Re-architecting datacenter networks and stacks for low latency and high performance

Mark Handley
University College London
London, UK
m.handley@cs.ucl.ac.uk

Costin Raiciu
Alexandru Agache
Andrei Voinescu
University Politehnica of Bucharest
Bucharest, Romania
firstname.lastname@cs.pub.ro

Andrew W. Moore
Gianni Antichi
Marcin Wójcik
University of Cambridge
Cambridge, UK
firstname.lastname@cl.cam.ac.uk

ABSTRACT

Modern datacenter networks provide very high capacity via redundant Clos topologies and low switch latency, but transport protocols rarely deliver matching performance. We present NDP, a novel datacenter transport architecture that achieves near-optimal completion times for short transfers and high flow throughput in a wide range of scenarios, including incast. NDP switch buffers are very shallow and when they fill the switches trim packets to headers and priority forward the headers. This gives receivers a full view of instantaneous demand from all senders, and is the basis for our novel, high-performance, multipath-aware transport protocol that can deal gracefully with massive incast events and prioritize traffic from different senders on RTT timescales. We implemented NDP in Linux hosts with DPDK, in a software switch, in a NetFPGA-based hardware switch, and in P4. We evaluate NDP's performance in our implementations and in large-scale simulations, simultaneously demonstrating support for very low-latency and high throughput.

CCS CONCEPTS

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

KEYWORDS

Datacenters; Network Stacks; Transport Protocols

ACM Reference format:
Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. 2017. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of , , ,* 14 pages. https://doi.org/10.1145/3098822.3098825

1 INTRODUCTION

Datacenters have evolved rapidly over the last few years, with Clos[1, 17] topologies becoming commonplace, and a new emphasis on low latency, first with improved transport protocols such as DCTCP[4] and more recently with solutions such as RDMA over Converged Ethernet v2[25] that use Ethernet flow control in switches[23] to avoid packet loss caused by congestion.

In a lightly loaded network, Ethernet flow control can give very good low-delay performance[20] for request/response flows that dominate datacenter workloads. Packets are queued rather than lost, if necessary producing back-pressure, pausing forwarding across several switches, and so no time is wasted on being overly conservative at start-up or waiting for retransmission timeouts. However, based on experience deploying RoCEv2 at Microsoft[20], Gau *et al.*note that a lossless network does not guarantee low latency. When congestion occurs, queues build up and PFC pause frames are generated. Both queues and PFC pause frames increase network latency. They conclude *"how to achieve low network latency and high network throughput at the same time for RDMA is still an open problem."*

In this paper we present a new datacenter protocol architecture, NDP, that takes a different approach to simultaneously achieving both low delay and high throughput. NDP has no connection setup handshake, and allows flows to start sending instantly at full rate. We use per-packet multipath load balancing, which avoids core network congestion at the expense of reordering, and in switches use an approach similar to Cut Payload (CP)[9], which trims the payloads of packets when a switch queue fills. This gives a network that is lossless for metadata, but not for traffic payloads. In spite of reordering, lossless metadata gives the receiver a complete picture regarding inbound traffic and we take advantage of it to build a radical new transport protocol that achieves very low latency for short flows, with minimal interference between flows to different destinations even in pathological traffic patterns.

We have implemented NDP in Linux hosts, in a software switch, in a hardware switch based on NetFPGA SUME[41], in P4[29], and in simulation. We will demonstrate that NDP achieves:

- Better short-flow performance than DCTCP or DCQCN.
- Greater than 95% of the maximum network capacity in a heavily loaded network with switch queues of only eight packets.
- Near-perfect delay and fairness in incast[18] scenarios.
- Minimal interference between flows to different hosts.
- Effective prioritization of straggler traffic during incasts.

2 DESIGN SPACE

Intra-datacenter network traffic primarily consists of request/response RPC-like protocols. Mean network utilization is rarely very high, but applications can be very bursty. The big problem today is latency, especially for short RPC-like workloads. At the cost of head-of-line blocking, today's applications often reuse TCP connections across multiple requests to amortize the latency cost of the TCP handshake.

Is it possible to improve the protocol stack so much that every request could use a new connection and at the same time expect to get close to the raw latency and bandwidth of the underlying network, even under heavy load? We will show that these goals are achievable, but to do so involves changes to how traffic is routed, how switches cope with overload, and most importantly, requires a completely different transport protocol from those used today. Before we describe our solution in §3, we first highlight the key architectural points that must be considered.

2.1 End-to-end Service Demands

What do applications want from a datacenter network?

Location Independence. It shouldn't matter which machine in a datacenter the elements of a distributed application are run on. This is commonly achieved using high-capacity parallel Clos topologies[1, 17]. Such topologies have sufficient cross-sectional bandwidth that the core network should not be a bottleneck.

Low Latency. Clos networks can supply bandwidth, modulo issues with load balancing between paths, but often fall short in providing low latency service. Predictable very low latency request/response behavior is the key application demand, and it is the hardest to satisfy. This is more important than large file transfer performance, though high throughput is still a requirement, especially for storage servers. The strategy must be to optimize for low latency first.

Incast. Datacenter workloads often require sending requests to large numbers of workers and then handling their near-simultaneous responses, causing a problem called incast. A good networking stack should shield applications from the side-effects of incast traffic patterns gracefully while providing low latency.

Priority. It is also common for a receiver to handle many incoming flows corresponding to *different* requests simultaneously. For example, it may have fanned out two different requests to workers, and the responses to those requests are now arriving with the last responses to the first request overlapping the first responses to the second request. Many applications need all the responses to a request before they can proceed. A very desirable property is for the receiver to be able to prioritize arriving traffic from stragglers. The receiver is the only entity that can dynamically prioritize its inbound traffic, and this impacts protocol design.

2.2 Transport Protocol

Current datacenter transport protocols satisfy some of these application requirements, but satisfying all of them places some unusual demands on datacenter transport protocols.

Zero-RTT connection setup. To minimize latency, many applications would like *zero-RTT delivery* for small outgoing transfers (or one RTT for request/response). We need a protocol that doesn't require a handshake to complete before sending data, but this poses security and correctness issues.

Fast start. Another implication of zero-RTT delivery is that a transport protocol can't *probe* for available bandwidth—to minimize latency, it must assume bandwidth is available, optimistically send a full initial window, and then react appropriately when it isn't. In contrast to the Internet, simpler solutions are possible in datacenter environments, because link speeds and network delays (except for queuing delays) can mostly be known in advance.

Per-packet ECMP. One problem with Clos topologies is that per-flow ECMP hashing of flows to paths can cause unintended flow collisions; one deployment[20] found this reduced throughput by 40%. For large transfers, multipath protocols such as MPTCP can establish enough subflows to find unused paths[31], but they can do little to help with the latency of very short transfers. The only solution here is to stripe across multiple paths on a per-packet basis. This complicates transport protocol design.

Reorder-tolerant handshake. If we perform a zero-RTT transfer with per-packet multipath forwarding in a Clos network, even the very first window of packets may arrive in a random order. This effect has implications for connection setup: the first packet to arrive will not be the first packet of the connection. Such a transport protocol must be capable of establishing connection state no matter which packet from the initial window is first to arrive.

Optimized for Incast. Although Clos networks are well-provisioned for core-capacity, incast traffic can make life difficult for any transport protocol when applications fan out requests to many workers simultaneously. Such traffic patterns can cause high packet loss rates, especially if the transport protocol is aggressive in the first RTT. To handle this gracefully requires some assistance from the switches.

2.3 Switch Service Model

Application requirements, the transport protocol and the service model at network switches are tightly coupled, and need to be optimized holistically. Of particular relevance is what happens when a switch port is congested. The switch service model heavily influences the design space of both protocol and congestion control algorithms, and couples tightly with forwarding behavior: per-packet multipath load balancing is ideal as it minimizes hotspots, but it complicates the ability of end-systems to infer network congestion and increases importance of graceful overload behavior.

Loss as a congestion feedback mechanism has the advantage that dropped packets don't use bottleneck bandwidth, and loss only impacts flows traversing the congested link—not all schemes have these properties. The downside is that it leads to uncertainty as to a packet's outcome. Duplicate or selective ACKs to trigger retransmissions only work well for long-lived flows. With short flows, tail loss is common, and then you have to fall back on retransmission timeouts (RTO). Short RTOs are only safe if you can constrain the delay in the network, so you need to maintain short queues[4] which in turn constrain the congestion control schemes you can use. Loss also couples badly with per-packet multipath forwarding; because the packets of a flow arrive out of order, loss detection is greatly complicated - fast retransmit is often not possible because its not rare for a packet to arrive out of sequence by a whole window.

ECN helps significantly. DCTCP uses ECN[32] with a sharp threshold for packet marking, and a congestion control scheme that aims to push in and out of the marking regime. This greatly reduces loss for long-lived flows, and allows the use of small buffers, reducing queuing delay. For short flows though, ECN has less benefit because the flow doesn't have time to react to the ECN feedback. In practice switches use large shared buffers in conjunction with ECN and this reduces incast losses, but retransmit timers must be less

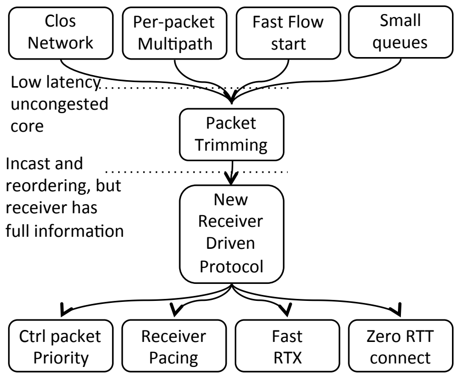


Figure 1: Key components of NDP

aggressive. ECN does have the advantage though that it interacts quite well with per-packet multipath forwarding, given a transport protocol design that can tolerate reordering.

Lossless Ethernet using 802.3X Pause frames [23] or 802.1 Qbb priority-based flow control (PFC)[24] can prevent loss, avoiding the need for aggressive RTO in protocols. At low utilizations, this can be effective at achieving low delay—a burst will arrive at the maximum rate that the link can forward, with no need to wait for retransmissions. The problem comes at higher utilizations in tiered topologies, where flows that happen to hash to the same outgoing port, and use the same priority in the case of 802.1Qbb, can cause incoming ports to be paused. This causes collateral damage to other flows traversing the same incoming port destined for different output ports. With large incasts, pausing can cascade back up towards core switches. Lossless Ethernet also interacts badly with per-packet multipath forwarding, as different switches may pause traffic at different times, exacerbating reordering and complicating end-system design.

Cut Payload (CP)[9] tries to get the benefits of lossless without quite being lossless. It drops packet payloads, but not packet headers, relieving overload while avoiding uncertainty as to packet outcomes. It shows great promise, but there are two problems. First, in severe overload, it is susceptible to congestion collapse, where only headers get forwarded. Second, because the headers are queued in a FIFO manner, tail "loss" costs at least one RTT. In addition, CP, as originally proposed uses single-path forwarding for each flow.

3 DESIGN

Our primary goals are *low completion latency* for short flows, and *predictable high throughput* for longer flows. To fully satisfy these goals, NDP impacts the whole stack, including switch behavior, routing, and a completely new transport protocol. We lead with a brief but simplified design rationale to show how the pieces in Figure 1 fit together, then fill in the details in the rest of this section.

A Clos topology has sufficient bandwidth in the core to satisfy all demand, so long as it is perfectly load-balanced. To avoid flow collisions on core links, which impact both latency and throughput, load-balancing each flow across many paths is essential. Balancing short flows requires per-packet multipath load-balancing, but inevitably packets will get reordered.

To achieve minimal short-flow latency, senders cannot probe before sending: they must send the first RTT at line rate. This works well most of the time. When senders perform per-packet multipath

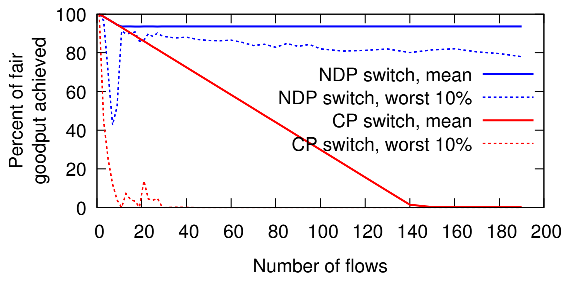


Figure 2: Collapse and Phase Problems with CP

load balancing, if sending at line rate causes congestion, it is because several senders are sending to the same receiver. Even then, the receiver's link is fully occupied, so this is not, by itself, a problem.

To guarantee low latency, switch queues must be small. This means colliding flows will overflow the queue. Packet loss, combined with multipath reordering, make it impossible to infer what happened and retransmit quickly enough to avoid impacting latency; this violates the low latency goal. Completely preventing packet loss adds queuing delay; if this is done by pausing inbound traffic, as with lossless Ethernet, this impacts other unrelated traffic, violating its low latency and predictable high throughput goals. We seek a middle ground between packet loss and lossless.

Packet trimming, similar to that performed by CP, is such a middle ground. Switch queues can be small, and the receiver still discovers which packets were sent by examining the trimmed headers it receives. However, to minimize retransmission latency, trimmed headers and control packets need to be prioritized. Arriving trimmed headers tell the receiver exactly what the demand is, so by using a receiver-pulled protocol, the receiver can then precisely control incoming traffic. This avoids persistent overload and allows more important packets to be pulled first, at the receiver's discretion.

3.1 NDP Switch Service Model

With CP, when the queue at a switch fills beyond a fixed threshold, rather than dropping a packet, the switch trims off the packet payload, queuing just the header. The rationale is that packets are not lost silently, allowing rapid retransmission without waiting for a timeout. With the short distances in a datacenter network, such retransmissions can arrive very quickly.

Alongside switch changes, CP proposes minor changes to TCP to improve incast performance. We wish to go well beyond CP, and use packet trimming as the basis of an extremely aggressive network architecture, focused on very low delay service. However, there are several problems that can arise if vanilla CP is used.

First, CP can suffer from a form of congestion collapse. Figure 2 shows what happens when packets arrive at a switch at a significantly higher rate than can be supported by the outgoing link. Many unresponsive flows converge on a 10Gb/s link that can only support one of them, as in extreme server incast scenarios. The figure shows the percent of the ideal fair-share goodput that is achieved. The mean goodput of the CP flows decreases, as an increasing fraction of the link is occupied by trimmed packet headers. This figure shows the best case for CP, with 9KB jumbograms. With 1500 byte packets the collapse is much faster.

Second, datacenter networks are very regular, so phase effects[14] can occur, leading to unfair throughput. The dashed curves in Figure 2 show the mean goodput of the worst performing 10% of the flows. Phase effects can render CP very unfair, though we note that this figure shows simulation results; real-world phase effects can sometimes be reduced by variability in the timing of packet transmissions due to OS scheduling.

Finally, CP aims to provide low delay feedback that packets have been lost. However, because CP uses a FIFO queue, feedback can only be sent after all the preceding packets have been received, resulting in a delay before a retransmission is elicited. We would like to run very small buffers in the switches, have one of those queues overflow, and for the retransmission to arrive before the queue has had a chance to drain. This isn't possible with FIFO queuing.

NDP switches make three main changes to CP. First, an NDP switch maintains two queues: a lower priority queue for data packets and a higher priority queue for trimmed headers, ACKs and NACKs[1]. This may seem counter-intuitive, but it provides the earliest possible feedback that a packet didn't make it, usually allowing a retransmission to arrive before the offending queue had even had time to drain. This can provide at least as good low delay behavior as lossless Ethernet, without the collateral damage caused by pausing.

Second, the switch performs weighted round robin between the high priority "header queue" and the lower priority "data packet queue". With a 10:1 ratio of headers to packets, this allows early feedback without being susceptible to congestion collapse.

Finally, when a data packet arrives and the low priority queue is full, the switch decides with 50% probability whether to trim the newly arrived packet, or the data packet at the tail of the low priority queue. This breaks up phase effects. Figure 2 shows how an NDP switch avoids CP's collapse, and also avoids strong phase effects.

3.1.1 Routing

We want NDP switches to perform per-packet multihop forwarding, so as to evenly distribute traffic bursts across all the parallel paths that are available between source and destination. This could be done in at least four ways:

- Perform per-packet ECMP; switches randomly choose the next hop for each packet;
- Explicitly source-route the traffic;
- Use label-switched paths; the sender chooses the label;
- Destination addresses indicates the path to be taken; the sender chooses between destination addresses.

For load-balancing purposes the latter three are equivalent—the sender chooses a path—they differ in how the sender expresses that path. Our experiments show that if the senders choose the paths, they can do a better job of load balancing than if the switches randomly choose paths. This allows the use of slightly smaller switch buffers.

Unlike in the Internet, in a datacenter, senders can know the topology, so know how many paths are available to a destination. Each NDP sender takes the list of paths to a destination, randomly permutes it, then sends packets on paths in this order. After it has sent one packet on each path, it randomly permutes the list of paths again, and the process repeats. This spreads packets equally across all paths while avoiding inadvertent synchronization between two senders. Such load-balancing is important to achieving very low delay. If we use very small data packet queues (only eight packets), our experiments show that this simple scheme can increase the maximum capacity of the network by as much as 10% over a per-packet random path choice.

Depending on whether the network is L2 or L3-switched, either label-switched paths or destination addresses can be used to choose a path. In an L2 FatTree network for example, a label-switched path only needs to be set up as far as each core switch, with destination L2 addresses taking over from there, as a FatTree only has one path from a core switch to each host. In an L3 FatTree, each host gets multiple IP addresses, one for each core switch. By choosing the destination address, the sender chooses the core switch a packet traverses.

3.2 Transport Protocol

NDP uses a receiver-driven transport protocol designed specifically to take advantage of multipath forwarding, packet trimming, and short switch queues. The goal at each step is first to minimize delay for short transfers, then to maximize throughput for larger transfers.

When starting up a connection, a transport protocol could be pessimistic, like TCP, and assume that there is minimal spare network capacity. TCP starts sending data after the three-way handshake completes, initially with a small congestion window[11], and doubles it each RTT until it has filled the pipe. Starting slowly is appropriate in the Internet, where RTTs and link bandwidths differ by orders of magnitude, and where the consequences of being more aggressive are severe. In a datacenter, though, link speeds and baseline RTTs are much more homogeneous, and can be known in advance. Also, network utilization is often relatively low [7]. In such a network, to minimize delay we must be optimistic and assume there will be enough capacity to send a full window of data in the first RTT of a connection without probing. If switch buffers are small, in a low-delay datacenter environment a full window is likely to be only about 12 packets given the speed-of-light latencies and hop-counts.

However, if it turns outs that there is insufficient capacity, packets will be lost. With a normal transport protocol, the combination of per-packet multipath forwarding and being aggressive in the first RTT is a recipe for confusion. Some packets arrive, but in a random order, and some don't. It is impossible to tell quickly what actually happened, and so the sender must fall back on conservative retransmission timeouts to remedy the situation.

Increasing switch buffering could mitigate this situation somewhat, at the expense of increasing delay, but can't prevent loss with large incasts. ECN also cannot prevent loss with aggressive short flows. Pause frames can prevent loss, and could help significantly here, but we will show in § 6.1 that this brings its own significant problems in terms of delay to unrelated flows.

This is where packet trimming in the NDP switches really comes into its own. Headers of trimmed packets arriving at the receiver consume little bottleneck bandwidth, but inform the receiver precisely which packets were sent. The order of packet arrivals is unimportant when it comes to inferring what happened. Priority queuing ensures that these headers arrive quickly, and that control packets such as NACKs returned to the sender arrive quickly; indeed quickly enough to elicit a retransmission that arrives before the overflowing queue

[1]Also PULL packets, which we will introduce shortly.

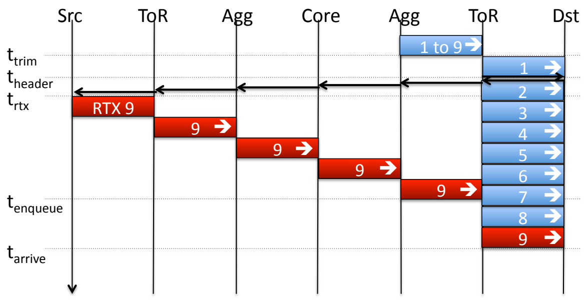


Figure 3: Packet trimming enables low-delay retransmission

has had time to drain, so the link does not go idle. This is illustrated in Figure 3. At time t_{trim} packets from nine different sources arrive nearly simultaneously at the ToR switch. The eight-packet queue to the destination link fills, and the packet from source 9 is trimmed. After packet 1 finishes being forwarded, packet 9's header gets priority treatment. At t_{header} it arrives at the receiver, which generates a NACK packet[2]. Packet 9 is retransmitted at t_{rtx} and arrives at the ToR switch queue while packet 7 is still being forwarded. The link to the destination never goes idle, and packet 9 arrives at t_{arrive}, the same time it would have arrived if PFC had prevented its loss by pausing the upstream switch.

In a Clos topology employing per-packet multipath, the only hot spots that can build are when traffic from many sources converges on a receiver. With NDP, trimmed headers indicate the precise demand to the receiver; it knows exactly which senders want to send which data to it, so it is best placed to decide what to do after the first RTT of a connection. After sending a full window of data at line rate, NDP senders stop sending. From then on, the protocol is receiver-driven. An NDP receiver requests packets from the senders, pacing the sending of those requests so that the data packets they elicit arrive at a rate that matches the receiver's link speed. The data requested can be retransmissions of trimmed packets, or can be new data from the rest of the transfer. The protocol thus works as follows:

- The sender sends a full window of data without waiting for a response. Data packets carry packet sequence numbers.
- For each header[3] that arrives, the receiver immediately sends a NACK to inform the sender to prepare the packet for retransmission (but not yet send it).
- For each data packet that arrives, the receiver immediately sends an ACK to inform the sender that the packet arrived, and so the buffer can be freed.
- For every header or packet that arrives, the receiver adds a *PULL packet* to its pull queue that will, in due course, be sent to the corresponding sender. A receiver only has one pull queue, shared by all connections for which it is the receiver.
- A PULL packet contains the connection ID and a per-sender *pull counter* that increments on each PULL packet sent to that sender.
- The receiver sends out PULL packets from the per-interface pull queue, paced so that the data packets they elicit from the sender then arrive at the receiver's link rate. Pull packets from different connections are serviced fairly by default, or with strict prioritization when a flow has higher priority.
- When a PULL packet arrives at the sender, the sender will send as many data packets as the pull counter increments by. Any packets queued for retransmission are sent first, followed by new data.
- When the sender runs out of data to send, it marks the last packet. When the last packet arrives, the receiver removes any pull packets for that sender from its pull queue to avoid sending unnecessary pull packets. Any subsequent data the sender later wants to send will be pushed rather than pulled.

Due to packet trimming, it is very rare for a packet to be actually lost; usually this is due to corruption. As ACKs and NACKs are sent immediately, are priority-forwarded, and all switch queues are small, the sender can know very quickly if a packet was actually lost. With eight packet switch queues, 9KB jumbograms, and store-and-forward switches in a 10Gb/s FatTree topology, each packet takes $7.2\mu s$ to serialize. Taking into account NDP's priority queuing, the worst-case network RTT is approximately $400\mu s$, with typical RTTs being much shorter. This allows a very short retransmission timeout to be used to provide reliability for such corrupted packets.

PULL packets perform a role similar to TCP's ACK-clock, but are usually[4] separated from ACKs to allow them to be paced without impacting the retransmission timeout mechanism. For example, in a large incast scenario, PULLs may spend a comparatively long time in the receiver's pull queue before the pacer allows them to be sent, but we don't want to also delay ACKs because doing so requires being much more conservative with retransmission timeouts.

The emergent behavior is that the first RTT of data in a connection is pushed, and subsequent RTTs of data are pulled so as to arrive at the receiver's line rate. In an incast scenario, if many senders send simultaneously, many of their first window of packets will be trimmed, but subsequently receiver pulling ensures that the aggregate arrival rate from all senders matches the receiver's link speed, with few or no packets being trimmed.

3.2.1 Coping with Reordering

Due to per-packet multipath forwarding, it is normal for both data packets and reverse-path ACKs, NACKs and PULLs to be reordered. The basic protocol design is robust to reordering, as it does not need to make inference about loss from other packets' sequence numbers. However, reordering still needs to be taken into account.

Although PULL packets are priority-queued, they don't preempt data packets, so PULL packets sent on different paths often arrive out of order, increasing the burstiness of retransmissions. To reduce this, PULLs carry a pull sequence number. The receiver has a separate pull sequence space for each connection, incrementing it by one for each pull sent. On receipt of a PULL, the sender transmits as many packets as the pull sequence number increases by. For example, if a PULL is delayed, the next PULL sent may arrive first via a different path, and will pull two packets rather than one. This reduces burstiness a little.

3.2.2 The First RTT

Unlike in TCP, where the SYN/SYN-ACK handshake happens ahead of data exchange, we wish NDP data to be sent in the first RTT. This adds three new requirements:

- Be robust to requests that spoof source IP addresses.

[2]This NACK has the PULL bit set, requesting retransmission.
[3]When we refer to headers in this context, we are referring to the headers of packets whose payload was trimmed off by a switch
[4]If there is only one sender, PULL packets don't need extra pacing because data packets arrive paced appropriately. In such cases we can send combined PULLACK.

- Ensure no connection is inadvertently processed twice.
- Cope with multipath reordering within the first RTT.

T/TCP[8] and TCP Fast Open[10] both extend TCP to send data in the first RTT. TFO prevents spoofing by presenting a token given by the server in a previous connection, but does not ensure at-most-once semantics. T/TCP uses monotonically increasing connection IDs giving at-most-once semantics, but it is not robust to spoofing. Neither copes well if the SYN is not the first packet to arrive.

NDP's requirements are slightly different. Spoofing can be prevented in the hypervisor or NIC, or using VXLAN for multi-tenant datacenters, so isn't a big problem. At-most-once semantics are crucial though, but T/TCP's solution is not robust to multipath reordering between back-to-back short connections. NDP solves this by instead by keeping time-wait state at both the client and server, so either can reject duplicate connections. As the maximum segment lifetime is under 1ms, the amount of extra state is fairly small.

Finally, multiple packets may be sent in the first RTT, but the first to arrive is often not the first sent. To be robust, every packet in the first RTT carries the SYN flag, together with the offset of its sequence number from the first packet in the connection. This allows connection state to be established by whichever packet arrives first.

3.2.3 *Robustness Optimizations*

If the network behaves properly, the protocol above performs very well. However, sometimes links or switches fail. This is normally detected by a routing protocol, and the failure is then routed around. NDP packets are source-routed, so NDP hosts also need to receive these routing updates to know which paths to avoid. However, before the routing protocol has informed everyone, packets arriving at a failed link will be lost. Other more subtle failures are also possible, such as a 10Gb/s link deciding to negotiate to 1Gb/s, resulting in a hot spot that will not immediately be detected by routing. Previous work has shown that using single-path congestion control (e.g. TCP) and in-network packet-spraying results in heavily reduced performance in these scenarios, because the transport protocol is not aware that only one of its paths is misbehaving [12, 31].

NDP incorporates optimizations that greatly improve performance in such cases. Senders keep a scoreboard for paths, and the sender keeps track of which path each packet traverses. When an ACK or NACK arrives, the ACK or NACK-count for the path the data packet was sent over is incremented. Normally, in a Clos topology running NDP, all paths should have a very similar ratio of ACKs to NACKs. However, if a failure has caused asymmetry, some links will have excessive NACK counts. When the sender permutes the path list, it temporarily removes outliers from the path set.

Packet loss should almost never occur. An NDP sender that retransmits a lost packet always resends it on a different path. A path loss counter is also incremented each time a packet is lost. Any paths that are outliers with regards to packet loss are also temporarily removed from the path set.

These mechanisms allow NDP to be robust to networks where paths no longer have similar performance, for whatever reason, with minimal loss in performance. Traditional protocols that rely on per-flow ECMP multipath forwarding have a harder time with such failures, and rely on routing to detect and avoid bad paths.

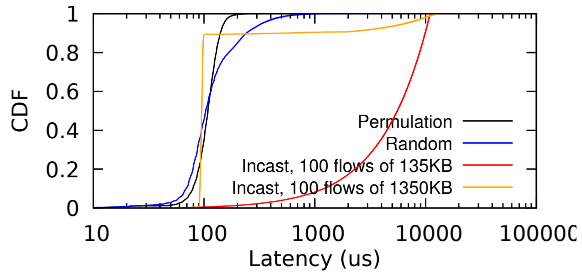


Figure 4: Delivery Latency with various traffic matrices.

3.2.4 *Return-to-Sender*

Packet trimming can cope with large incasts without needing to drop any headers. However, extremely large incasts may overflow the header queue, causing loss. The missing packets would be resent rapidly, when the sender's RTO expires. With small queues ensuring a 400μs maximum RTT, the *maximum* RTO could safely be as low as 1ms. During incast, a top-of-rack switch queue can hold eight 9KB packets and (in the same amount of memory) 1125 64-byte headers before it overflows. The receiver will PULL retransmissions of these 1125 packets, pacing their arrival to keep its link saturated. At 10Gb/s, 1125 packets of 9KB will occupy the receiver's link for 8ms so any packets resent due to RTO could still reach the queue that overflowed before the link goes idle.

However, sometimes a whole transfer will fit in a single packet, and that transfer may be of high priority—it may, for example, be the straggler from a previous request. If such a packet is lost, relying on the RTO adds unnecessary delay. As an optimization, when the header queue overflows, the switch can swap the sender and receiver's addresses, and return the header to the sender. The sender could then resend the offending packet. However, *always* resending could cause an echo of the original incast. NDP only resends if it is not expecting more PULLs—i.e, there are no packets ACKed or NACKed but not yet pulled, or if all other packets from the first window were also returned. This avoids incast echo, but keeps the pull-clock going. NDP also resends if most packets recently were ACKed rather than NACKed—this indicates an asymmetric network, where resending on a different working path makes sense.

Return-to-sender is an optimization; in our experience it only kicks in with very large incasts. In a Clos topology it essentially makes NDP lossless for metadata; an RTO only triggers when packets are corrupted or there is a failure.

Figure 4 shows results from a 432-node FatTree simulation that demonstrates these mechanisms in action, giving a CDF of latency from when a packet is first sent to when it is Acked at the sender, including any delay due to retransmissions. The Permutation curve shows when every host sends and receives from one other host and Random shows when each host sends to a random host - in both cases these fully load the FatTree, but the median latency remains around 100μs. The Incast curves show what happens when 100 nodes send simultaneously to a single node; they differ in the size of the transfer. With 135KB, all nodes send the entire file in the first RTT; this not only results in high trimming rates, but also overflows the header queue, with 25% of headers being returned to sender. Despite this, the last packet arrives in 11,055μs, only 2% later than the theoretical best arrival time. With 1350KB, the first window of data takes the same amount of time as the 135KB transfers, but the remainder of the transfer proceeds smoothly with no trimming and a median latency of 95μs.

Congestion Control

The astute reader may by now be wondering what NDP does for congestion control. The answer is simple: NDP performs no congestion control whatsoever in a Clos topology. As we will show, congestion control is simply unnecessary with the right combination of network service model and transport protocol. Broadly speaking, Internet congestion control serves two roles: it avoids congestion collapse, and it ensures fairness. NDP achieves both without having an explicit window adaptation mechanism.

Avoiding congestion collapse. As we saw in Section 2.3, NDP switches avoid CP's congestion collapse by ensuring that most of a link is used by data packets. Collapse might also occur if packets were discarded at the receiver, as with pre-Jacobson retransmission timeouts[26] or with fragmentation[27, 33]. Due to packet trimming, NDP senders rarely need to rely on the RTO, so collapse due to unnecessary retransmissions is not possible.

Collapse could happen if a large fraction of packets are discarded close to the receiver having already displaced other packets earlier on their path[15]. However, in a Clos topology with NDP's per-packet multipath routing, packets are almost never trimmed on the uplinks to the core switches because it is not possible to concentrate traffic there. When they are trimmed on uplinks, this is due to imperfect load balancing; this is where NDP's source-based load balancing provides a win over per-packet random ECMP performed by switches. Even under high load, packets trimmed here comprise a tiny fraction of overall traffic—for example, in simulations of a 128-node FatTree running a full permutation traffic matrix, where every node receives from one node and sends to another node at its linkspeed of 10Gb/s, we see 0.01% of packets trimmed on uplinks when the sources load balance, compared to 2.4% when the switches load-balance.

Significant trimming only really happens during incasts, with most packets being trimmed on the links from top-of-rack switches to hosts, and a few being trimmed between upper pod switches and lower pod switches. Thus packets that are trimmed by the ToR switches have only rarely displaced packets earlier in the topology.

Fairness. NDP achieves excellent fairness without needing additional mechanisms. All competing flows start with the same window, so there's no need to worry about convergence. The primary point when flows compete is for capacity to the receiver, and the receiver has a complete view of what is happening. Receiver fairness is achieved by using a fair queuing scheme for packets in the pull-queue that belong to different connections. Finally, deliberate unfairness is possible, because the receiver knows its own priorities, and can pull high priority traffic more often than low priority traffic.

One case of unfairness that cannot be receiver-managed is where a flow to one receiver competes with an incast to another receiver on the same ToR switch. NDP mitigates such unfairness in one RTT because, after that, receiver-pacing of PULLs removes the overload.

Limitations of NDP

Our experimental evaluation in §5 and §6 shows that NDP is very close to optimal in fully-provisioned folded Clos topologies, even with asymmetries, and regardless of the traffic patterns. Here we discuss NDP's limitations outside such networks.

In asymmetric topologies such as BCube [19] and Jellyfish [36], NDP will behave poorly because it will spray packets on different length paths that are costly to use when the network is heavily loaded. For such networks, sender-based per-path multipath congestion control has been shown to work well [31]; it remains an open question how to reconcile per-path congestion control with our pull-based receiver driven protocol.

Congestion control would also be desirable on heavily oversubscribed networks where the core is persistently congested, as NDP's aggressive design will lead to continuous packet trimming even after the first RTT. We show in our evaluation that NDP still provides better performance than DCTCP in such cases (see §6.3), but some form of congestion control would be useful to reduce server retransmission load.

One final question regards deployment: when P4 switches are widely deployed in datacenters, running NDP is as simple as deploying the switch implementation (§4) and the end-system stack. However, NDP may shut out competing TCP traffic. It is, however, simple to ensure coexistence with TCP by serving NDP and TCP from different queues, fair-queuing between them. The TCP queue will be larger (100s of packets) while NDP's will be small (8 packets), coupled with a similarly sized header queue.

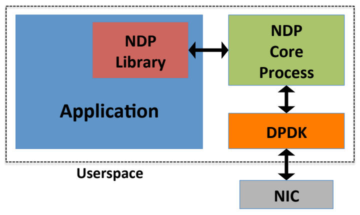


Figure 5: NDP software implementation architecture.

4 IMPLEMENTATION

We implemented NDP in Linux end-systems, a software switch based on DPDK[13], a hardware switch using the 10Gb/s NetFPGA SUME[41] platform, and in P4[29]. We also implemented NDP in the *htsim* high-speed network simulator, based on datacenter network implementations from [31]. We use the Linux and NetFPGA implementations to demonstrate performance at small scale on real hardware, and the simulations to demonstrate NDP's scaling properties. We have also developed a P4 implementation of the NDP switch as a proof of concept that NDP processing is simple enough to be easily deployed in programmable switches.

Linux Implementation. The goal of our Linux NDP implementation is to investigate NDP performance and validate the NDP protocol design. Normally we would expect NDP to be implemented in the OS kernel to allow accurate control of timing. To permit rapid experimentation, our approach instead implements NDP in userspace, using the DPDK library to achieve low-latency network access, and using a dedicated core to ensure accurate PULL pacing and low latency retransmissions. The architecture is shown in Figure 5. The NDP core process mediates NIC access and maintains the pull queue as all NDP connections must share this. The core process also handles fast retransmissions caused by NACKs. A library provides the NDP API to applications, and interacts with the core process via

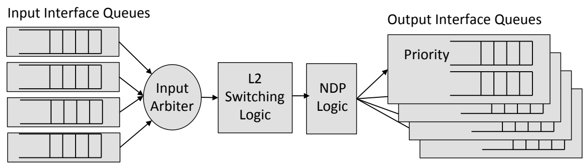


Figure 6: NDP switch architecture on NetFPGA-SUME

shared memory, passing commands such as *connect* and *listen* via a communications ring buffer, and data for each active socket via three ring buffers: RX, TX and RTX, and a shared buffer pool. The library also handles packet retransmissions due to timeouts.

The NDP core main loop checks for new application registrations, runs commands from library instances, sends the first RTT of packets from the TX ring of new connections, and handles incoming packets. Arriving data, ACK and NACK packets are placed in the appropriate socket's RX ring. Arriving PULLs cause data packets to be sent from the socket's RTX or TX ring, with RTX given priority. Arriving data packets (or headers) cause PULLs to be added to the pull queue.

A separate pull queue thread, running on its own CPU core, dequeues these PULLs one-by-one at the appropriate time and sends them; currently this thread spins to ensure appropriate granularity, but in the future this overhead might be avoided with NIC support.

Data packets can also trigger the initialization of a new socket if the SYN bit is set and listen was previously called. NACKs are passed to the library to avoid spurious timeouts, but the NDP core also adds the corresponding buffer index to the socket's RTX ring, allowing very fast retransmission.

Since NDP is a zero-RTT protocol, the *connect* command from the library only informs the NDP core about a new active socket. The connection will be established when data is sent. The *listen* command informs the NDP core about a new passive socket, but it also reserves a number of sockets for any incoming connections because the lack of an initial handshake means we must be ready to accept incoming packet trains on the fly.

NDP-enabled Hardware Switch. Ideally NDP's trimming and priority queuing would be implemented in switch ASICs. We prototyped such a solution using the NetFPGA-SUME platform [41], a reconfigurable hardware platform with four 10Gb/s Ethernet interfaces incorporating a Xilinx Virtex-7 FPGA together with QDRII+ and DDR3 memory resources.

Figure 6 shows the high level NDP switch design. Packets enter through one of the 10Gb/s interfaces and are stored in a 36Kbit interface input queue. The arbiter takes packets from the input queues using a deficit round-robin (DRR) scheduling policy, and feeds them to the L2 switching logic via a 256bit wide 200MHz bus, fast enough to support more than 40Gb/s. When many small packets arrive on one interface and many large ones arrive on others, DRR ensures that the input queue receiving small packets does not overflow.

After a conventional L2 forwarding decision is made, the packet reaches the NDP logic. Each output port has a low priority and a high priority output queue, each 12KByte long. NDP control packets are forwarded to the high priority queue. For remaining packets, the NDP logic checks the low priority queue length, enqueuing the packet if there is space. Otherwise the packet is trimmed and placed in the high-priority queue. If that queue is full, the packet is dropped. Note that a full implementation should randomly decide whether

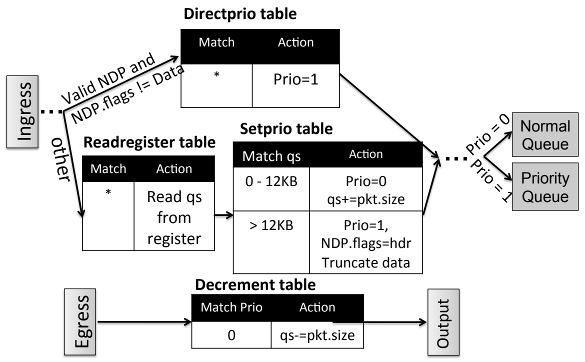


Figure 7: NDP switch implementation in P4.

to trim the last packet in the packet queue or the current packet, to break up phase effects.

The NDP switch uses 63561 LUTs (using 14.6% of the Virtex7's capacity), 77176 FlipFlops (8.9%) and 231 blocks of RAM (15.7%). In comparison, the reference L2 switch uses 11.4%, 8.1% and 13.2% respectively, so the complexity added by NDP is small. Of the additional resources, 80% is used for the priority output queues.

NDP Switch implementation in P4. The design, shown in Fig. 7, assumes the existence of at least two queues between the ingress and egress pipelines, with the `egress_priority` metadata deciding which packet goes into which queue. The NDP modifications are demonstrated on the simple switch device assuming a single output interface, but they could be added to any P4 switch and easily modified to handle multiple output ports.

The implementation needs to know the size of the two per-port queues to decide whether the packet should be trimmed. To this end, it could leverage current queue size registers to make the decision of whether to send packets to the priority queue or not; however not all P4 platforms will have this register, so we have chosen to implement a register with a similar functionality by counting all packets that go into the normal buffer and packets that enter the egress pipeline. As Match/Action tables in P4 only match on packet data, we use an additional table (*Readregister*) to read `qs` from the register and save it as packet metadata. If `qs` is below the allowed buffer size, packets will go in the normal queue. Once we hit the threshold, packets will be truncated (using a P4 primitive action called `truncate`) and fed into the priority queue. NDP packets without a data payload automatically enter the priority queue, due to the *Directprio* table. The egress pipeline only handles queue size book-keeping: the `qs` register is decreased if the packet came from the normal queue.

5 EVALUATION

We wish to understand how NDP will perform in large datacenters with real workloads, but lack the ability to run such tests at this time. Instead we profile our Linux NDP implementation using the NetFPGA NDP switch to understand small scale performance and how NDP interacts with the host OS. Next, we compare the Linux implementation with our simulator to determine the extent to which real-world artifacts impact the faithfulness of simulation. Finally we evaluate NDP in simulation to examine how it scales. We use a wide range of traffic patterns and scenarios, and compare its behavior to

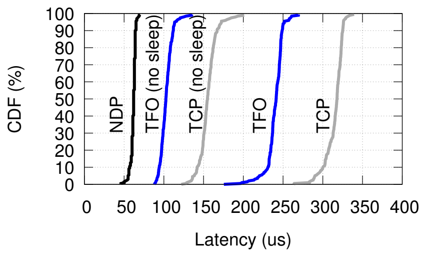


Figure 8: Time to perform a 1KB RPC over NDP, TCP Fast Open & TCP.

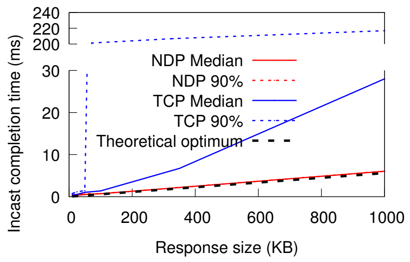


Figure 9: Seven-to-one incast with varying file sizes in our testbed.

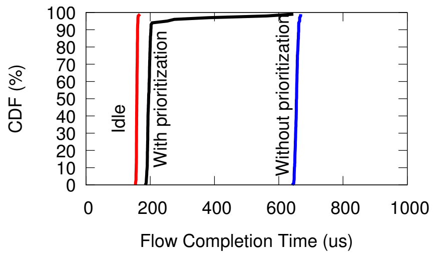


Figure 10: Prioritizing a short flow over six long flows to the same host.

MPTCP, DCTCP using ECN in the switches, and DCQCN[40] using lossless Ethernet.

We also ran tests with our P4 NDP switch using the reference P4 switch to verify its correctness; we omit results from these runs due to the poor performance of the reference P4 software switch.

5.1 Linux NDP Performance

NDP aims to provide a low-latency transport. To evaluate the latency of our Linux implementation with no confounding factors, we connected two servers back-to-back, and ran an application that makes repeated RPC calls, sending and receiving 1KB of data to measure the application-level delay. We compare our NDP prototype to Linux kernel TCP and to TCP Fast Open (TFO), a TCP optimization proposed by Google that allows sending data on TCP SYNs. Note that TFO does not guarantee that connections are processed only once, whereas NDP does.

As Figure 8 shows, median latency using NDP is 62 μs. TCP Fast Open takes four times longer and regular TCP takes five times longer. NDP RPC latency is comparable to RDMA latency reported in recent works ([20], Fig. 6).

How does NDP achieve such low latency? To understand, we also implemented a simple *ping* application over DPDK, and measured latency with 1KB pings. It takes only 22μs to send a ping using DPDK and get the reply. This implies that NDP protocol processing and application processing (40 μs) dominate the NDP RPC time. NDP relies on DPDK, dedicating a core to packet processing, with NIC buffers mapped directly to user space. In contrast, TCP uses interrupts and also copies data between kernel and user space.

To understand the gains achieved by NDP, we initially examined the cost of interupts and packet copies, but these overheads were at most 50μs, so do not explain the large differences we observe, especially compared to TFO. After further examination, we found that deep CPU sleep states were to blame for most of the difference: as both TFO and TCP rely on interrupts, the CPU goes into deep sleep and it takes roughly 160 μs to wake up, severely inflating latency. We disable sleep states deeper than C1 and plot the latency of TFO and TCP in Fig. 8. TFO and TCP now do better, but NDP's latency is still just over half that of TFO and a third that of TCP. In principle, TFO could be further optimized to obtain similar performance to NDP if it spun polling for data rather than using interrupts.

Incast. The most difficult traffic pattern for a transport protocol to handle with low latency is incast. We will evaluate small incasts using our Linux stack and NetFPGA switches, and larger incasts in simulation. Our testbed is a 8-server two-tier FatTree topology constructed using six four-port NetFPGA NDP switches.

As our first experiment, we ran a 7 to 1 incast: the frontend application on one server makes simultaneous requests to the other seven servers, which immediately reply. Traffic will concentrate on many switches in the topology, potentially leading to unfairness for the more remote servers. We vary the size of the incast responses, and measure the completion time of the last flow. Figure 9 shows the median and 90th percentile completion times for TCP and NDP for response sizes between 10KB and 1MB. NDP is within 5% of the theoretical optimal completion time, and the 90% percentile for NDP is within 10% of the median; the two lines overlap in the figure.

TCP's median flow completion time also grows linearly with response size, but NDP is four times faster. TCP's median flows do not suffer timeouts, recovering from loss using fast retransmission. There are several other reasons why TCP is slower, including the three-way handshake, interrupt delays, stack processing, additional data copies, process scheduling and sub-optimal congestion responses. These all add up, hurting response time. TCP's 90th percentile is dominated by retransmit timeouts, as MinRTO is 200ms in Linux. This impact might be reduced by lowering MinRTO and disabling delayed ACKs to avoid them triggering spurious retransmissions.

Benefits of prioritization. NDP's pull-based design gives it good control over flow completion times, especially in the common case where the destination link is the bottleneck; this sets NDP apart from most existing solutions where the transport is built around sender-based congestion control.

We examine a case where a host receives a short flow from one sender and long flows from six other senders; all flows start simultaneously, so there is a great deal of contention especially for the last hop in the first round trip time. By default, NDP will pace all senders to 1/7 of the link. In this case, however, the receiver prioritizes the short flow by sending its pulls before those of the long flows.

Fig. 10 shows the results for when the short flow sends 200KB: we measure the flow completion time (FCT) when there is no competing traffic (labelled "idle"), and compare against the FCT when competing with the six other senders, both with and without prioritization. Prioritization works remarkably well: the FCT of the short flow increases by only 50μs when competing with the long flows when priority is used, compared to 500μs when it is not. We tested flow sizes ranging from 10KB to 1MB: in all cases, the difference between idle and priority was under 50μs. After the first RTT, during which a few tens of packets from the long flows are also delivered, the short flow fully occupies the receiver's link until it finishes.

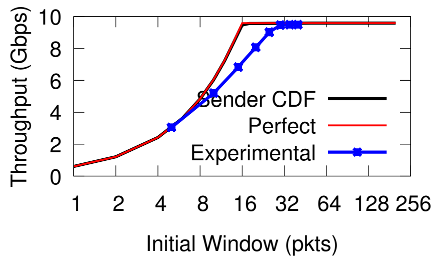


Figure 11: Throughput as a function of *IW* in simulation and practice

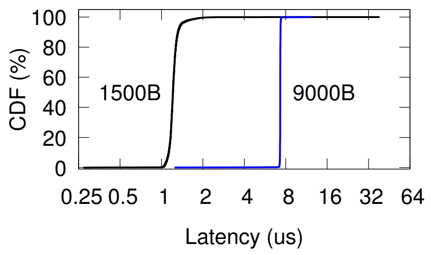


Figure 12: Pull spacing measured at the sender for different packet sizes.

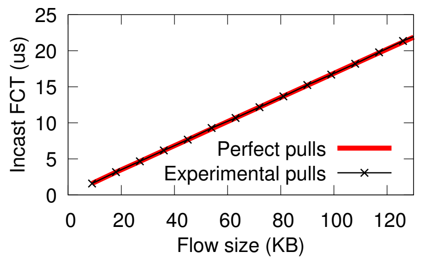


Figure 13: Incast performance: perfect versus measured pull spacing.

6 SIMULATION

Before looking at large scale experiments, we examine how the behavior of our simulator diverges from that of our Linux implementation in the same setting. The main differences we identified are host processing delays, and the pacing of PULLs. While our simulator perfectly paces PULLs, our implementation can not.

To understand the effect of processing delays, we connected two servers back-to-back and measuring the achieved throughput as a function of the initial window (IW). Fig. 11 shows that to achieve similar throughput the initial window of the prototype must be 25 packets instead of 15 (i.e. 15KB larger); these extra packets are buffered in the end systems, and are needed to cover host processing delays not modelled by the simulator. This implies that NDP latency results from simulation are slightly over-optimistic. However, unmodelled processing delays for TCP are higher, so any comparison from simulations is slightly biased in favor of TCP.

Fig. 12 shows the actual spacing of PULLs for both 1500B and 9000B packets, as measured by the sender. While the median values match the target spacing ($1.2\mu s$ and $7.2\mu s$ respectively), there is some variance with 1500B packets.

To observe the effect of this imperfect pull spacing, we added code to the simulator that draws pull spacing intervals from the experimentally measured distribution. First, we re-ran the pairwise transfer in Fig. 11 above; the result labelled "Sender CDF" overlays the "Perfect" curve: pull spacing does not affect throughput because the window is large enough to cover small "gaps" in PULLs.

Next, we ran a permutation experiment, where each node in a 432 node FatTree sends to one node and receives from another node, fully loading the datacenter. We used 1500B packets and compared perfect pull spacing to pull spacing from our experimental results. The difference in throughput is 1.2%, which is negligible.

Finally, we ran a 200:1 incast experiment varying incast flow size and measuring completion time of the last flow. Fig. 13 shows, there is no discernible difference in the flow completion times when imperfect pull spacing is used. Together, these experiments show that real-world artifacts have minimal impact on NDP and provide confidence in predictions based on large-scale simulation results.

6.1 Comparisons to existing works

We measure the ability of NDP to utilize Clos datacenter networks by running a permutation: this is a worst-case traffic matrix where each server opens a single long-running connection to another random server such that each server has exactly one incoming connection. We compare NDP with Multipath TCP [31], DCTCP and the newly proposed DCQCN protocol which is, essentially, a way of running DCTCP over lossless Ethernet networks. NDP switches use 8-packet output queues whereas, to ensure good performance, DCTCP and MPTCP use 200 packet output queues and DCQCN uses 200 buffers per port, shared between interfaces. DCTCP and DCQCN marking thresholds are 30 and 20 packets respectively, as recommended.

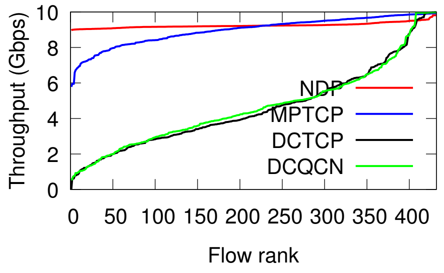


Figure 14 Per-flow throughput, permutation traffic matrix, 432-node FatTree.

Fig. 14 shows the throughput achieved by each host in increasing order. DCTCP and DCQCN use a single path and suffer from collisions resulting from ECMP: mean utilization is around 40%, and there are some flows that achieve less than 1Gb/s, despite there being sufficient capacity provisioned to offer every flow 10Gb/s. Multipath TCP does much better: utilization is 89%, and the worst flow achieves 6Gb/s. NDP has an utilization of 92% and offers much better fairness across the flows: even the slowest flow gets 9Gb/s.

What is the effect of buffer size on small flow completion times? We expect NDP should benefit from smaller in-network buffers, with DCTCP and DCQCN having longer flow completion times. Two nodes repeatedly exchange 90KB transfers to test latency, while all the other nodes each source four long running connections to a random destination. As there is no contention at the source or destination of the 90KB flows, this tests the effect of standing queues in the network on short flows run by otherwise idle hosts.

A CDF of short flow completion times is shown in Fig. 15. NDP's worst case latency is just twice the theoretical optimum transfer time in an idle network, and is three times lower in the median and four times lower at the 99% compared to DCTCP. The main reason for this difference is that NDP's buffers are much smaller than those achieved by DCTCP in an overloaded Fat Tree network. DCQCN has slightly worse performance than DCTCP because PAUSE frames are triggered sporadically. MPTCP achieves the worst FCT of all solutions, with a median and tail ten times larger than those of NDP, due to its greedy filling of network buffers. During these experiments NDP and MPTCP achieved 80% network utilization, wheras DCTCP and DCQCN achieve ~75%.

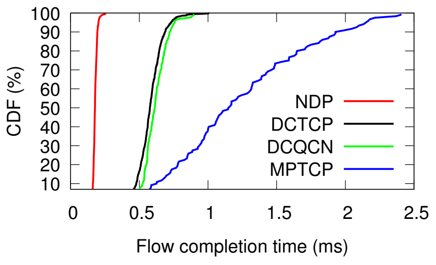


Figure 15: FCT for 90KB flows with random background load, 432 node FatTree.

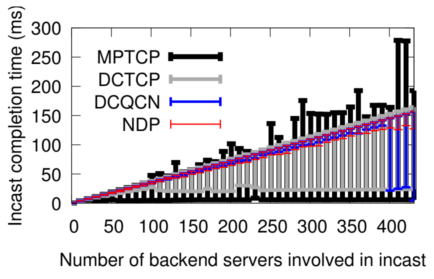


Figure 16: Incast performance vs number of senders, 432-node FatTree.

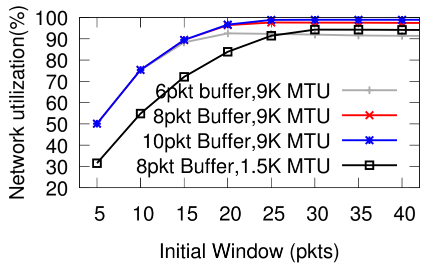


Figure 17: Effects of *IW* and switch buffer sizes on permutation throughput.

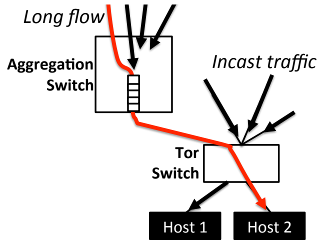


Figure 18: Experimental setup to measure collateral damage of incast on nearby flows.

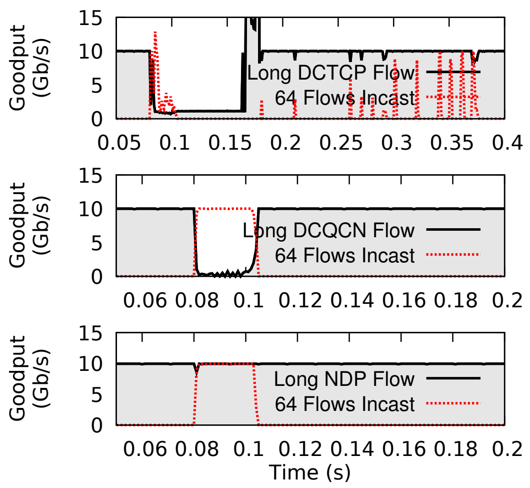


Figure 19: Collateral damage caused by 64-flow incast with DCTCP (top), DCQCN (center), NDP (bottom).

Next, we test an incast traffic pattern where a frontend fans out work to many backend servers and then receives their replies; such use cases are common in web-search applications, among others. Simultaneous arrival of responses causes tremendous buffer pressure for the switch port leading to the frontend, resulting in synchronized losses. We vary the number of backend servers while keeping the response size constant (450KB) and measure the flow completion times. In general, the last flow completion time is the metric of interest, but in Fig. 16 we also show the completion time of the fastest flow to highlight the "fairness" of the different schemes.

Even if we use aggressively small timers of Vasudevan *et al.*[38], MPTCP (and any tail-loss TCP variant) is crippled by synchronized losses leading to large and unpredictable FCTs. DCTCP uses ECN to get early feedback, as well as large shared buffer switches to absorb initial bursts, so it does significantly better than traditional TCP over tail drop switches. DCTCP is, on average, just 5% slower than the theoretical optimal. DCQCN and NDP do even better, with a completion time just 1% slower than optimal.

Next, note the spread in flow completion times for the different protocols. DCTCP has a wide range (as high as seven times) between its fastest and slowest flow. NDP has a very balanced allocation, with the slowest flow taking at most 20% longer to finish than the fastest one; this is a direct consequence of the NDP switch. Finally, DCQCN has a very tight allocation up until 350KB response size, when it operates in ECN-marking regime (with a smaller threshold than DCTCP). Beyond that, lossless operation kicks in and severely skews flow completion times.

We also enabled prioritization in NDP for a single incast sender: its pulls will be placed at the head of the pull queue of the receiver. Prioritization is very effective: the preferred flow's completion time is just 1ms with 100 incast senders, and 3.5ms with 432.

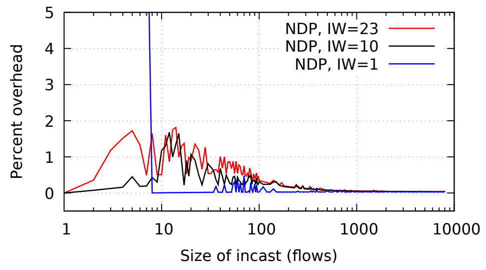
6.1.1 Side effects of Incast Traffic

How does incast affect nearby traffic? We run two separate experiments, each involving a large incast. In the first experiment the incast is long-lived and runs alongside a permutation traffic matrix. The metric of interest here is total network utilization: NDP reaches 92% and DCTCP 40% utilization, the same as the permutation running alone. With DCQCN, however, the network suffers congestion collapse: utilization drops to 17% as the incast triggers PFC, severely affecting the throughput of most flows in the datacenter.

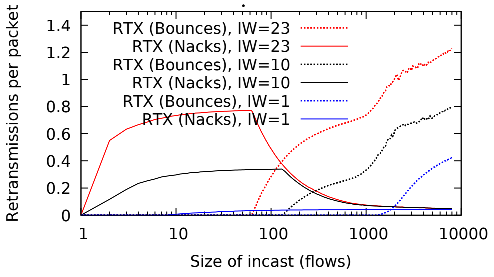
In our second experiment, shown in Fig. 18, we run one long-lived flow to host 1, then start a short-lived 64-to-1 incast traffic pattern to host 2, with each incast flow sending 900KB. Both hosts are on the same ToR switch. The results are shown in Fig. 19. With DCTCP, the incast causes loss both at the ToR switch, and at the aggregation switch port leading to the ToR. Both the long flow and the incast flows take some time to recover. The burst above 10Gb/s at t=0.17 appears when retransmissions arrive, allowing already received data to finally be released in-order to the application.

With DCQCN, loss is prevented, and the incast flows finish quickly (note the different x-axis), but PFC causes the upstream switches to be paused repeatedly, impacting the long flow. This sort of collateral damage due to pausing is the primary downside of PFC.

With NDP, incast causes trimming during the first RTT. The long flow suffers a small dip in throughput of less than 1ms due to this initial burst. After the first RTT, the receiver paces the remaining incast packets and the long flow recovers to get full throughput again.



(a) Completion time overhead, over best possible



(b) Retransmission overhead costs

Figure 20: Overhead vs size of incast, 8192-node FatTree

6.2 Sensitivity analysis

NDP uses very small switch buffers and a fixed initial window (IW) at the sender; these are the only two parameters in a NDP network. The IW incorporates information about the bottleneck speed and network latency, and will be set by administrators. We use simulation to understand how these parameters impact performance, and whether good performance requires careful tuning.

The most important metric here is throughput in the worst-case traffic matrix, the permutation. We vary the values for IW and measure the mean throughput achieved by each host for a variety of switch buffer sizes; we plot the results in figure 17. First, notice that an IW of 20 packets is needed to fully utilize the network; further, when IW is less than 15 the size of the switch buffers does not matter. Regardless the value of IW, using six packet buffers slightly underutilizes the network (90% utilization). Slightly larger queues (8 packets) result in more than 95% utilization; and this is the queue size we have used throughout this paper. Finally, notice how further increasing the IW decreases throughput slightly: this is because more buffer pressure is created that results in more headers. When using 1.5KB packets, an IW of thirty packets results in 95% network utilization. The results are remarkable, given the total size of the switch buffers in bytes is just 12KB.

For incast workloads, a smaller IW is preferable as fewer packets are trimmed in the first RTT. We used an IW of 30, both in simulation and in deployment, except where noted otherwise.

Larger topologies. Although our packet-level simulator is fast, it still takes tens of minutes to simulate a permutation matrix in a 432 node FatTree, which is the topology we used throughout our evaluation. Previous work has shown that the permutation traffic matrix has similar macroscopic behavior for 128, 1024 and 8192-node Fat Trees [31] when 100 packet buffers are used in conjunction with Multipath TCP. Are eight packet switch buffers enough to guarantee high utilization with NDP in large networks too?

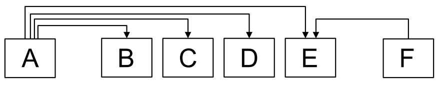
We ran a permutation experiment with increasingly larger topologies using eight packet buffers, 9KB MTU and IW of 30. Network utilization gently decreases from 98% (128 node FatTree) to 90% in a 8192 node FatTree. NDP achieves 5% more utilization than MPTCP with eight subflows ([31],Fig. 2) while using less than a tenth of its buffers.

Large scale incast Is there any fundamental limit to the size of an incast that NDP can cope with? We ran incasts ranging from one flow (no incast) up to 8000 simultaneous flows, each of 270,000 bytes (30 packets) in an 8192-node FatTree and measured the flow completion time of the last flow. Fig. 20a shows the time overhead as a percentage of the best theoretical last-flow completion time; this assumes the link to the receiver is completely saturated until the last flow finishes, and every packet is received only once. With a 23 packet IW (suggested by Fig. 11 as a good default for non-incast traffic), small incasts see the worst overheads, but still finish within 2% of optimal. These overheads are due to headers forwarded in the first RTT and due to not quite filling the receiver's link in the final RTT. For larger incasts, the time overhead is negligible. Low though these overheads are, they can be reduced further if the application knows a large incast is likely. Curves for initial windows of a ten packets and one packet are shown. Ten packets is close to the smallest window that can fill the pipe with an unloaded network and store-and-forward switches. For incasts smaller than eight flows, the overhead using a one-packet IW is high as there need to be at least eight packets sent per RTT to fill the receiver's link.

Although we care most about delay, it is also interesting to examine the cost senders pay to achieve this low delay. Fig. 20b shows the mean number of retransmissions per packet, and the mechanism (return-to-sender bounce or NACK) by which the sender was informed of the need to resend. For smaller incasts, NACKs are the main mechanism. Above 100 flows, return-to-sender becomes the main mechanism. Above 2000 flows, some packets suffer a second return-to-sender before getting through. Even with the largest incasts and a 23-packet IW, the mean number of retransmissions barely exceeds one. However, it would make sense for applications that know they will create a large incast to reduce the initial window.

Sender-limited traffic. Consider the traffic pattern in Fig. 21, where host A sends to hosts B,C,D and E, and host F also sends to host E. Under normal incast, traffic to E would be equally split between sources A and F, but in this case A cannot send enough to fill half of E's link. Do the pulls from E get wasted?

We simulated this scenario; the throughputs are listed in Fig. 21. Both the link from A and the link to E are saturated, and the four flows sourced by A divide the capacity almost perfectly. How does NDP achieve such a good outcome? The reason is that E performs fair queuing on its pull queue. At E, there are always PULL packets queued for F. However, as packets from A arrive less often, the fair queue ensures that PULL packets for A are sent as soon as arriving data packets can generate them. This ensures E sends pull packets to A at the same rate A sends data packets to E. The remainder of E's PULLs go to F, ensuring E's link remains full. B, C and D cannot send PULLs to A any faster than data packets from A arrive. Thus PULLs arrive at A roughly equally from B, C, D, and E, and this ensures A's outgoing link is both full and equally shared.



A→B	2.51 Gb/s		
A→C	2.50 Gb/s		
A→D	2.51 Gb/s		
A→E	2.38 Gb/s	Total from A:	9.90Gb/s
F→E	7.55 Gb/s	Total to E:	9.93Gb/s

Figure 21: Sender limited topology and achieved throughputs.

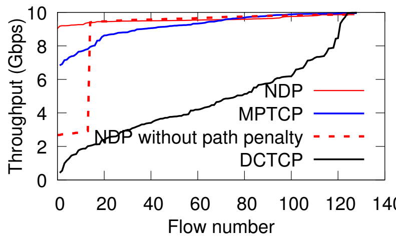


Figure 22: Permutation throughput with failures: a core switch to upper-pod link is dropped to 1Gbps.

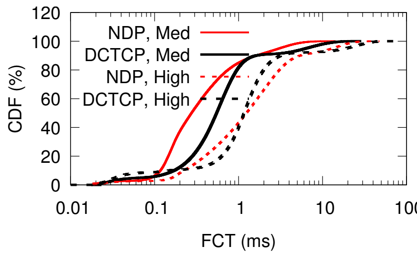


Figure 23: FCTs for the Facebook web workload in the oversubscribed topology.

Who needs packet trimming? A valid question is whether we can get the benefits of NDP without changing network switches. PHost [16] is a receiver-driven transport that runs very small packet buffers and per-packet ECMP, but does not use packet trimming. We have implemented pHost by changing TCP in htsim; pHost senders burst at line-rate in the first RTT to get the best short flow performance. To understand the merits of pHost and the added value of trimming, we ran pHost on regular drop-tail 432-node FatTree networks with eight packet buffers. The 432-to-1 incast in Fig. 16 takes pHost 1s to 1.5s to complete. Not only is this ten times worse than NDP (140ms), it is also 4-5 times worse than MPTCP. We also ran a permutation traffic matrix, finding that pHost only achieves 70% utilization despite its use of packet spraying (NDP reaches 95% utilization).

Handling asymmetry. All networks we tested so far are symmetric; per-packet ECMP work wells in such scenarios. It is more difficult to do well when networks are asymmetric, for instance due to failures. We ran a permutation experiment in a 128-node topology where the speed of one link between a core and upper pod switch is reduced to 1Gbps. The results in Fig. 22 show that both NDP and MPTCP handle the failure really well because they keep per-path congestion information and pull traffic away from congested paths. The path penalty mechanism NDP uses (see §3.2.3) is crucial in asymmetric topologies: without it NDP does quite poorly, with 15 flows reaching just 3Gbps of throughput. A few DCTCP flows are also affected: the worst hit connection only achieves 0.4Gbps.

Overload. On a fully provisioned Clos topology NDP's behaviour is near optimal. However, not all datacenters provide full cross-sectional bandwidth; what happens when the core network is over-subscribed, or where packet sizes are smaller than an MTU? In such cases, we expect NDP to trim many packets and also achieve a lower compression ratio when trimming occurs.

We use measurements from Facebook's network to setup our experiment [34]. The simulated topology is a three-tier FatTree containing 512 servers connected via 10Gbps links to ToR switch. The uplink connectivity from ToRs to aggregation switches is four times less than the server connectivity, giving 4:1 oversubscription.

Of the three types of network traffic presented in [34] (web, cache and Hadoop), we use the least favourable to NDP: the web traffic pattern has really small packets, giving poor compression, and almost no rack-level locality, meaning that almost all traffic must traverse the oversubscribed core. The flow sizes are drawn from the distribution in [34] (Fig.6.a) and flow arrivals are closed loop with a median inter-flow gap of 1ms. We vary load by increasing the number of simultaneous connections per host and measure load by examining the core utilization.

We compare NDP with DCTCP and plot the flow completion times distribution in Fig. 23. We run two separate tests: first, we run a moderately loaded network (five simultaneous connections per host) where 40% of all packets sent by NDP hosts are trimmed by the ToR switch. In this case, NDP's median FCT is half that of DCTCP's, and a third in the 99th percentile.

Next, we load the network even more, with each host sourcing ten connections simultaneously to other randomly chosen hosts. 70% of packets are trimmed at the first ToR switch; this is very close to the worst case given the core network is 4:1 oversubscribed. Despite this, NDP performs robustly, providing slightly better performance than DCTCP both in the median and the tail. NDP does not suffer from congestion collapse; we observe that if a packet makes it past the ToR switch, it is likely to reach the destination, with a probability of being trimmed of only 2-5%.

To conclude, NDP is robust even in this oversubscribed network, performing better than DCTCP. This is not to say than NDP should be used as is in massively congested networks: the number of packets trimmed in such cases is wasteful. When most packets are trimmed a simple congestion control algorithm could reduce the pull rate, avoiding persistent overload, but there is no need to be conservative as NDP works well enough with no congestion control.

7 RELATED WORK

There is a large body of work aiming to tackle various aspects of datacenter transport: most focuses on either achieving low latency [5, 6, 16, 22, 28, 30] or high throughput [2, 12, 31]. We have discussed at length and compared to the most relevant deployed previous works, namely DCTCP/ DCQCN for low delay and MPTCP for high throughput; here we overview the rest.

pFabric [6] aims for shortest-flow first scheduling at datacenter scale by switching packets based on strict priorities. While enticing in theory, pFabric is difficult to deploy as it entrusts hosts to correctly prioritize their traffic. Hull [5] reserves capacity to ensure low latency for short flows and uses phantom queues to identify potential congestion; it requires accurate packet pacing at endpoints. Fastpass [30] uses centralized scheduling to achieve low latency but is scale-limited. If one can modify applications to explicitly state deadlines, there is a wide range of proposals that will help including PDQ, Deadline-aware Datacenter TCP (D^2TCP) and D^3 [22, 37, 39].

TIMELY [28], an alternative congestion control mechanism to DCQCN for lossless networks, relies exclusively on RTT as a congestion metric. As with DCQCN, TIMELY cannot completely prevent pause frames and their negative impact on innocent network traffic. Overall, the greatest limitations of all these works is that they only focus on low latency, while ignoring network utilization or large-flow performance.

At the other end of the spectrum, researchers have been keen to tackle collisions due to per-flow ECMP with centralized scheduling in Hedera [2], via protocol changes in Packet Spraying [12] and Presto [21], or via network subflow-treatment in Conga [3] and switch-redesign of LocalFlow [35]. These works do not target short flow completion times, and obtain FCTs comparable to those of an unoptimized network running large packet buffers.

NDP stands out because it achieves both low latency and high throughput in all traffic conditions.

8 CONCLUSIONS

We presented NDP, a new architecture for datacenter networking that includes a modified switch queuing algorithm, together with per-packet multipath forwarding, and a novel transport protocol that takes advantage of these network mechanisms. NDP exhibits excellent low-latency behavior, both in our implementation, and in large scale simulations. It provides much better isolation between different workloads than mechanisms such DCQCN that rely on lossless Ethernet to achieve low delay.

Our current implementation is moderately expensive in terms of CPU resources required from end systems, because of the need for accurate pacing of PULLs and low-latency retransmissions. Both these functions would be very simple for Ethernet NICs to implement; indeed, cheap WiFi NICs already handle retransmissions and careful packet timing. Were NDP to be deployed at scale, we expect that smart NICs would greatly reduce the CPU overhead of NDP.

Acknowledgements

This work was partly funded by the SSICLOPS H2020 project (644866). We would like to thank Giuseppe Lettieri, the anonymous reviewers and our shepherd Amin Vahdat for their very useful feedback that helped us improve this paper.

REFERENCES

[1] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proc. ACM SIGCOMM*, Aug. 2010.

[2] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *Proc. Usenix NSDI*, 2010.

[3] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese. CONGA: Distributed Congestion-aware Load Balancing for Datacenters. In *Proc. ACM SIGCOMM 2014*, pages 503–514.

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

[5] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda. Less is more: trading a little bandwidth for ultra-low latency in the data center. In *Proc. Usenix NSDI*, pages 253–266, 2012.

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

[7] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 267–280. ACM, 2010.

[8] R. Braden. RFC 1644: T/TCP – TCP extensions for transactions functional specification. Technical report, RFC Editor, July 1994.

[9] P. Cheng, F. Ren, R. Shu, and C. Lin. Catch the whole lot in an action: Rapid precise packet loss notification in data centers. In *Proc. Usenix NSDI*, 2014.

[10] Y. Cheng, J. Chu, S. Radhakrishnan, and A. Jain. RFC 7413: TCP fast open. Technical report, RFC Editor, Dec. 2014.

[11] J. Chu, N. Dukkipati, Y. Cheng, and M. Mathis. RFC 6928: Increasing TCP's initial window. Technical report, RFC Editor, Apr. 2013.

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

[13] DPDK Data Plane Development Kit. http://dpdk.org. Accessed: 2017-01-27.

[14] S. Floyd and V. Jacobson. Traffic phase effects in packet-switched gateways. *SIGCOMM Comput. Commun. Rev.*, 21(2):26–42, Apr. 1991.

[15] S. Floyd and J. Kempf. RFC 3714: IAB concerns regarding congestion control for voice traffic in the internet. Technical report, RFC Editor, Mar. 2004.

[16] P. X. Gao, A. Narayan, G. Kumar, R. Agarwal, S. Ratnasamy, and S. Shenker. pHost: Distributed Near-optimal Datacenter Transport Over Commodity Network Fabric. In *Proc. ACM CoNEXT*, 2015.

[17] A. Greenberg el al. VL2: a scalable and flexible data center network. In *Proc. ACM SIGCOMM*, Aug. 2009.

[18] R. Griffith, Y. Chen, J. Liu, A. Joseph, and R. Katz. Understanding TCP incast throughput collapse in datacenter networks. In *Proc. WREN Workshop*, 2009.

[19] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu. Bcube: A high performance, server-centric network architecture for modular data centers. In *Proc. ACM SIGCOMM 2009*.

[20] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn. Rdma over commodity ethernet at scale. In *Proc. ACM SIGCOMM 2016*, pages 202–215.

[21] K. He, E. Rozner, K. Agarwal, W. Felter, J. Carter, and A. Akella. Presto: Edge-based load balancing for fast datacenter networks. In *Proc. ACM SIGCOMM 2015*, pages 465–478.

[22] C.-Y. Hong, M. Caesar, and P. B. Godfrey. Finishing flows quickly with preemptive scheduling. In *Proc. ACM SIGCOMM 2012*.

[23] IEEE DCB. 802.3bd - MAC Control Frame for Priority-based Flow Control Project. *http://www.ieee802.org/3/bd/*, 2010. Superseding IEEE 802.3x Full Duplex and Flow Control.

[24] IEEE DCB. 802.1Qbb - Priority-based Flow Control. *http://www.ieee802.org/1/pages/802.1bb.html*, 2011.

[25] Infiniband Trade Association. RoCEv2. *https://cw.infinibandta.org/document/dl/7781*, Sept. 2014.

[26] V. Jacobson and M. J. Karels. Congestion avoidance and control. In *Proc. ACM SIGCOMM*, Stanford, CA, Aug. 1988.

[27] C. Kent and J. Mogul. Fragmentation considered harmful. In *Proc. ACM SIGCOMM*, Aug. 1987.

[28] R. Mittal, V. T. Lam, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, and D. Zats. Timely: Rtt-based congestion control for the datacenter. In *Proce. ACM SIGCOMM 2015*, pages 537–550.

[29] The P4 Language Consortium. P4$_{16}$ language specification version 1.0.0. 2016.

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

[31] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving datacenter performance and robustness with Multipath TCP. In *Proc. ACM SIGCOMM*, Aug. 2011.

[32] K. Ramakrishnan, S. Floyd, and D. Black. RFC 3168: the addition of explicit congestion notification (ECN) to IP. Technical report, RFC Editor, Sept. 2001.

[33] A. Romanow and S. Floyd. Dynamics of TCP traffic over ATM networks. In *Proc. ACM SIGCOMM*, London, 1994.

[34] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the social network's (datacenter) network. In *Proc. ACM SIGCOMM 2015*, pages 123–137.

[35] S. Sen, D. Shue, S. Ihm, and M. J. Freedman. Scalable, optimal flow routing in datacenters via local link balancing. In *Proc. ACM CoNEXT 2013*, pages 151–162.

[36] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey. Jellyfish: Networking data centers randomly. In *Proc. Usenix NSDI 2012*.

[37] B. Vamanan, J. Hasan, and T. Vijaykumar. Deadline-aware datacenter tcp (d2tcp). *ACM SIGCOMM Computer Communication Review*, 42(4):115–126, 2012.

[38] 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 *Proc.ACM SIGCOMM 2009*, pages 303–314.

[39] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron. Better never than late: Meeting deadlines in datacenter networks. In *Proc. SIGCOMM '11*, 2011.

[40] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang. Congestion control for large-scale rdma deployments. In *Proc. ACM SIGCOMM 2015*, pages 523–536.

[41] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore. NetFPGA SUME: Toward 100 Gbps as research commodity. *Micro*, 34(5), 2014.