DCQCN, it is useful to point out that historically, RDMA was deployed using InfiniBand (IB) [19, 21] technology. IB uses a custom networking stack, and purpose-built hardware. The IB link layer (L2) uses hop-by-hop, credit-based flow control to prevent packet drops due to buffer overflow. The lossless L2 allows the IB transport protocol (L4) to be simple and highly efficient. Much of the IB protocol stack is implemented on the NIC. IB supports RDMA with so-called single-sided operations, in which a server registers a memory buffer with its NIC, and clients read (write) from (to) it, without further involvement of the server's CPU.

However, the IB networking stack cannot be easily de-

SIGCOMM '15, August 17–21, 2015, London, United Kingdom
 ISBN 978-1-4503-3542-3/15/08…$15.00
DOI: http://dx.doi.org/10.1145/2785956.2787484

ployed in modern datacenters. Modern datacenters are built with IP and Ethernet technologies, and the IB stack is incompatible with these. DC operators are reluctant to deploy and manage two separate networks within the same datacenter. Thus, to enable RDMA over Ethernet and IP networks, the RDMA over Converged Ethernet (RoCE) [20] standard, and its successor RoCEv2 [22] have been defined. RoCEv2 retains the IB transport layer, but replaces IB networking layer (L3) with IP and UDP encapsulation, and replaces IB L2 with Ethernet. The IP header is needed for routing, while the UDP header is needed for ECMP [15].

To enable efficient operation, like IB, RoCEv2 must also be deployed over a lossless L2. To this end, RoCE is deployed using Priority-based Flow Control (PFC) [18]. PFC allows an Ethernet switch to avoid buffer overflow by forcing the immediate upstream entity (either another switch or a host NIC) to pause data transmission. However, PFC is a coarse-grained mechanism. It operates at port (or, port plus priority) level, and does not distinguish between flows. This can cause congestion-spreading, leading to poor performance [1, 37].

The fundamental solution to PFC's limitations is a flow-level congestion control protocol. In our environment, the protocol must meet the following requirements: (i) function over lossless, L3 routed, datacenter networks, (ii) incur low CPU overhead on end hosts, and (iii) provide hyper-fast start in the common case of no congestion. Current proposals for congestion control in DC networks do not meet all our requirements. For example, QCN [17] does not support L3 networks. DCTCP [2] and iWarp [35] include a slow start phase, which can result in poor performance for bursty storage workloads. DCTCP and TCP-Bolt [37] are implemented in software, and can have high CPU overhead.

Since none of the current proposals meet all our requirements, we have designed DCQCN. DCQCN is an end-to-end congestion control protocol for RoCEv2, to enable deployment of RDMA in large, IP-routed datacenter networks. DCQCN requires only the standard RED [13] and ECN [34] support from the datacenter switches. The rest of the protocol functionality is implemented on the end host NICs. DCQCN provides fast convergence to fairness, achieves high link utilization, ensures low queue buildup, and low queue oscillations.

The paper is organized as follows. In §2 we present evidence to justify the need for DCQCN. The detailed design of DCQCN is presented in §3, along with a brief summary of hardware implementation. In §4 we show how to set the PFC and ECN buffer thresholds to ensure correct operation of DCQCN. In §5 we describe a fluid model of DCQCN, and use it to tune protocol parameters. In §6, we evaluate the performance of DCQCN using a 3-tier testbed and traces from our datacenters. Our evaluation shows that DCQCN dramatically improves throughput and fairness of RoCEv2 RDMA traffic. In some scenarios, it allows us to handle as much as 16x more user traffic. Finally, in §7, we discuss practical issues such as non-congestion packet losses.

2. THE NEED FOR DCQCN

To justify the need for DCQCN, we will show that TCP stacks cannot provide high bandwidth with low CPU overhead and ultra-low latency, while RDMA over RoCEv2 can. Next, we will show that PFC can hurt performance of RoCEv2. Finally, we will argue that existing solutions to cure PFC's ills are not suitable for our needs.

2.1 Conventional TCP stacks perform poorly

We now compare throughput, CPU overhead and latency of RoCEv2 and conventional TCP stacks. These experiments use two machines (Intel Xeon E5-2660 2.2GHz, 16 core, 128GB RAM, 40Gbps NICs, Windows Server 2012R2) connected via a 40Gbps switch.

Throughput and CPU utilization: To measure TCP throughput, we use Iperf [46] customized for our environment. Specifically, we enable LSO [47], RSS [49], and zero-copy operations and use 16 threads. To measure RDMA throughput, we use a custom tool that uses IB READ operation to transfer data. With RDMA, a single thread saturates the link.

Figure 1(a) shows that TCP has high CPU overhead. For example, with 4MB message size, to drive full throughput, TCP consumes, on average, over 20% CPU cycles across all cores. At smaller message sizes, TCP cannot saturate the link as CPU becomes the bottleneck. Marinos et.al. [29] have reported similarly poor TCP performance for Linux and FreeBSD. Even the user-level stack they propose consumes over 20% CPU cycles. In contrast, the CPU utilization of the RDMA client is under 3%, even for small message sizes. The RDMA server, as expected, consumes almost no CPU cycles.

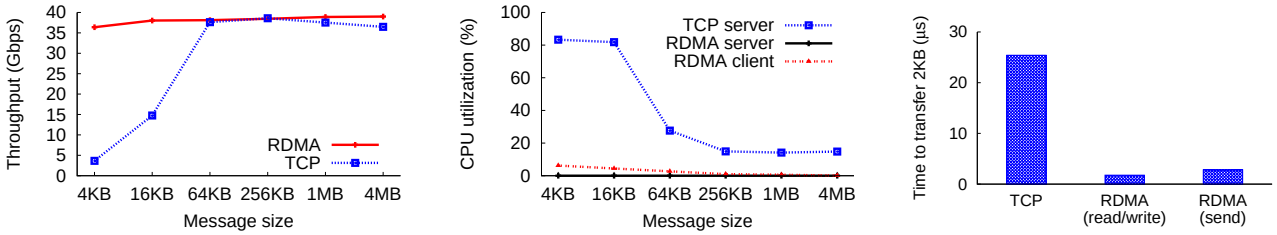
Latency: Latency is the key metric for small transfers. We now compare the average user-level latency of transferring a 2K message, using TCP and RDMA. To minimize TCP latency, the connections were pre-established and warmed, and Nagle was disabled. Latency was measured using a high resolution ($\leq 1\ \mu s$) timer [48]. There was no other traffic on the network.

Figure 1(c) shows that TCP latency (25.4 μs) is significantly higher than RDMA (1.7 μs for Read/Write and 2.8 μs for Send). Similar TCP latency has reported in [10] for Windows, and in [27] for Linux.

2.2 PFC has limitations

RoCEv2 needs PFC to enable a drop-free Ethernet fabric. PFC prevents buffer overflow on Ethernet switches and NICs. The switches and NICs track ingress queues. When the queue exceeds a certain threshold, a PAUSE message is sent to the upstream entity. The uplink entity then stops sending on that link till it gets an RESUME message. PFC specifies upto eight priority classes. PAUSE/RESUME messages specify the priority class they apply to.

The problem is that the PAUSE mechanism operates on a per port (and priority) basis – not on a per-flow basis. This can lead to head-of-line blocking problems; resulting in poor performance for individual flows. We now illustrate



(a) Mean Throughput (b) Mean CPU Utilization (c) Mean Latency

Figure 1: Throughput, CPU consumption and latency of TCP and RDMA

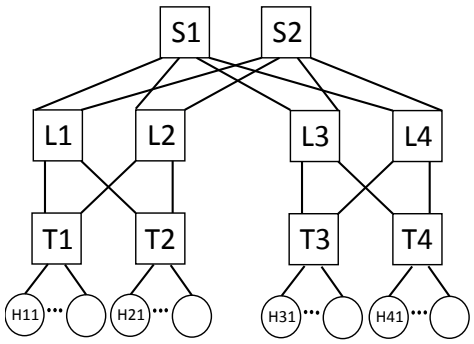
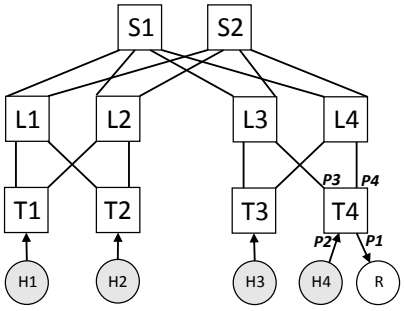
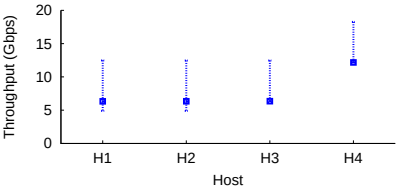


Figure 2: Testbed topology. All links are 40Gbps. All switches are Arista 7050QX32. There are four ToRs (T1-T4), four leaves (L1-L4) and two spines (S1-S2). Each ToR represents a different IP subnet. Routing and ECMP is done via BGP. Servers have multiple cores, large RAMs, and 40Gbps NICs.

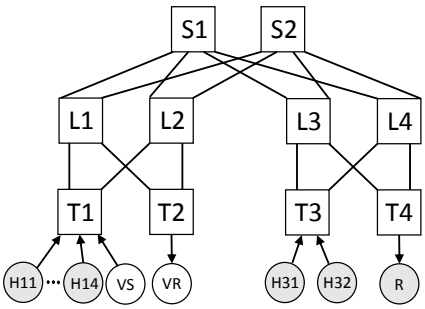


(a) Topology

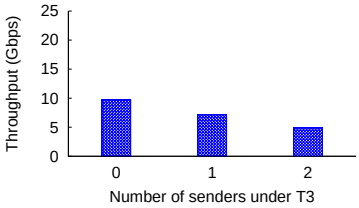


(b) Throughput of individual senders

Figure 3: PFC Unfairness



(a) Topology



(b) Median throughput of victim flow

Figure 4: Victim flow problem

the problems using a 3-tier testbed (Figure 2) representative of modern datacenter networks.

Unfairness: Consider Figure 3(a). Four senders (H1-H4) send data to the single receiver (R) using RDMA WRITE operation. All senders use the same priority class. Ideally, the four senders should equally share bottleneck link (T4 to R). However, with PFC, there is unfairness. When queue starts building up on T4, it pauses incoming links (ports P2-P4). However, P2 carries just one flow (from H4), while P3 and P4 may carry multiple flows since H1, H2 and H3 must share these two ports, depending on how ECMP maps the flows. Thus, H4 receives higher throughput than H1-H3. This is known as the parking lot problem [14].

This is shown in Figure 3(b), which shows the min, median and max throughput achieved by H1-H4, measured over 1000 4MB data transfers. H4 gets as much as 20Gbps throughput, e.g. when ECMP maps all of H1-H3 to either P3 or P4. H4's minimum throughput is higher than the maximum throughput of H1-H3.

Victim flow: Because PAUSE frames can have a cascading effect, a flow can be hurt by congestion that is not even on its path. Consider Figure 4(a). Four senders (H11-H14), send data to R. In addition, we have a "victim flow" – VS sending to VR. Figure 4(b) shows the median throughput (250 transfers of 250MB each) of the victim flow.

When there are no senders under T3, in the median case (two of H11-H14 map to T1-L1, others to T1-L2. Each of H11-H14 gets 10Gbps throughput. VS maps to one of T1's uplinks), one might expect VS to get 20Gbps throughput. However, we see that it only gets 10Gbps. This is due to cascading PAUSEs. As T4 is the bottleneck of H11-H14 incast, it ends up PAUSEing its incoming links. This in turn leads to L3 and L4 to pause their incoming links, and so forth. Eventually, L1 and L2 end up pausing T1's uplinks to them, and T1 is forced to PAUSE the senders. The flows on T1 that use these uplinks are equally affected by these PAUSEs, regardless of their destinations – this is also known as the head-of-the-line blocking problem.

The problem gets worse as we start senders H31 and H32 that also send to R. We see that the median throughput further falls from 10Gbps to 4.5Gbps, even though no path from H31 and H32 to R has any links in common with the path between VS and VR. This happens because H31 and H32 compete with H11-H14 on L3 and L4, make them PAUSE S1 and S2 longer, and eventually make T1 PAUSE senders longer.

Summary: These experiments show that flows in RoCEv2 deployments may see lower throughput and/or high variability due to PFC's congestion-spreading characteristics.

2.3 Existing proposals are inadequate

A number of proposals have tried to address PFC's limitations. Some have argued that ECMP can mitigate the problem by spreading traffic on multiple links. Experiments in previous section show that this is not always the case. The PFC standard itself includes a notion of priorities to address the head-of-the-line blocking problem. However, the standard supports only 8 priority classes, and both scenarios shown above can be made arbitrarily worse by expanding the topology and adding more senders. Moreover, flows within the same class will still suffer from PFC's limitations.

The fundamental solution to the PFC's problems is to use flow-level congestion control. If appropriate congestion control is applied on a per-flow basis, PFC will be rarely triggered, and thus the problems described earlier in this section will be avoided.

The Quantized Congestion Notification (QCN) [17] standard was defined for this purpose. QCN enables flow-level congestion control within an L2 domain. Flows are defined using source/destination MAC address and a flow id field. A switch computes a congestion metric upon each packet arrival. Its value depends on the difference between the instantaneous queue size and the desired equilibrium queue size, along with other factors. The switch then probabilistically (probability depends on the severity of the congestion) sends the quantized value of the congestion metric as feedback to the source of the arriving packet. The source reduces its sending rate in response to congestion feedback. Since no feedback is sent if there is no congestion, the sender increases its sending rate using internal timers and counters.

QCN cannot be used in IP-routed networks because the definition of a flow is based entirely on L2 addresses. In IP-routed networks the original Ethernet header is not preserved as the packet travels through the network. Thus a congested switch cannot determine the target to send the congestion feedback to.

We considered extending the QCN protocol to IP-routed networks. However, this is not trivial to implement. At minimum, extending QCN to IP-routed networks requires using the IP five-tuple as flow identifier, and adding IP and UDP headers to the congestion notification packet to enable it to reach the right destination. Implementing this requires hardware changes to both the NICs and the switches. Making changes to the switches is especially problematic, as the QCN functionality is deeply integrated into the ASICs. It usually takes months, if not years for ASIC vendors to implement, validate and release a new switch ASIC. Thus, updating the chip design was not an option for us.

In §8 we will discuss why other proposals such as TCP-Bolt [37] and iWarp [35] do not meet our needs. Since the existing proposals are not adequate, for our purpose, we propose DCQCN.

3. THE DCQCN ALGORITHM

DCQCN is a rate-based, end-to-end congestion protocol, that builds upon QCN [17] and DCTCP [2]. Most of the DCQCN functionality is implemented in the NICs.

As mentioned earlier, we had three core requirements for DCQCN: (i) ability to function over lossless, L3 routed, datacenter networks, (ii) low CPU overhead and (iii) hyper-fast start in the common case of no congestion. In addition, we also want DCQCN to provide fast convergence to fair bandwidth allocation, avoid oscillations around the stable point, maintain low queue length, and ensure high link utilization.

There were also some practical concerns: we could not demand any custom functionality from the switches, and since the protocol is implemented in NIC, we had to be mindful of implementation overhead and complexity.

The DCQCN algorithm consists of the sender (reaction point (RP)), the switch (congestion point (CP)), and the receiver, (notification point (NP)).

3.1 Algorithm

CP Algorithm: The CP algorithm is same as DCTCP. At an egress queue, an arriving packet is ECN [34]-marked if the queue length exceeds a threshold. This is accomplished using RED [13] functionality (Figure 5) supported on all modern switches. To mimic DCTCP, we can set $K_{min} = K_{max} = K$, and $P_{max} = 1$. Later, we will see that this is not the optimal setting.

NP Algorithm: ECN-marked packets arriving at NP indicate congestion in the network. NP conveys this information back to the sender. The RoCEv2 standard defines explicit Congestion Notification Packets (CNP) [19] for this purpose. The NP algorithm specifies how and when CNPs should be generated.

The algorithm follows the state machine in Figure 6 for each flow. If a marked packet arrives for a flow, and no CNP has been sent for the flow in last N microseconds, a CNP is sent immediately. Then, the NIC generates at most one CNP packet every N microseconds for the flow, if any packet that arrives within that time window was marked. We use $N = 50\mu$s in our deployment. Processing a marked packet, and generating the CNP are expensive operations, so we minimize the activity for each marked packet. We discuss the implications in §5.

RP Algorithm: When an RP (i.e. the flow sender) gets a CNP, it reduces its current rate (R_C) and updates the value of the rate reduction factor, α, like DCTCP, and remembers current rate as target rate (R_T) for later recovery. The values are updated as follows:[1]

$$\begin{aligned} R_T &= R_C, \\ R_C &= R_C(1-\alpha/2), \\ \alpha &= (1-g)\alpha + g, \end{aligned} \tag{1}$$

The NP generates no feedback if it does not get any marked packets. Thus, if RP gets no feedback for K time units, it

[1]Initial value of α is 1.

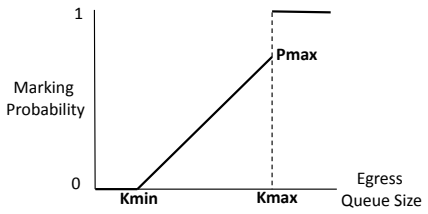


Figure 5: Switch packet marking algorithm

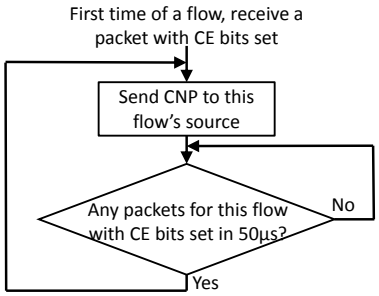


Figure 6: NP state machine

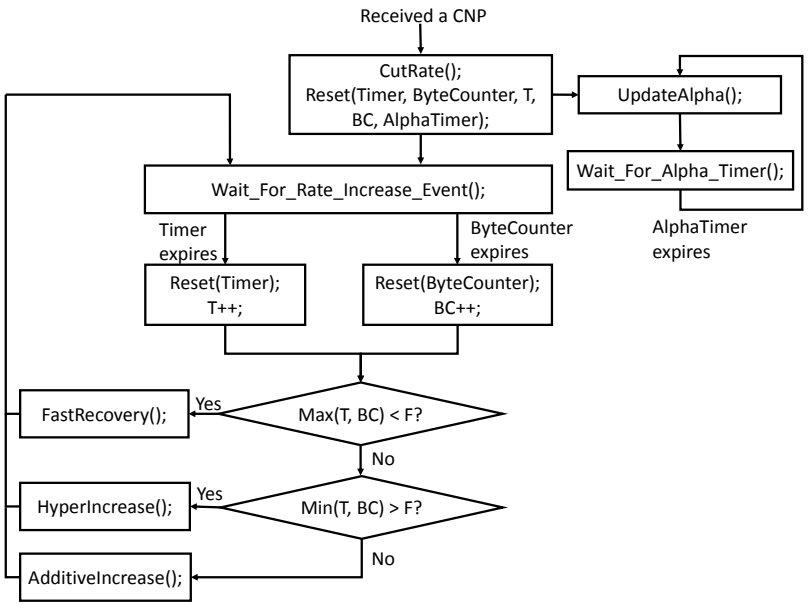


Figure 7: Pseudocode of the RP algorithm

updates α, as shown in Equation (2). Note that K must be larger than the CNP generation timer. Our implementation uses $K = 55\mu$s. See §5 for further discussion.

$$\alpha = \quad (1-g)\alpha, \tag{2}$$

Furthermore, RP increases its sending rate using a timer and a byte counter, in a manner identical to QCN [17]. The byte counter increases rate for every B bytes, while the timer increases rate every T time units. The timer ensures that the flow can recover quickly even when its rate has dropped to a low value. The two parameters can be tuned to achieve the desired aggressiveness. The rate increase has two main phases: fast recovery, where the rate is rapidly increased towards fixed target rate for $F = 5$ successive iterations:

$$R_C = \quad (R_T + R_C)/2, \tag{3}$$

Fast recovery is followed by an additive increase, where the current rate slowly approaches the target rate, and target rate is increased in fixed steps R_{AI}:

$$\begin{aligned} R_T &= \quad R_T + R_{AI}, \\ R_C &= \quad (R_T + R_C)/2, \end{aligned} \tag{4}$$

There is also a hyper increase phase for fast ramp up. Figure 7 shows the state machine. See [17] for more details.

Note that there is no slow start phase. When a flow starts, it sends at full line rate, if there are no other active flows from the host.[2] This design decision optimizes the common case where flows transfer a relatively small amount of data, and the network is not congested [25].

3.2 Benefits

By providing per-flow congestion control, DCQCN alleviates PFC's limitations. To illustrate this, we repeat the experiments in §2.2, with DCQCN enabled (parameters set according to guidelines in §4 and §5.

[2]Otherwise, starting rate is defined by local QoS policies.

Figure 8 shows that DCQCN solves the unfairness problem depicted in Figure 3. All four flows get equal share of the bottleneck bandwidth, and there is little variance. Figure 9 shows that DCQCN solves the victim flow problem depicted in Figure 4. With DCQCN, the throughput of VS-VR flow does not change as we add senders under T3.

3.3 Discussion

CNP generation: DCQCN is not particularly sensitive to congestion on the reverse path, as the send rate does not depend on accurate RTT estimation like TIMELY [31]. Still, we send CNPs with high priority, to avoid missing the CNP deadline, and to enable faster convergence. Note that no CNPs are generated in the common case of no congestion.

Rate based congestion control: DCQCN is a rate-based congestion control scheme. We adopted a rate-based approach because it was simple to implement than the window based approach, and allowed for finer-grained control.

Parameters: DCQCN is based on DCTCP and QCN, but it differs from each in key respects. For example, unlike QCN, there is no quantized feedback, and unlike DCTCP there is no "per-ack" feedback. Thus, the parameter settings recommended for DCTCP and QCN cannot be blindly used with DCQCN. In §5, we use a fluid model of the DCQCN to establish the optimal parameter settings.

The need for PFC: DCQCN does not obviate the need for PFC. With DCQCN, flows start at line rate. Without PFC, this can lead to packet loss and poor performance (§6).

Hardware implementation: The NP and RP state machines are implemented on the NIC. The RP state machine requires keeping one timer and one counter for each flow that is being rate limited, apart from a small amount of other state such as the current value of alpha. This state is maintained on the NIC die. The rate limiting is on a per-packet

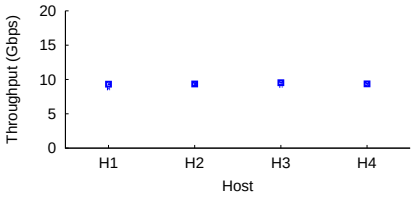


Figure 8: Throughput of individual senders with DCQCN. Compare to Figure 3(b).

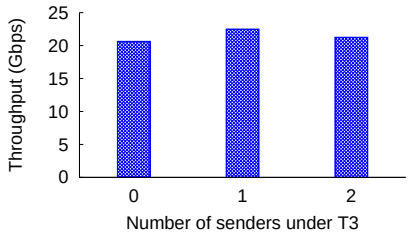


Figure 9: Median throughput of "victim" flow with DCQCN. Compare to Figure 4(b).

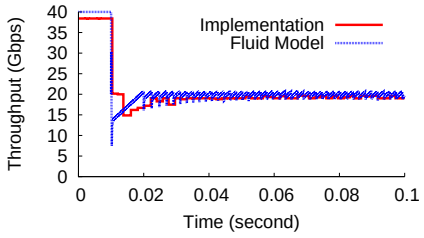


Figure 10: Fluid model closely matches implementation.

granularity. The implementation of NP state machine in ConnectX-3 Pro can generate CNPs at the rate of one per 1-5 microseconds. At link rate of 40Gbps, the receiver can receive about 166 full-sized (1500 byte MTU) packets every 50 microseconds. Thus, the NP can typically support CNP generation at required rate for 10-20 congested flows. The current version(ConnectX-4) can generate CNPs at the required rate for over 200 flows.

4. BUFFER SETTINGS

Correct operation of DCQCN requires balancing two conflicting requirements: (i) PFC is not triggered too early – i.e. before giving ECN a chance to send congestion feedback, and (ii) PFC is not triggered too late – thereby causing packet loss due to buffer overflow.

We now calculate the values of three key switch parameters: t_{flight}, t_{PFC} and t_{ECN}, to ensure that these two requirements are met even in the *worst case*. Note that different switch vendors use different terms for these settings; we use generic names. The discussion is relevant to any shared-buffer switch, but the calculations are specific to switches like Arista 7050QX32, that use the Broadcom Trident II chipset. These switches have 32 full duplex 40Gbps ports, 12MB of shared buffer and support 8 PFC priorities.

Headroom buffer t_{flight}: A PAUSE message sent to an upstream device takes some time to arrive and take effect. To avoid packet drops, the PAUSE sender must reserve enough buffer to process any packets it may receive during this time. This includes packets that were in flight when the PAUSE was sent, and the packets sent by the upstream device while it is processing the PAUSE message. The worst-case calculations must consider several additional factors (e.g., a switch cannot abandon a packet transmission it has begun) [8]. Following guidelines in [8], and assuming a 1500 byte MTU, we get $t_{flight} = 22.4KB$ per port, per priority.

PFC Threshold t_{PFC}: This is the maximum size an ingress queue can grow to, before a PAUSE message is sent to the upstream device. Each PFC priority gets its own queue at each ingress port. Thus, if the total switch buffer is B, and there are n ports, it follows that $t_{PFC} \leq (B - 8nt_{flight})/(8n)$. For our switches, we get $t_{PFC} \leq 24.47KB$. The switch sends RESUME message when the queue falls below t_{PFC} by two MTU.

ECN Threshold t_{ECN}: Once an egress queue exceeds this threshold, the switch starts marking packets on that queue (K_{min} in Figure 5). For DCQCN to be effective, this threshold must be such that PFC threshold is not reached before the switch has a chance to mark packets with ECN.

Note however, that ECN marking is done on *egress* queue while PAUSE messages are sent based on *ingress* queue. Thus, t_{ECN} is an egress queue threshold, while t_{PFC} is an ingress queue threshold.

The worst case scenario is that packets pending on *all* egress queues come from a *single* ingress queue. To *guarantee* that PFC is not triggered on this ingress queue before ECN is triggered on *any* of the egress queues, we need: $t_{PFC} > n * t_{ECN}$ Using the upper bound on the value of t_{PFC}, we get $t_{ECN} < 0.85KB$. This is less than one MTU and hence infeasible.

However, not only we do not have to use the upper bound on t_{PFC}, we do not even have to use a fixed value for t_{PFC}. Since the switch buffer is shared among all ports, t_{PFC} should depend on how much of the shared buffer is free. Intuitively, if the buffer is largely empty, we can afford to wait longer to trigger PAUSE. The Trident II chipset in our switch allows us to configure a parameter β such that: $t_{PFC} = \beta(B - 8nt_{flight} - s)/8$, where s is the amount of buffer that is currently occupied. A higher β triggers PFC less often, while a lower value triggers PFC more aggressively. Note that s is equal to the sum of packets pending on all egress queues. Thus, just before ECN is triggered on any egress port, we have: $s \leq n * t_{ECN}$. Hence, to ensure that ECN is always triggered before PFC, we set: $t_{ECN} < \beta(B - 8nt_{flight})/(8n(\beta + 1))$. Obviously, larger β leaves more room for t_{ECN}. In our testbed, we use $\beta = 8$, which leads to $t_{ECN} < 21.75KB$.

Discussion: The above analysis is conservative, and ensures that PFC is not triggered on our switches before ECN even in the worst case and when all 8 PFC priorities are used. With fewer priorities, or with larger switch buffers, the threshold values will be different.

The analysis *does not* imply that PFC will *never* be triggered. All we ensure is that at any switch, PFC is not triggered *before* ECN. It takes some time for the senders to receive the ECN feedback and reduce their sending rate. During this time, PFC may be triggered. As discussed before, we rely on PFC to allow senders to start at line rate.

Variable	Description
R_c	Current Rate
R_t	Target Rate
α	See Equation (1)
q	Queue Size
t	Time

Table 1: Fluid model variables

Parameter	Description
$K_{min}, K_{max}, P_{max}$	See Figure 5
g	See Equation (1)
N	Number of flows at bottleneck
C	Bandwidth of bottleneck link
F	Fast recovery steps (fixed at 5)
B	Byte counter for rate increase
T	Timer for rate increase
R_{AI}	Rate increase step (fixed at 40Mbps)
$\tau*$	Control loop delay
τ'	Interval of Equation (2)

Table 2: Fluid model parameters

5. ANALYSIS OF DCQCN

We use a fluid model of DCQCN to determine the right parameter settings for good performance.

5.1 Fluid model

Variables and parameters used in the fluid model are listed in Tables 1 and 2, respectively. The model is described in Equations (5)-(9). Like [3, 4], we model N greedy flows at a single bottleneck with capacity C. We assume that DCQCN is triggered well before PFC – and thus ignore PFC in the following analysis.

Equation (5) models the probability of a packet getting marked at the bottleneck. Setting $K_{\min} == K_{\max}$, gives DCTCP-like "cut-off"behavior. We model the more general behavior for reasons discussed later. Equation (6) models the evolution of queue at the bottleneck. We have assumed that all flows have equal rate. We will relax this assumption later. Equation (7) models the evolution of α at the RP.

Equations (8) and (9) model the evolution of current and target sending rate at RP, according to the algorithm shown in Figure 7, and QCN specification [17]. We model the rate decrease, as well as the rate increase due to byte counter and timer, but like [4], ignore the hyper additive increase phase. $\tau*$ models the delay of the control loop. It includes the RTT and NP's CNP generation interval. For simplicity, we use $\tau* = 50\mu s$ (the maximum possible delay of CNP generation).

By setting the LHS of Equations (6)-(9) to zero, we see that the fluid model has an unique fixed point when satisfying Equation (10).

$$R_C(t) = \frac{C}{N} \tag{10}$$

This fixed point represents the state of all flows getting fair bandwidth share C/N. The rest of the fluid model becomes three equations with three variables, R_T, α and p. We solve the equations numerically to determine the ECN marking probability p at the fixed point. The solution of p is unique.

$$p(t) = \begin{cases} 0, & q(t) \le K_{\min} \\ \frac{q(t)-K_{\min}}{K_{\max}-K_{\min}} p_{\max}, & K_{\min} < q(t) \le K_{\max} \\ 1, & q(t) > K_{\max} \end{cases} \tag{5}$$

$$\frac{dq}{dt} = NR_C(t) - C \tag{6}$$

$$\frac{d\alpha}{dt} = \frac{g}{\tau'}\left(\left(1-(1-p(t-\tau*))^{\tau' R_C(t-\tau*)}\right) - \alpha(t)\right) \tag{7}$$

$$\begin{aligned}\frac{dR_T}{dt} =& -\frac{R_T(t)-R_C(t)}{\tau}\left(1-(1-p(t-\tau*))^{\tau R_C(t-\tau*)}\right) \\ &+ R_{AI}R_C(t-\tau*)\frac{(1-p(t-\tau*))^{FB}p(t-\tau*)}{(1-p(t-\tau*))^{-B}-1} \\ &+ R_{AI}R_C(t-\tau*)\frac{(1-p(t-\tau*))^{FTR_C(t-\tau*)}p(t-\tau*)}{(1-p(t-\tau*))^{-TR_C(t-\tau*)}-1}\end{aligned} \tag{8}$$

$$\begin{aligned}\frac{dR_C}{dt} =& -\frac{R_C(t)\alpha(t)}{2\tau}\left(1-(1-p(t-\tau*))^{\tau R_C(t-\tau*)}\right) \\ &+ \frac{R_T(t)-R_C(t)}{2}\frac{R_C(t-\tau*)p(t-\tau*)}{(1-p(t-\tau*))^{-B}-1} \\ &+ \frac{R_T(t)-R_C(t)}{2}\frac{R_C(t-\tau*)p(t-\tau*)}{(1-p(t-\tau*))^{-TR_C(t-\tau*)}-1}\end{aligned} \tag{9}$$

Fluid Model of DCQCN

We omit the proof for brevity. We verified that for reasonable settings, p is less than 1%. According to equation (5), when RED-ECN is enabled, there exists a fixed queue length point close to $K_{\min}$ since p is close to 0. The value of g plays an important role in determining the stability of the queue, as we shall later see.

To analyze convergence properties of DCQCN protocol, it is necessary to extend the fluid model to flows with different rates. Take two flows as an example. We model the evolution of each flow's current and target sending rates, as well as their α separately – i.e. we write Equations (7)-(9) for each flow. The flows are coupled by their impact on the queue:

$$\frac{dq}{dt} = R_{C1}(t) + R_{C2}(t) - C \tag{11}$$

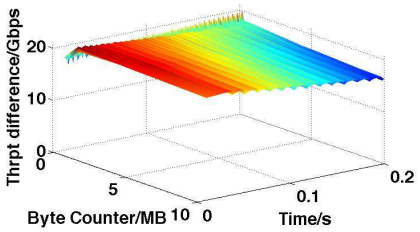
We solve this model numerically to understand the impact of various parameters (Table 2) on DCQCN's behavior; particularly convergence and queue buildup.

Experiments show that the fluid model matches the implementation quite well. We show one example in Figure 10. We defer discussion to §6.

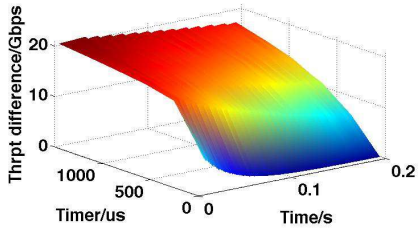
5.2 Parameter selection

We focus on a two-flow system, where one flow starts at 40Gbps, and the other starts at 0Gbps. For each parameter setting, we use numerical analysis to solve the first 200 milliseconds. The results are shown in Figure 11. The Z-axis shows difference in throughput of the two flows. For readability, we omit the first 10ms.

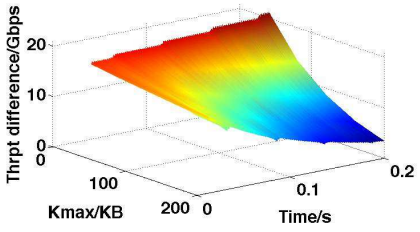
We start with recommended parameters from QCN and DCTCP specifications. Specifically, $B = 150KB, T = 1.5ms, K_{\min} = K_{\max} = 40KB, P_{\max} = 1$, and $g = 1/16$.



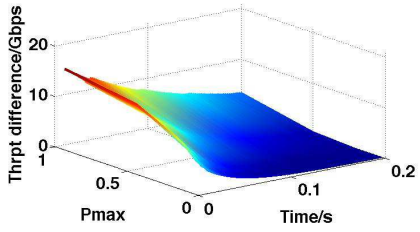
(a) Sweep Byte Counter with strawman parameters



(b) Sweep Timer with 10MB Byte Counter

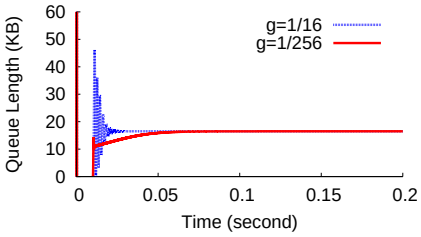


(c) Sweep $K_{\max}$ with strawman parameters

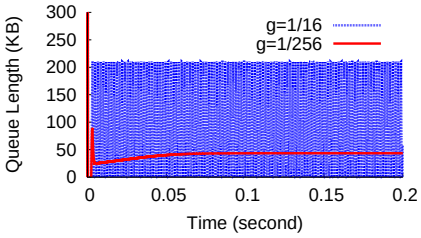


(d) Sweep $P_{\max}$ with $K_{\max} = 200KB$

Figure 11: Parameter sweeping for best convergence. Lower throughput difference (z-axis) means better convergence.



(a) 2:1 Incast



(b) 16:1 Incast

Figure 12: Testing different g for best queue length and queue stability.

Unfortunately we find that with these parameter values, the flows cannot converge (innermost edge of Figure 11(a)). With these settings, QCN byte counter dominates rate increase, and the faster flow grows faster. QCN compensates for this using probabilistic feedback, which DCQCN does not have. Instead, the solution is to either slow down the byte counter, or speed up the rate increase timer. Figure 11(a) shows that slowing down byte counter helps, but it reduces convergence speed. Speeding up the timer (Figure 11(b)) is better. Note that the timer cannot be smaller than 50μs, which is NP's CNP generation interval. As the timer is set aggressively, the byte counter should be set large, *e.g.,* 10MB, to avoid triggering the hyper increase phase too fast, which may harm convergence.

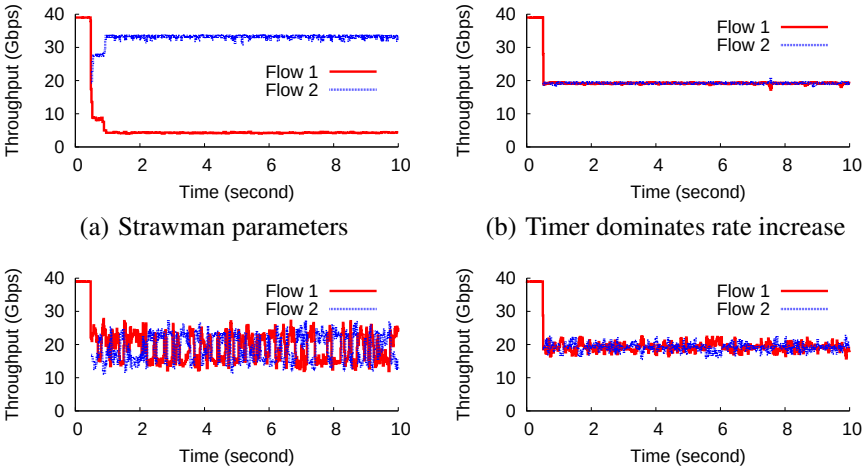
Another solution is to use a RED-like probabilistic packet marking scheme with a small $P_{\max}$, instead of the cut-off (mark all packets once queue length exceeds certain limit) behavior used in DCTCP. The intuition is that DCQCN CNP generation is driven by a timer, and by using RED-like marking scheme with a small $P_{\max}$, we increase the likelihood that the larger flow will get more CNPs, and hence back off faster. Figures 11(c) and 11(d) confirm this intuition.

In conclusion, to achieve fast convergence to fairness, we configure RED-ECN on switches with $K_{\max} = 200KB$ and $P_{\max} = 1\%$. We set $K_{\min} = 5KB$ since we found it sufficient to maintain 100% throughput. Though a 5KB $K_{\min}$ seems shallow, the marking probability around $K_{\min}$ is very little. Fluid model predicts that the stable queue length is usually one order of magnitude larger than 5KB $K_{\min}$. Our deployment experience and experiments (see §6) confirm this. We also set rate increase timer as 55us, and byte counter to 10MB. If the feedback delay is significantly different from the value we assumed (e.g. if RTT is high), parameter values will be different.

The queue length at the bottleneck depends on the value of g. We tested different g from 2:1 incast to 16:1 incast (Figure 12) and found smaller g leads to lower queue length and lower variation. Lower g leads to slightly slower convergence, but it is a price worth paying for lower oscillations.

We now briefly discuss the remaining parameters. Recall that CNP generation interval is fixed at 50μs, due to hardware limitations. We have assumed that feedback delay ($\tau*$) is equal to this value. We have verified that flows converge fast and stably even when an additional 50μs latency is added. We have chosen the α update interval τ' and minimum timer value to be 55μs. These values need to be larger than CNP generation interval to prevent unwarranted rate increases between reception of successive CNPs. We believe that the 5μs margin is sufficient for current hardware. DCQCN is generally insensitive to τ' value: 110μs (twice as default) barely slows down convergence. We also analyzed impact of R_{AI} and F. They both offer trade-off between convergence speed and throughput oscillations. In addition, R_{AI}, working with g, influences DCQCN scalability. For example, in current settings, there is no buffer starvation with 16:1 incast (Figure 12). Halving R_{AI} reduces the convergence speed, but it ensures no buffer starvation with 32:1 incast. The current values are an acceptable compromise.

Discussion The DCQCN model is based on QCN model presented in [4], with two differences. First, the rate reduction in DCQCN is significantly different. Our model also addresses both byte counter and timer mechanisms for rate increase, while [4] only models the byte counter. In future,



(a) Strawman parameters (b) Timer dominates rate increase

(c) Enable RED-ECN (d) RED-ECN plus timer solution

Figure 13: Validating parameter values with testbed microbenchmarks

Parameter	Value
Timer	55μs
Byte Counter	10MB
$K_{\max}$	200KB
$K_{\min}$	5KB
$P_{\max}$	1%
g	1/256

Figure 14: DCQCN Parameters used in our datacenters

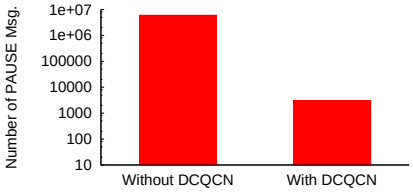


Figure 15: Total number of PAUSE messages received at S1 and S2, with and without DCQCN

we plan to analyze the stability of DCQCN following techniques in [4].

6. RESULTS

We now evaluate the performance of DCQCN in a variety of settings, using Mellanox ConnectX-3 Pro NICs. The results are divided in two parts. First, we validate the findings of the fluid model from §5 using microbenchmarks in a small testbed. This step is essential before we can use the parameter settings derived from the model in actual deployment. Next, we evaluate DCQCN using benchmark traffic derived from traffic traces that roughly model the ongoing DCQCN deployment in our datacenters.

6.1 Microbenchmarks

Validating fluid model: The first step is to show that the fluid model is a good representation of the implementation. We use a simple setup consisting of three machines, connected via an Arista 7050QX32 switch. One of the three machines act as a receiver, while the other two act as greedy senders. The first sender starts at time 0, while the second starts after 10 milliseconds. DCQCN parameters are set according to the discussion in §5, and the switch is configured as per the discussion in §4. Figure 10 shows the evolution of the sending rate of the second sender for the first 100 milliseconds. We see that the model closely matches the firmware implementation.

Validating parameter settings: While we have shown that the fluid model is a good representation of our implementation, we still need to validate the individual parameter values predicted by the model with real hardware. We conduct several tests for this purpose, using the same 3-machine setup as before. In each test, the first sender starts at time 0, and the second sender starts 500 milliseconds later.

We first test the strawman parameter values. Figure 13(a) confirms that as the model predicts, there is unfairness between the two competing flows.

Next we verify that speeding up the rate increase timer alleviates the unfairness. Figure 13(b) shows that this is indeed the case (timer value = 55μs). Note that we used DCTCP-like cut-off ECN for this experiment.

We also test the second solution – i.e. RED-like marking alleviates the unfairness problem, even without changing the timer. To test, we configure the switch with $K_{\min} = 5KB$, $K_{\max} = 200KB$ and $P_{\max} = 1\%$. Figure 13(c) shows that the two flows, on average get similar throughput. However, we see a drawback that isn't shown by fluid model: the throughput is unstable due to the randomness in marking.

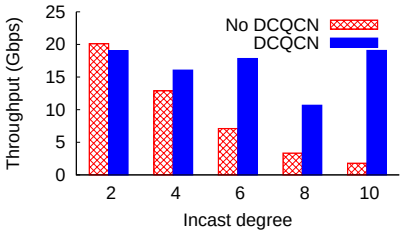
These results would imply that in our deployment, we should mark packets like DCTCP, and rely on faster rate increase timer to achieve fairness. However, we have found that in multi-bottleneck scenario (§7), a combination of faster timer and RED-like marking achieves better performance. Figure 13(d) shows the behavior of the combined solution for the simple two-flow scenario.

We have also verified that the value of $g = 1/256$ works well in this scenario, as predicted by the fluid model. Finally, using 20 machines connected via a single switch, we verified that with 55 μs timer, RED-ECN and $g = 1/256$, the total throughput is always more than 39Gbps for K:1 incast, $K = 1 \ldots 19$. The switch counter shows that the queue length never exceeds 100KB (translates to 20μs at 40Gbps).

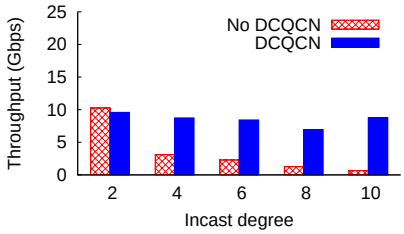
In summary, we use the parameters shown in Table 14 in our deployment, and also for the rest of the evaluation.

6.2 Benchmark traffic

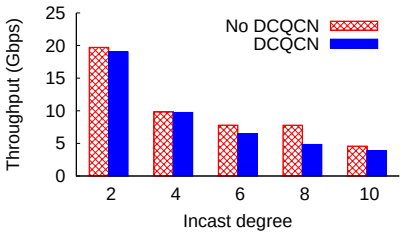
One important scenario for large-scale RDMA deployment is the backend network for cloud storage service. In such a network, the traffic consists of user requests, as well as traffic generated by relatively rare events such as disk recovery. The disk recovery traffic has incast-like character-



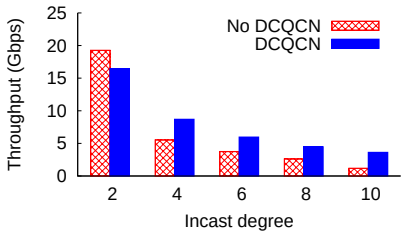
(a) Median throughput of user flows.



(b) 10th percentile throughput of user flows.

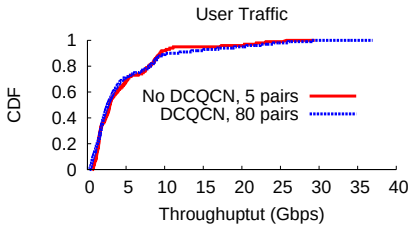


(c) Median throughput of incast flows.

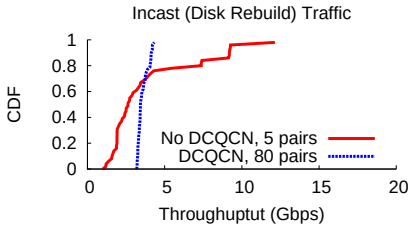


(d) 10th percentile throughput of incast flows.

Figure 16: DCQCN performance with Benchmark Traffic



(a) CDF of User traffic



(b) CDF of Incast traffic

Figure 17: DCQCN can handle higher user traffic without hurting performance

istics, as failed disks are repaired by fetching backups from several other servers [16]. We now evaluate the performance of DCQCN in this scenario using the 3-tier testbed shown in Figure 2, with five hosts connected to each ToR.

To mimic user request traffic, we use traces collected from a single cluster in our datacenter. The data was collected for one day on cluster of 480 machines, and includes traffic from over 10 million flows. The trace cannot be directly replayed on our testbed. Instead, like prior work [2, 5, 6], we extract salient characteristics of the trace data, such as flow size distribution and generate synthetic traffic to match these characteristics. We model disk recovery traffic as incast. We don't claim that this benchmark models the real traffic exactly - only that it is a reasonable proxy in a testbed setting.

To simulate user traffic, each host communicates with one or more randomly selected host, and transfers data using distributions derived from traces. The number of communicating pairs is fixed at 20 (we will vary this in a later experiment). The traffic also includes a *single* disk rebuild event (since these are not very frequent). We vary the degree of incast from 2 to 10, to model different erasure coding patterns [16]. We repeat the experiment five times, with and without DCQCN. DCQCN parameters are set according to Table 14. The communicating pairs as well as the hosts participating in the incast are selected randomly for each run. Each run lasts for two minutes.

The metric of interest is the median as well as the tail end (10th percentile) of the throughput[3]. Improved median means better utilization of the network, while improved tail performance leads to more predictable application performance, which in turn leads to better user experience.

The results are shown in Figure 16. Figure 16(a) and 16(b) show that without DCQCN, the throughput of user traffic falls rapidly, as the degree of incast goes up. This result is surprising, because even though the degree of incast goes up, the total traffic generated by incast senders stays the same (because the bottleneck is always at the receiver). The reason is the "damage" caused by PFC. As the degree of incast goes up, more PAUSE messages are generated. As these cascade through the network, they wreak havoc on user traffic. PAUSE messages that affect downlinks from spine switches can cause extensive damage, as many flows pass through these switches. For example, in a single run of the experiment with incast degree 10, the two spine switches together receive over 6 million PAUSE messages. Thus, scenarios like unfairness and victim flow (§2.2) become more likely and affect the throughput of user flows. In contrast, when DCQCN is used, the spine switches see just 3000 PAUSE messages (Figure 15). As a result, the performance of user traffic is much better. Indeed, with DCQCN, as the degree of incast goes up, there is little change in the median and tail throughput of user traffic – exactly as one would hope for.

The disk rebuild traffic also benefits from DCQCN because DCQCN helps divides the bandwidth fairly among competing flows. In the ideal case, the throughput of each incast flow should be equal to 40Gbps divided by degree of incast. Figure 16(d) shows that with DCQCN, the 10th percentile throughput is very close to this value, indicating a high degree of fairness. In contrast, without DCQCN, the 10th percentile throughput is much lower. For example, with incast degree of 10, 10th percentile throughput without DCQCN is less than 1.12Gbps, while with DCQCN it is 3.43Gbps. Note that the median throughput (Figure 16(c)) with DCQCN appears to be lower. This, however, is deceptive: without DCQCN the tail end suffers, and other flows

[3]90th percentile of response time.

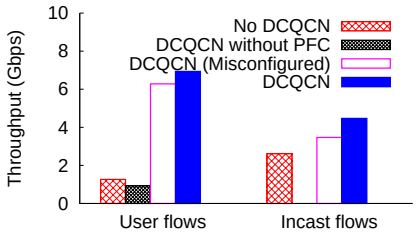


Figure 18: 10th percentile throughput of four configurations for 8:1 incast

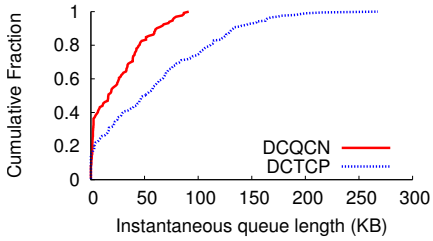


Figure 19: Queue length CDF

grab more than their "fair share". Thus, the median seems higher. With DCQCN, median and 10th percentile values are nearly identical.

Higher user load: In Figure 16, we saw that DCQCN continues to provide good performance to user traffic even as the degree of incast goes up. Now, we hold the degree of incast constant at 10, and vary the amount of use traffic by varying the number of simultaneously communicating pairs from 5 to 80. In Figure 17(a), we see that the performance of user traffic with 5 communicating pairs when no DCQCN is used, matches the performance of user traffic with 80 communicating pairs, with DCQCN. In other words, with DCQCN, we can handle 16x more user traffic, without performance degradation. Figure 17(b) shows that even the performance of disk rebuild traffic is more uniform and fair with DCQCN than without it.

Need for PFC and importance of buffer thresholds: We now demonstrate the need for PFC, and also the need for configuring correct buffer thresholds. To do so, we repeat the experiment of Figure 16 with two other settings. First, we disable PFC entirely, although we do use DCQCN. Second, we enable PFC, but deliberately "misconfigure" the buffer thresholds as follows: instead of using dynamic t_{PFC}, we use the static upper bound – i.e. $t_{PFC} = 24.47KB$ and set $t_{ECN} = 120KB$, i.e. five times this value. With these thresholds, it is no longer guaranteed that ECN will be triggered before PFC.

The results from the experiment are shown in Figure 18 for the 8:1 incast case. In addition to the two configurations discussed above, we also show the DCQCN and No DCQCN (i.e. PFC only) bars, which are copied from Figures 16(b) and 16(d).

Consider the performance when PFC is completely disabled. Recall that DCQCN has no slow start, and hence it relies on PFC to prevent packet loss during congestion. Without PFC, packet losses are common, and this leads to poor performance. In fact, the 10th percentile incast throughput is *zero* – indicating that some of the flows are simply unable to recover from persistent packet losses. This result underscores the need to use DCQCN with PFC.

Next, consider the performance when PFC is enabled, but buffer thresholds are misconfigured. With these thresholds, PFC may be triggered before ECN – thus preventing the full benefit of DCQCN. We see that this is indeed the case. The tail throughput of both user and incast traffic is better than what it would be without DCQCN (i.e. with PFC only), but worse than what it would be with DCQCN. This result underscores the need for correct buffer threshold settings.

6.3 Impact on latency

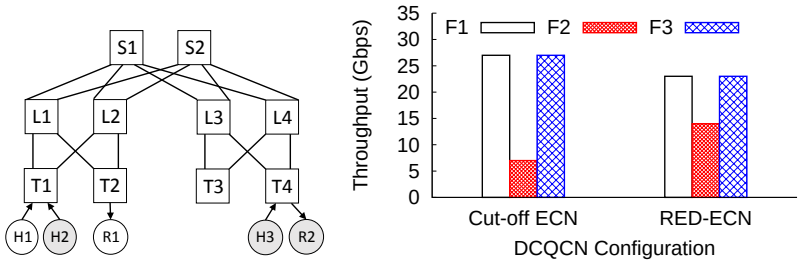
The above benchmarks use throughput as the primary metric. However, DCQCN can also reduce latency, compared to DCTCP. As a microbenchmark, we looked at the queue length of the egress queue of the congested port during 20:1 incast, with both DCTCP and DCQCN. For DCTCP, we configured 160KB ECN threshold according to the guidelines provided in [2]. DCQCN was configured as described earlier.

With this setup, the queue length with DCQCN is significantly shorter than that with DCTCP (Figure 19). For example, the 95th-percentile queue length is 76.6KB with DCQCN and 162.9KB with DCTCP. These numbers align well with DCQCN fluid model and DCTCP model [2]. DCTCP leads to longer queue, because DCTCP requires a larger ECN threshold to absorb packet bursts, which is a result of the interaction between the OS and the NIC. DCQCN uses hardware rate limiters, and thus can use shallower $K_{\min}$.

7. DISCUSSION

Multi-bottleneck scenario: We mentioned earlier that in a multi-bottleneck scenario combination of faster timer and RED-like marking gives better performance than faster timer alone. We now illustrate this. The multi-bottleneck problem is also known as the parking lot scenario. An example is shown in Figure 20(a). There are three flows: $f1$: H1->R1, $f2$: H2->R2, and $f3$: H3->R2. Consider the case when ECMP maps $f1$ and $f2$ to the same uplink from T1. With this mapping, $f2$ ends up with two bottlenecks (T1 uplink and T4->R2), while $f1$ and $f3$ have only one bottleneck each. Max-min fairness requires each flow should get throughput of 20Gbps. However, with common congestion control protocols like DCTCP, the flow with two bottleneck gets lower throughput because it has higher probability of getting congestion signal. DCQCN, if used with DCTCP-like "cut-off" marking ($K_{\min} = K_{\max}, P_{\max} = 1$, see Figure 5) has the same problem. The problem can be mitigated (but not completely solved) using a less abrupt, RED-like marking scheme (see parameters in Table 14).

The intuition is the same as explained before: CNP generation is driven by a timer, so with RED-like packet marking, we increase the probability that the faster flow will get more CNPs. Figure 20(b) shows the performance of the two marking schemes, and indicates that the intuition is correct. We



(a) Multi-bottleneck topology (b) Throughput of the three flows. F2 is the long flow (H2->R2)

Figure 20: Multi-bottleneck

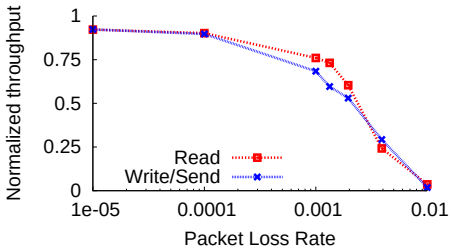


Figure 21: Impact of packet loss rate on throughput

plan to extend the fluid model to gain a better understanding of this scenario and come up with a more concrete solution.

Packet loss: PFC prevents packet loss due to buffer overflow. However, packet corruption, intermittent port failures, or accidental switch or server misconfiguration can induce packet loss in large networks [41]. Unlike TCP, RoCEv2 transport layer assumes that packet losses are rare. For example, when using READ operation [19], the protocol provides no information to the sender of the data about whether packets were correctly received. The receiver can detect packet loss via break in sequence numbers, and must initiate any loss recovery. The ConnectX-3 Pro implements a simple go-back-N loss recovery scheme. This is sufficient for environments where packet loss rate low, as seen in Figure 21. Throughput degrades rapidly once the packet loss rate exceeds 0.1%. With go-back-n loss recovery model, the size of the message matters as well. In our deployment, most message transfers are expected to be less than 4MB in size.

Deadlock: A commonly expressed concern is that use of PAUSE can lead to routing deadlocks [37]. Deadlock formation requires a set of flows that have a cyclic dependency on each other's buffers, as shown in [37]. In a clos-structured network servers are only connected to ToRs, and traffic between a pair of servers never passes through another server, or more than two ToRs. Thus a cyclic buffer dependency cannot arise, without malfunctioning or misconfigured equipment. We omit detailed proof due to lack of space. We are currently studying how to avoid outages that may be caused by a malfunctioning card (e.g. port or a NIC that spews PFCs).

TCP friendliness: TCP-friendliness was not one of the design goals for DCQCN, so we have not attempted to study in the interaction between TCP and DCQCN in this paper. It is easy to isolate TCP and DCQCN traffic from each other at the switch level using 802.3 priority tags, and enforce different transmission priorities, switch buffer utilization limits, and rate limits for each type of traffic.

Incremental deployment: Incremental deployment is not a concern in a datacenter environment.

Large-scale evaluation: We have designed DCQCN for deployment in large datacenter networks. The evaluation in this paper is based on a small, controlled, testbed environment. We have also evaluated DCQCN via large scale simulations (three-tier network of over 800 machines). We omit simulation results due to lack of space, and also because they are qualitatively similar to results from testbed evaluation. A full scale deployment of DCQCN in our datacenters is in progress.

8. RELATED WORK

There is a vast amount of research literature on congestion control, and datacenter networking technologies. Here, we only cover a few closely related ideas that we have not discussed elsewhere in the paper.

Dynamic buffer allocation among output queued switches is discussed in [7]. A number of papers have discussed various aspects of QCN's performance. For example, [9] discusses performance of a QCN in a TCP incast scenario. RECN-IQ [32] is a congestion management proposal for input-queued switches. While related, ideas in these proposals are not directly applicable to our scenario.

The iWarp standard [35] is an alternative to RoCEv2. It implements the full TCP stack on the NIC and builds a reliable connection using TCP's end-to-end loss recovery. RDMA can be enabled atop such a network. Due to end-to-end loss recovery, iWarp cannot offer ultra-low latency like RoCEv2. iWarp is also likely to suffer from well-known TCP problems in a datacenter environment, including poor incast performance [39]. In general, the iWarp technology has lagged behind RoCEv2, because of the complexity involved in implementing the full TCP stack in hardware. While 40Gbps iWarp hardware has only recently become available [43], RoCE2 vendors have started shipping 100Gbps NICs [44].

TCP Bolt [37] is a solution to overcome PFC's limitations. It also relies on flow-level congestion control, enabled by ECN. TCP Bolt is essentially DCTCP [2] without slow start. However, TCP-Bolt is implemented in end host stacks, and thus will have high CPU overhead and high latency. The protocol also does not consider the interplay between PFC and ECN buffer thresholds. Since we did not have access to a TCP-Bolt implementation for Windows, we were unable to compare the performance of DCQCN and TCP-Bolt.

We also note that DCTCP and iWarp have a slow start phase, which can result in poor performance for bursty stor-

age workloads. Furthermore, DCTCP, iWarp and TCP-Bolt are all window-based algorithms. DCQCN is rate-based, and uses hardware rate limiters. Thus, DCQCN offers a more fine-grained control over sending rate, resulting in lower queue lengths, as shown in §6.

TIMELY [31] is a congestion control scheme for datacenter traffic, including RDMA traffic. It uses fine-grained changes in RTT as a congestion signal, instead of ECN marks. Compared to DCTCP, TIMELY can significantly reduce queuing delay. Reducing CPU utilization of end hosts is not a goal for TIMELY. TIMELY was developed at Google in parallel with our work on DCQCN. It would be interesting to compare the performance of DCQCN and TIMELY under various scenarios.

User-level stacks such as Sandstorm [29] are one way to reduce TCP's CPU overhead and latency. While they do have lower CPU overhead than conventional stacks, their overhead is significantly higher than RoCEv2. Some of these stacks also have to be closely integrated with individual applications to be effective. TCP's latency can potentially also be reduced using technologies such as Netmap [28], Intel Data Direct I/O [45] and mTCP [23].

A number of proposals for reducing latency of TCP flows in datacenter and other environments have been put forth. Recent examples include HULL [5], pFabric [6] Detail [42], Fastpass [33] and RCP [12]. These proposals require significant changes to switch architectures, and to our knowledge, are not being deployed on a large scale. Other TCP offloading work optimize TCP performance only for specific scenarios, e.g. for VM [26].

Recent proposals consider the possibility in leveraging machine learning to adapt TCP parameters for better performance [36, 40]. The idea may also apply to DCQCN. We will investigate this in the future.

9. CONCLUSION AND FUTURE WORK

The history of congestion control algorithms is the history of struggle between responsiveness and stability. While we have presented DCQCN as a solution to cure PFC's ills, it also represents a new way to address the tussle between responsiveness and stability, at least in the datacenter environment. DCQCN uses the blunt, but fast PFC flow control to prevent packet losses just in time, and uses a fine-grained and slower end-to-end congestion control to adjust sending rate to avoid triggering PFC persistently. The combination allows DCQCN to be both responsive in short term, and stable over long term. We believe that this design point warrants exploration in a number of different directions. For example, we are exploring the use of DCQCN in other environments, with different RTTs and link bandwidths. More specifically, we are working to model and tune DCQCN and PFC for 100 and 400Gbps networks. We are also planning to investigate the stability of the DCQCN algorithm using the model described in §5.

Acknowledgements

We would like to thank Diego Crupnicoff, Liran Liss, Hillel Chapman, Marcel Apfelbaum, Dvir Aizenman and Alex Shpiner of Mellanox for their help with the design and implementation of DCQCN. We also like to thank the anonymous SIGCOMM reviewers and our shepherd, Nandita Dukkipati for their helpful comments.

10. REFERENCES

[1] M. Alizadeh, B. Atikoglu, A. Kabbani, A. Lakshmikantha, R. Pan, B. Prabhakar, and M. Seaman. Data center transport mechanisms: Congestion control theory and IEEE standardization. In *Allerton*, 2008.

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

[3] M. Alizadeh, A. Javanmard, and B. Prabhakar. Analysis of DCTCP: Stability, convergence and fairness. In *SIGMETRICS*, 2011.

[4] M. Alizadeh, A. Kabbani, B. Atikoglu, and B. Prabhakar. Stability analysis of QCN: the averaging principle. In *SIGMETRICS*, 2011.

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

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

[7] A. K. Choudhury and E. L. Hahne. Dynamic queue length thresholds for shared-memory packet switches. *IEEE/ACM Transactions on Networking*, 6(2), 1998.

[8] Cisco. Priority flow control: Build reliable layer 2 infrastructure. http://www.cisco.com/en/US/prod/collateral/switches/ps9441/ps9670/white_paper_c11-542809_ns783_Networking_Solutions_White_Paper.html.

[9] P. Devkota and A. L. N. Reddy. Performance of quantized congestion notification in TCP incast scenarios of data centers. In *MASCOTS*, 2012.

[10] A. Dragojevic, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast remote memory. In *NSDI*, 2014.

[11] J. Duetto, I. Johnson, J. Flich, F. Naven, P. Garcia, and T. Nachiondo. A new scalable and cost-effective congestion management strategy for lossless multistage interconnection networks. In *HPCA*, 2005.

[12] N. Dukkipati. Rate control protocol (RCP): Congestion control to make flows complete quickly. In *PhD diss., Stanford University*, 2007.

[13] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1:397–413, 1993.

[14] E. G. Gran, M. Eimot, S.-A. Reinemo, T. Skeie,

O. Lysne, L. P. Huse, and G. Shainer. First experiences with congestion control in infiniband hardware. In *IPDPS*, 2010.

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

[16] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. Erasure coding in Windows Azure storage. In *USENIX ATC*, 2012.

[17] IEEE. 802.11Qau. Congestion notification, 2010.

[18] IEEE. 802.11Qbb. Priority based flow control, 2011.

[19] Infiniband Trade Association. InfiniBand architecture volume 1, general specifications, release 1.2.1, 2008.

[20] Infiniband Trade Association. Supplement to InfiniBand architecture specification volume 1 release 1.2.2 annex A16: RDMA over converged ethernet (RoCE), 2010.

[21] Infiniband Trade Association. InfiniBand architecture volume 2, physical specifications, release 1.3, 2012.

[22] Infiniband Trade Association. Supplement to InfiniBand architecture specification volume 1 release 1.2.2 annex A17: RoCEv2 (IP routable RoCE), 2014.

[23] E. Jeong, S. Woo, A. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mTCP: a highly scalable user-level TCP stack for multicore systems. In *NSDI*, 2014.

[24] S. Kamil, L. Oliker, A. Pinar, and J. Shalf. Communication requirements and interconnect optimization for high-end scientific applications. *IEEE TPDS*, 21:188–202, 2009.

[25] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. The nature of datacenter traffic: Measurements and analysis. In *IMC*, 2009.

[26] A. Kangarlou et al. vSnoop: Improving TCP throughput in virtualized environments via acknowledgement offload. In *SC*, 2010.

[27] S. Larsen, P. Sarangam, and R. Huggahalli. Architectural breakdown of end-to-end latency in a TCP/IP network. In *SBAC-PAD*, 2007.

[28] Luigi Rizzo. netmap: a novel framework for fast packet I/O. In *USENIX ATC*, 2012.

[29] I. Marinos, R. N. Watson, and M. Handley. Network stack specialization for performance. In *SIGCOMM*, 2014.

[30] C. Mitchell, Y. Geng, and J. Li. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *USENIX ATC*, 2013.

[31] R. Mitta, E. Blem, N. Dukkipati, T. Lam, A. Vahdat, Y. Wang, H. Wassel, D. Wetherall, D. Zats, and M. Ghobadi. TIMELY: RTT-based congestion control for the datacenter. In *SIGCOMM*, 2015.

[32] G. Mora, P. J. Garcia, J. Flich, and J. Duato. RECN-IQ: A cost-effective input-queued switch architecture with congestion management. In *ICPP*, 2007.

[33] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. Fastpass: A centralized, zero-queue datacenter network. In *SIGCOMM*, 2014.

[34] K. Ramakrishnan, S. Floyd, and D. Black. The addition of explicit congestion notification (ECN). RFC 3168.

[35] M. Recio, B. Metzler, P. Culley, J. Hilland, and D. Garcia. A remote direct memory access protocol specification. RFC 5040.

[36] A. Sivaraman, K. Winstein, P. Thaker, and H. Balakrishnan. An experimental study of the learnability of congestion control. In *SIGCOMM*, 2014.

[37] B. Stephens, A. Cox, A. Singla, J. Carter, C. Dixon, and W. Felter. Practical DCB for improved data center networks. In *INFOCOMM*, 2014.

[38] H. Subramoni, S. Potluri, K. Kandalla, B. Barth, J. Vienne, J. Keasler, K. Tomko, K. Schulz, A. Moody, and D. Panda. Design of a scalable InfiniBand topology service to enable network-topology-aware placement of processes. In *SC*, 2012.

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

[40] K. Winstein and H. Balakrishnan. TCP ex machina: computer-generated congestion control. In *SIGCOMM*, 2013.

[41] X. Wu, D. Turner, G. Chen, D. Maltz, X. Yang, L. Yuan, and M. Zhang. Netpilot: Automating datacenter network failure mitigation. In *SIGCOMM*, 2012.

[42] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz. Detail: Reducing the flow completion time tail in datacenter networks. In *SIGCOMM*, 2012.

[43] Chelsio Terminator 5 ASIC. http://www.chelsio.com/nic/rdma-iwarp/.

[44] ConnectX-4 single/dual-port adapter supporting 100gb/s. http://www.mellanox.com/.

[45] Intel data direct I/O technology. http://www.intel.com/content/www/us/en/io/data-direct-i-o-technology-brief.html.

[46] Iperf - the TCP/UDP bandwidth measurement tool. http://iperf.fr.

[47] Offloading the segmentation of large TCP packets. http://msdn.microsoft.com/en-us/library/windows/hardware/ff568840(v=vs.85).aspx.

[48] QueryPerformanceCounter function. http://msdn.microsoft.com/en-us/library/windows/desktop/ms644904(v=vs.85).aspx.

[49] Receive Side Scaling (RSS). http://technet.microsoft.com/en-us/library/hh997036.aspx.