# An edge-queued datagram service for all datacenter traffic

*Vladimir Olteanu* [*‡], *Haggai Eran*[†#], *Dragos Dumitrescu*[*‡], *Adrian Popa*[*], *Cristi Baciu*[*],
*Mark Silberstein*[†], *Georgios Nikolaidis*[△], *Mark Handley*[○*], *Costin Raiciu*[*‡]
[*] Correct Networks, [†] Technion, [○] UCL, [△] Intel, [#] NVIDIA, [‡] University Politehnica of Bucharest

## Abstract

Modern datacenters support a wide range of protocols and in-network switch enhancements aimed at improving performance. Unfortunately, the resulting protocols often do not coexist gracefully because they inevitably interact via queuing in the network. In this paper we describe EQDS, a new datagram service for datacenters that moves almost all of the queuing out of the core network and into the sending host. This enables it to support multiple (conflicting) higher layer protocols, while only sending packets into the network according to any receiver-driven credit scheme. EQDS can transparently speed up legacy TCP and RDMA stacks, and enables transport protocol evolution, while benefiting from future switch enhancements without needing to modify higher layer stacks. We show through simulation and multiple implementations that EQDS can reduce FCT of legacy TCP by 2x, improve the NVMeOF-RDMA throughput by 30%, and safely run TCP alongside RDMA on the same network.

## 1 Introduction

Data center networks suffer from a range of unique problems that make it hard to effectively utilize the potential of the underlying high performance redundant multipath network topology. Notable issues include incast traffic patterns, flow collisions and transient congestion due to flow-level load balancing, interference between low-latency request/response traffic and bulk transfers, increasing requirements to offload work from the host CPU to avoid host stack bottlenecks, and the need to support special-purpose high performance protocols such as RDMA in the same network as legacy protocols.

These are all partially solved problems. There is a strong trend towards NIC offload, with datacenters deploying smart NICs and increasing ASIC support for specific transport protocols being offered by NIC vendors. However, moving transport state into NICs makes it harder for dissimilar protocols to coexist, and risks embodying the status quo in hardware.

At the same time, the research community has proposed a rich set of solutions such as phost[14], Homa[31], NDP [18], 1RMA [42] and Aeolus[20] which tackle incast and, to varying degrees, also address issues of load-balancing and low-latency request/response traffic. What these solutions share is a receiver-driven control loop that tightly manages inbound traffic, eliminating large in-network queues. Each of these, by itself, would be a substantial improvement on the status quo, but datacenters cannot simply migrate to a single new transport protocol. Even if there were buy-in as to which transport protocol to adopt, there are far too many legacy applications and operating systems that would need to be re-written. How then can we take the best ideas from the research community and deploy them in production while supporting a plethora of legacy protocols ranging from vanilla TCP to RDMA?

One strawman solution would be to simply pick a low latency receiver-driven transport protocol and tunnel all datacenter traffic over it. The great advantage of such a control loop is that it performs admission control to the physical network, allowing very small switch buffers to be used while still providing low end-to-end loss. Is it possible to use this to provide a new datagram service that higher layer protocols such as TCP, DCTCP or RDMA run over?

The difficulty is that TCP, DCTCP, RDMA and other protocols each have their own expectations when it comes to sharing the underlying network. In particular, they use interactions between flows mediated via queues in the switches to drive their own control loops. We cannot just eliminate switch queues and expect everything to still work - rather we need to move the queuing from the switches back to the network edge, either in the host or NIC. The low-latency control loop can then clock packets from these edge queues into the network.

We have designed and implemented just such a layer called Edge-Queued Datagram Service (EQDS). Rather than a regular transport protocol, EQDS provides a datagram service to higher layers, implemented via dynamic tunnels. Its receiver-driven control loop is loosely based on NDP and 1RMA[42], but can be extended to utilize other in-network mechanisms where these are available.

Moving the interaction between flows out of the switch queues and back into EQDS edge queues provides many advantages. The receiver directly controls when enqueued packets from different senders enter the network, ensuring isolation even when higher layer protocols run different control loops. For example, TCP and RDMA will not normally coexist gracefully when sending to the same host, but EQDS can mediate, eliminating loss and allowing fair sharing.

Different EQDS queuing disciplines can be also used for different protocols, each providing appropriate feedback to the higher layer control loop; this both improves higher layer protocol performance, and it also allows dissimilar protocols sending from the same host to be protected from each other.

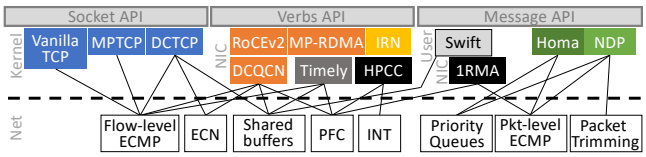


Figure 1: Fragmentation of datacenter networking

Adding new queuing disciplines to support innovative future transport mechanisms is also simplified as they only need to be deployed in the relevant sending hosts, not in switches.

Finally, EQDS uses packet spraying to balance load evenly in the network core, avoiding flow collisions, and increasing throughput. Legacy protocols such as TCP and RDMA do not normally cope well with reordering, so EQDS implements a reorder buffer in the receiver. With a conventional network such a reorder buffer might deliver the highest latency seen across all paths, but with good load balancing and short switch queues we find that EQDS reorder buffer latency is minimal.

In this paper we detail the design and implementation of EQDS and its on-demand zero-RTT tunnel protocol, and evaluate it running both natively in Linux hosts and offloaded to two brands of smart NIC. We show that the EQDS control loop operating on very short timescales does not adversely affect higher layer control loops such as TCP's Cubic or RDMA using DCQCN. Rather, it allows diverse higher-layer control loops to co-exist gracefully, protects latency-sensitive applications from queuing delays caused by bulk transfers, while increasing throughput by eliminating flow collisions.

## 2 Motivation

IP has been the narrow waist[7] of the Internet stack since from the early days of the Internet, providing basic end-to-end service. In reality, the narrow waist is not just IP: a functioning Internet also assumes some form of TCP-compatible congestion control and sufficient in-network queuing for it to do its job, though this lacks a clear layer in the stack.

This lack of abstraction has particularly hurt datacenter networking. Here, a plethora of work has pushed optimizations across boundaries, including to the host stack, switches or both. As a result of all these enhancements, what has emerged are multiple parallel stacks, each assuming a slightly different "basic" datagram service, that must be isolated from each other in the network to avoid them fighting (see Fig. 1). Further, optimizations for one stack often hurt the performance of others in the same network.

Many have improved on TCP congestion control[1, 28, 44], reducing vanilla TCP's need for large switch buffers. These TCP's are still built upon basic datagram service though, and probe network capacity to sense congestion. In so doing they interact with each other via queues, increasing latency.

Even datagram service itself has been tweaked, with many enhancements aimed at improving service for certain traffic classes, as in Fig. 1. For example, RoCEv2 can use PFC to provide lossless service as assumed by RDMA; this brings its own set of unique feature interaction problems [16, 29].

Protocols like TCP and RDMA also assume largely in-order delivery from the underlying datagram service. In datacenters, in-order delivery is provided by flow-level ECMP, though this wastes capacity in Clos topologies. To better use multipath networks, variants of these protocols have been proposed [38, 26, 29] but rarely deployed. Another source of performance problems as network speeds have increased has been the end-host stack implementation itself. Even TCP resorts to segmentation and checksum offloading, but application writers often use kernel bypass mechanisms such as DPDK or even offload all the work to the NIC using RDMA, and in so doing impose unique dynamic load on the network.

The web of dependencies between higher layer protocols and in-network enhancements makes deploying new protocols increasingly difficult. The root cause of the problem is that basic datagram service forces diverse higher layer protocols to interact via queues in the network. We argue that in-network queuing, beyond the minimum needed to smooth fan-in, is antithetical to building a high performance low-latency general-purpose datacenter network.

We propose a novel Edge-Queued Datagram Service as the new narrow waist for the datacenter networking stack. To transport stacks above, EQDS offers a what looks like a conventional datagram service via virtual interface queues in the host that buffer traffic and provide appropriate congestion feedback signals. EQDS then sends this traffic when possible, utilizing diverse in-network mechanisms to maximize utilization and minimize in-network queuing latency. EQDS shares the network at the hosts, allowing conflicting transports to run side-by-side on the same network.

## 3 Concept

We introduce the EQDS concept by means of example. Consider Figure 2a: two TCP senders send to a receiver across a conventional network where the bottleneck is at the final hop as is common in datacenters [5]. TCP needs to build a queue to sense congestion and back off, forcing any other flow sharing the queue to behave similarly to share the link reasonably. RDMA or other transports (see Figure 1) that do not will cause problems and need to be isolated from TCP.

In contrast, the EQDS concept is shown in Figure 2b. The underlying latency across modern datacenter networks is so low that protocols like NDP, Homa and Aeolus can use credit mechanisms whereby the receiver clocks packets from the sender as required, ensuring a standing queue never builds in the network. Protocols like TCP still need a queue to drive their control loop, but EQDS moves this queue to the sending hosts, where it is under the receiver's control. If many TCP senders create an incast, the default behavior they observe is almost identical to what would happen if the last hop switch runs fair queuing with a large amount of buffering.

As the queuing has been removed from the network itself, EQDS can run different queuing disciplines in the sending

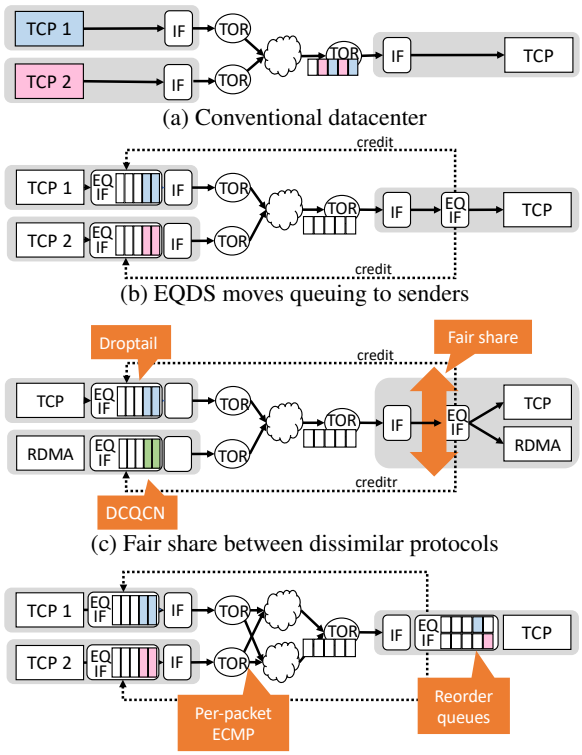


Figure 2: Overall EQDS concept

*EQDS Virtual Interface* (EQIF), as appropriate to the traffic being carried. Figure 2c shows a legacy TCP sender coexisting with an RDMA sender, with the bottleneck link being shared fairly, or with a proportional share, if that is deemed appropriate by the network or host administration policy.

Finally, Figure 2d shows EQDS using per-packet ECMP (aka packet spraying) in the underlying network to minimize latency and increase network capacity. Few current transport protocols cope well with the level of reordering this usually causes, but as EQDS keeps network queues very small, the reordering is easily managed by a short reorder queue in the receiving EQIF, so it is hidden from the higher level protocol.

In summary, the key EQDS concepts are: (1) move queuing out of the network leaving just the bare minimum required; (2) queue traffic in the sending host; release it when the receiver requests it; (3) run appropriate queue disciplines for different classes of application as they require; and (4) use per-packet ECMP to load-balance evenly so as to minimize latency but, by default, hide it from higher layer protocols. The EQIF virtual interfaces then become the control points for the network, enforcing sharing policies.

# 4 Design

To implement EQDS, we need four main components:

- One or more EQIFs on the sending host, which implement queuing disciplines to support higher level protocols;
- One EQIF on the receiving host, which implements a short reorder queue for best-effort in-order delivery service to protocols that are intolerant to reordering;
- A mechanism to encapsulate packets reliably across the network from sending EQIF to receiving EQIF;
- An edge-to-edge control loop to clock packets from sending EQIF to receiving EQIF.

With these in place diverse higher-layer protocols are carried over EQDS, which hides lower-layer in-network mechanisms. The edge-to-edge control loop may differ in different datacenter environments or even within one datacenter, depending on switch capabilities. In effect, EQDS has become the new narrow waist of the datacenter protocol stack.

In the virtualized protocol stack, EQDS operates at the same layer as VXLAN, encapsulating higher-layer traffic going to EQDS-capable destinations. To provide datagram service, EQDS needs to provide on-demand tunneling from EQIF to EQIF without prior setup, without spending an RTT performing a handshake to establish control state, and with the expectation that packets sent in the first RTT will be reordered by per-packet ECMP. This demands a novel tunnel protocol (§5). EQDS tunnels are unidirectional; two are setup, one in each direction, if user traffic is bidirectional.

EQIF per-destination tunnel state is established on packet receipt at the sender, and established at the receiver using a zero-RTT protocol. This state can be unilaterally discarded when idle at any time by either side to reduce memory usage and will simply be reestablished as needed.

## 4.1 EQDS control loop

Critical to EQDS performance and minimizing in-network queuing is the edge-to-edge control loop. EQDS allows different control-loop mechanisms to be used depending on the underlying network capabilities.

For a fully provisioned network our preferred in-network mechanism is packet trimming, which allows EQDS to use an NDP-derived control loop. This allows a burst of packets to be sent in the first RTT before credit-based control from the receiver takes over for subsequent RTTs. The sending EQIF keeps packets until they have been acknowledged by the receiving EQIF, or retransmits them on receipt of a NACK. In this manner, EQDS provides a highly reliable service, but it does not guarantee no packet loss whatsoever. A full reliability guarantee would prevent EQDS managing its own state effectively, risk resource starvation attacks, and would be pointless as full reliability requires end-to-end acknowledgment whereas EQDS may be implemented in the NIC so cannot protect data all the way to the receiving process.

Where packet trimming is unavailable, EQDS uses a 1RMA[42]-derived mechanism where the sending EQIF requests credit from the receiver, or it can use a Homa/Aeolus-derived mechanism where the first RTT of data is sent using

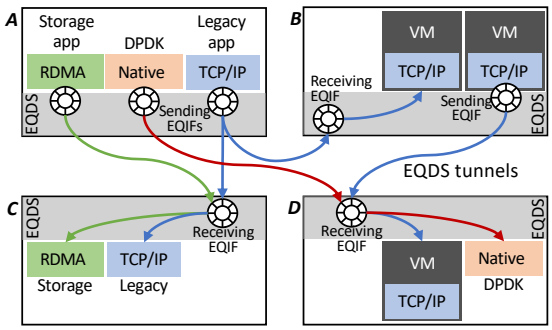


Figure 3: Multiple stacks run on the same substrate.

low-priority service. In an underprovisioned network, core network congestion is also possible, though it should be rare if per-packet ECMP is used. In such cases, in-band network telemetry (INT) might by used to implement an HPCC-style control loop. Our current implementation supports both NDP-style pro-active transmission and request-to-send. We expect future networks to innovate further in this area.

**EQIFs.** EQDS allows multiple protocol stacks to run on the same network substrate, as shown in Figure 3. The DPDK-based stack on host *A* is sending to its peer at *D* while a virtual machine at *B* also uses TCP to send to the VM at *D*. Without EQDS, the DPDK stack will saturate the link to *D*, starving TCP. This could be prevented using fair queuing in *D*'s ToR switch, but EQDS achieves the same effect without needing to configure switches and with finer grain control.

At *D*, a single *receiving EQIF* receives the flows from *A* and *B*. It maintains state (reorder queues, sequence numbers, etc) for incoming EQDS tunnels, allowing it to effectively manage all incoming traffic. By default, *D*'s receiving EQIF will send equal credits to *A* and *B*, ensuring a fair share and no overload. Proportional sharing or strict priority can also be achieved: the inbound sharing policy can be configured as needed - both by the network administrator and, if the VM and native stacks are run by the same user, by them too.

Three different stacks are in use at *A*: there are RDMA and TCP flows to *C*, a TCP flow to a VM at *B* and the DPDK native flow to *D*. This results in three *sending EQIF* virtual interfaces at *A* for the three different edge queue disciplines. The two TCP flows share the same sending EQIF which runs a TCP-compatible queue discipline. All three EQIFs at *A* cooperate using deficit round robin, so if the total traffic saturates *A*'s outgoing link, they will share it fairly (or unfairly, if that is the configured policy).

To summarize: an EQIF is a virtual interface that implements a specific queue discipline or feedback mechanism to higher-layer protocols. One sending EQIF contains multiple queues, each feeding an EQDS tunnel to a single host. Multiple transport flows from multiple VMs can share one EQIF so long as the protocols used can coexist in the same queue.

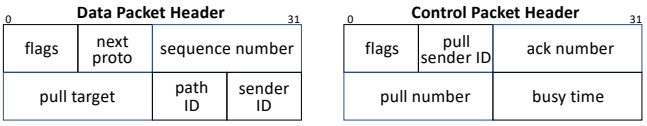


Figure 4: Tunnel Headers for Data / Control Packets

## 5 Tunnel protocol

To carry data from sending EQIF to receiving EQIF we need a new tunnel protocol. The primary requirements are:

- A receiving EQIF should clock packets from its set of sending EQIFs so as not to cause in-network queues to build. At the least, this means it will send credit at a rate that does not exceed the receiver's access link speed.
- A receiving EQIF can choose how to distribute credits to senders, with the default being to implement a fair share.
- The tunnel should expect per-packet ECMP service and be robust to reordering this causes. Where possible, the sending EQIF will determine the path taken by each packet.
- The tunnel should provide best-effort reliable and in-order delivery from the receiving EQIF to higher layer protocols. Losses and reordering should be rare enough to minimally impact the performance of higher layer protocols.
- The tunnel should support unreliable, out-of-order delivery to higher layer protocols that prefer minimal latency.
- The tunnel should come up on demand with no pre-data handshake required, and be discardable at any time. If the endpoints end up out of sync, it should self-synchronize.
- The tunnel should be able to take advantage of a range of underlying network enhancements without higher layer protocols needing to be aware of them.
- Both sending and receiving EQIFs should be able to impose policies for sharing, configured by the network operator and by the users (to the extent user policy does not conflict with operator policy).
- EQIFs must be capable of being implemented in fast NIC hardware with bounded memory resources.

These requirements necessitate an unusual tunnel protocol; it has many aspects of a transport protocol, but is soft-state, being established, dropped and reestablished based on packet arrivals, yet it provides fine-grain closed-loop control.

At a basic design level, EQDS is a tunneling protocol with an NDP-derived control loop, that runs on top of UDP. It can carry multiple types of traffic, including IP and VXLAN. Its control fields (shown in figure 4) were meant to complement VXLAN and there is no overlap in functionality between the two; indeed, EQDS could be implemented as a stateful extension to VXLAN encapsulation if required [1].

**Data Clocking.** As with NDP, the receiving EQIF sends PULL packets containing credit to the sending EQIF; the

[1] For testing, we encapsulated plain IP traffic, rather than VXLAN.

sending EQIF then sends data packets matching that credit from the corresponding tunnel queue in response. The receiving EQIF paces the sending of credit so that after the first RTT the aggregate arrival rate matches the incoming link speed.

To summarize NDP as described in [18]: when a sender starts, one RTT of data is sent without waiting for credit; after that, the sender waits for PULL packets from the receiver. This means that there can be an incast in the first RTT where loss occurs. NDP copes with this using *packet trimming* - the payload is removed from packets that would overflow the queue, and just the header is forwarded to the receiver. This makes the network lossless for metadata, though not for data, and informs the receiver of demand. The receiver can then request retransmission of trimmed data and transmission of new data using PULL packets. In a large incast it may take some time to send a PULL to a sender, so the receiver ACKs or NACKs each data packet so that the sender knows which packet buffers to free and which to add to its retransmit queue.

Where the underlying network supports trimming, an EQDS tunnel uses the NDP mechanism as described above for the first RTT, as it minimizes latency. EQDS supplements this with a request-to-send mechanism, where the sender directly requests credit from the receiver. This is used when trimming is unavailable and for intermittent bursty flows.

Conceptually, each EQDS tunnel maintains a constant bandwidth-delay product (BDP) of credit which is passed between sender and receiver. This credit either starts at the sender (NDP-like) or at the receiver (RTS). Credit flows from sender to receiver with data packets and from receiver to sender with PULL packets. EQDS differs from NDP in how it keeps this window constant in the presence of control packet loss, as NDP failed to do so in corner cases.

EQDS credit is expressed in bytes. To send a packet of size $b$ bytes, the sending EQIF must possess $b$ bytes of credit. PULL packets contain a *pull number* which starts at zero and increments for each PULL sent to a source. When the highest pull number seen by the sending EQIF increases by $n$, this grants $n$ MTUs of credit.

When a sending EQIF sends data, a *pull target* field in the header indicates to the receiver how much credit is desired beyond the current pull number. This is capped at one bandwidth-delay product (BDP) - typically 10 to 30 packets.

The receiving EQIF maintains an *active sender list* (ASL). An active sender is an incoming EQDS tunnel that has outstanding data to send. Every MTU-time the receiving EQIF will send one MTU of credit to the sender at the head of the ASL. If this causes the pull number to reach that sender's pull target, this credit will satisfy all the known demand from that sender. The sender will then be removed from the ASL and placed in an inactive senders set. If the pull number does not reach the pull target, the credit sent will not yet be sufficient, so the sender is re-inserted at the tail of the ASL. In this way all active senders get a fair share of capacity and credit is not sent to sending EQIFs that have no queued data.

The ASL is similar in concept to the NDP pull queue, but unlike NDP it ensures that a one-BDP credit window invariant always holds. Conceptually, the sum of credit stored at the sender, packets in flight, pulls in flight carrying credit, and credit implicitly stored in the ASL entry at the receiver is a constant so long as sufficient demand remains.

Implementing the ASL as a FIFO ensures incoming traffic is split fairly by default. To implement other sharing policies, a PIFO queue[43] can be used in place of a FIFO, allowing a wide range of policies to be implemented.

To avoid sources going idle and then immediately bursting again, the receiving EQIF tells senders the minimum time its access link will be saturated using the busy time field in control packets (the pull targets inform the receiver of the total queue size at every sender). Even in trimming networks, if a bursty sender restarts within this busy time, it always uses RTS before sending, as bursting would cause unnecessary trims; senders can burst after the busy time elapses.

**Tunnel setup and teardown.** A sending EQIF creates tunnel state when packets for a new destination arrive. It picks a sequence number with certain constraints (see Appendix A for details) and starts encapsulating packets without waiting for a handshake to complete, setting the SYN flag in all packets until it receives a matching SYN +ACK packet in response. This informs the receiver of the new tunnel and is robust to reordering caused by per-packet ECMP.

Either side can unilaterally drop tunnel state. As an optimization, each will inform the other when it does so, but such a teardown does not need to be signalled reliably, and neither end keeps time-wait state. Later, if new packets arrive at the sending EQIF, a new tunnel will be established. If a receiver tears down a tunnel from a sending EQIF that has queued packets, a new tunnel is immediately set up.

This simplicity allows the simple EQDS state machine in Appendix A, it allows EQDS to be self-synchronizing if the two endpoints end up in different states, and it minimizes state requirements - something that is important for EQDS implementation in hardware. We can get away with such a lightweight protocol because the EQDS service model only guarantees a best-effort attempt to avoid loss, duplication, or reordering. A conventional transport protocol like TCP needs to provide firmer guarantees to the application.

**Reorder Queue.** Per-packet ECMP greatly improves load balancing, reducing in-network queuing and latency, but may cause reordering. Trimming also causes reordering while awaiting retransmission. To avoid performance problems with higher-layer protocols, EQDS maintains a per-tunnel reorder queue in the receiving EQIF. With minimal in-network queuing, the delay difference between paths is small, so this queue does not grow much and is bounded by a BDP.

**Oversubscribed networks.** When the network core is oversubscribed and becomes a bottleneck, aggressive receiver-driven transports can result in high trim rates or in high la-

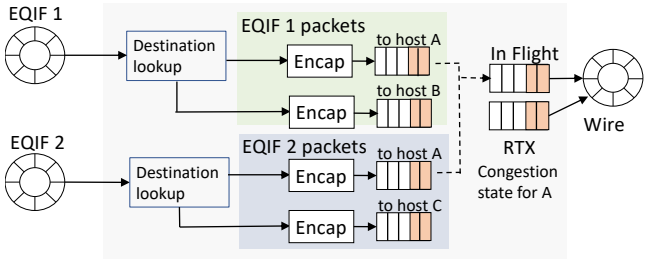


Figure 5: Transmit datapath for sending EQIFs

tency or loss when RTS is used. In such cases, EQDS will need to be enhanced to take into account other congestion signals such as ECN, latency or even in-band network telemetry[32] *in addition to* receiver pacing. As the receiver-driven transport handles the high-dynamic-range incast case, such congestion management only needs a relatively limited dynamic range and can be implemented by either the receiver reducing its pull rate or by then sender reducing its pull target. Developing such mechanisms is future work.

**Incremental deployment.** EQDS only encapsulates traffic to configured internal address ranges, so external and legacy traffic will also be present in a datacenter. How will they coexist? EQDS's packet spraying diffuses the effects of a flow across many core links, greatly reducing its impact on any legacy single-path flow. In our testbed when trimming is enabled, we use two priority classes to separate EQDS and non-EQDS traffic and ensure low latency for EQDS via small buffers even in the presence of legacy "elephant" flows.

Strict prioritization is probably undesirable as load levels rise, but weighted fair queuing between EQDS and non-EQDS traffic classes can maintain low latency for EQDS flows in the core, so long as the sprayed load-balanced EQDS aggregate does not exceed its allocated share. On ToR uplinks where EQDS traffic is less diffused, legacy "elephant" flows may impact some EQDS paths more than others. EQDS offers accurate per path latency and loss statistics that can be used to perform load-adaptive routing between paths, avoiding transient bottlenecks. Implementing these is future work.

## 6 Sending EQIF Specialization

Different types of traffic have different expectations of the underlying datacenter network. While a single EQDS tunnel protocol clocks all traffic from sending EQIF to receiving EQIF, higher-level protocols with differing network expectations are supported by different specialized sending EQIFs.

Fig. 5 shows how sending EQIF behaves as a virtual interface. Whenever a higher-layer protocol sends packets via that interface, EQDS encapsulates and enqueues them, pending sufficient credit being available. The sending EQIF maintains one queue per tunnel, allocated on demand if one does not already exist. When a packet is sent, it is moved from the send to the in-flight queue, but not freed until it is ACKed. If is it

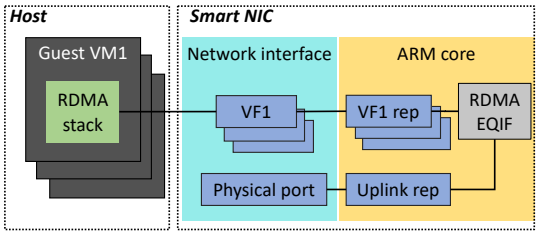


Figure 6: RDMA EQIF implementation.

NACKed or a retransmit timer expires, it is moved to the tunnel's retransmit queue. When credit is available, retransmitted packets are sent first, then new ones.

Each sending EQIF provides a specific type of service. We currently support three service classes: TCP-compatible, RDMA, and native. They differ in their queue discipline and in how feedback is presented to end-to-end traffic.

When multiple sending EQIFs are in use at the same host, as in Figure 5, credit avoids overloading the receiver's downlink, but credit from multiple receivers can exceed the uplink speed. When this happens, deficit round robin is used to share the physical interface fairly between the various EQIF queues. Queue priorities can also be configured if desired.

Multiple host stacks or virtual machines sending the same traffic class can share the same EQIF. For example, QUIC and TCP may use the same TCP-compatible EQIF, whereas RDMA would use a different EQIF. None of these legacy stacks need be aware they are running over EQDS.

**TCP-compatible EQIF Class.** Our TCP EQIF implements a simple drop-tail queue for non-ECN traffic and a RED queue for ECN-capable traffic. The goal of the queue is to absorb traffic when TCP is sending faster than the receiver wants. When the queue fills, a packet will be dropped. The queue needs to be large enough that TCP's congestion control can operate and saturate the receiver's link - typically this will be upwards of 30 packets.

The worst case for TCP is when many flows incast to the same receiver. With a default ten-packet initial window, packets will be queued in the EQIF queue until the receiver sends credit, which may take some time. TCP's 250 ms minimum RTO time helps here - even large incasts can usually complete within 250 ms. If packets are queued longer than this, an RTO may occur, but our experience is that even this has little impact on performance; TCP detects a spurious timeout via the Eiffel algorithm[37, 36], corrects its congestion window, and updates the RTO to prevent further timeouts.

We find that vanilla TCP running over EQDS almost always outperforms TCP running natively, even when not competing with incompatible flows. Much of this win comes from EQDS's use of per-packet ECMP.

**RDMA EQIF Class.** RDMA requires a separate EQIF to avoid fighting with other traffic, to avoid loss seriously impacting RDMA performance, and to provide appropriate flow control feedback to the RDMA implementation.

RDMA is typically implemented in the NIC. Ideally an RDMA EQIF implementation would be coupled with the hardware transport implementation to directly flow-control RDMA traffic. We currently deploy our prototype RDMA EQIF in a smart NIC, as shown in Fig. 6. We use port representors [8] to interpose on RDMA-enabled devices the SmartNIC exposes to the host and its virtual machines.

Our EQIF does not modify the NIC's RDMA implementation, but it does need to tell the sender to slow down when the TX queue to a destination grows. Depending on the SmartNIC model, we use different techniques to control the sending rate of the RDMA engine. For the Stingray, we issue PFC PAUSE packets to slow down the sender; this works well, but it has the side effect of slowing all traffic coming out of the RDMA engine, not just the one to the backlogged destination. The BlueField 2 supports DCQCN, so our implementation uses this to control RDMA. We could use ECN to signal DCQCN flows to slow down, but we prefer to send congestion notification packets (CNPs) directly from the sending EQIF to the RDMA sender. This reduces the length of the DCQCN control loop and allows one-time tuning of the DCQCN parameters to this constant delay. Unfortunately, RDMA RC packets lack the source QP number in their headers, which is needed for sending back a response, so we develop a connection tracking module [10] for RDMA CM, enabling CNP generation.

Our current smart NIC implementation moves packets from the host to the ARM cores and then to the wire; with bidirectional traffic the SmartNIC's interconnect can become a bottleneck. It should be possible to only move the packet headers to the ARM cores, but our implementation does not support this yet. To avoid our results being affected we configure our RDMA testbed to a lower rate (10 Gbps). A NIC designed for EQDS would not add this additional latency. Despite this, our implementation increases RDMA performance in many cases while allowing coexistence with other protocols.

**The Native EQIF** is the preferred option for performance-oriented, EQDS-aware transports. It uses a shared memory area to store packets with lockless descriptor rings used to move packets to and from the EQIF via a zero-copy API. This EQIF offers additional low level information to host transports including the size of the TX buffer to the destination, the size of the destination's pull-queue, per-packet and per-path network RTTs and delivery notifications.

Latency information provided by the Native EQIF is similar to that provided by 1RMA, so for cases where core congestion is common, delay-based congestion controllers such as Swift [24] or Timely [28] can be implemented on top.

We have implemented *eqdsperf*,a performance testing tool over the Native EQIF, as well as a lightweight UDP stack that is optimized to run over EQDS. Applications using the UDP socket API can simply be linked to our stack, either statically at compile time, or dynamically using LD _PRELOAD.

# 7 Implementation

We implemented two versions of EQDS, one using DPDK and one as a Linux kernel module. The goal is to add minimal overhead, but inevitably there are tradeoffs to be made.

Our DPDK implementation was built with performance in mind and uses two CPU cores, regardless of load. These can be host CPU cores, but it is preferable to use the ARM cores on a smart NIC. The main thread takes turns reading packet batches from the host-facing and network-facing NICs. Once read, packets are processed to completion. At the sender, this includes NACKs and PULLs triggering the (re)transmission of queued packets, and at the receiver trimmed packets elicit NACKs while packets that fill a hole at the head of the reorder queue trigger the release of waiting packets. The main thread also checks for timer expiration, largely eliminating the need for synchronization. A second thread is used to pace PULLs and to send deferred ACKs, providing accurate PULL pacing at the expense of burning a second core. When the NIC supports fine-grained pacing (e.g. Intel Columbiaville), EQDS can offload pull-pacing to the NIC, reducing CPU usage.

Our kernel implementation is aimed at being cheap and easy to deploy. As is usual on Linux, outgoing packets are processed in the context of the sending process. They are captured by a hook after routing has taken place. If there is enough credit, processing continues until they are handed to the NIC's driver. Inbound, EQDS acts like a UDP service; all incoming packets, both data and control, are processed in a soft IRQ and fed to a kernel UDP socket. Data packets that can be forwarded straight away are re-injected into the IP stack after decapsulation. If control or data packets have to be sent back, a Layer 3 socket is used.

High Resolution Timers are used for PULL pacing with their handlers executed in a soft IRQ. The downside of using kernel timers is that their timing is at the mercy of the Linux scheduler. This adds jitter to the PULL pacing. To mitigate such jitter, senders must use higher window values than usual.

Offloads are key to achieving high TCP performance with Linux, so our implementation leverages both TSO and GRO. With TSO, TCP will send large segments, EQDS will encapsulate them, then an EQDS-unaware NIC will split the encapsulated packet, copying the extra headers verbatim in front of the inner TCP/IP headers. The problem is that all the split packets will have the same EQDS sequence number. Fortunately, due to a peculiarity in how TSO is performed, there is a workaround. A NIC increments the IP ID of every segment following the first. We send all EQDS packets with an IP ID of zero, and leave gaps in the EQDS sequence space. The receiver then adds the received IP ID to the received sequence number to obtain the full sequence number. Future EQDS-aware NICs would remove the need for this workaround. Inevitably, using TSO causes a burst of unpaced packets to be sent which, as with timer jitter, requires a small increase in the window used by EQDS.

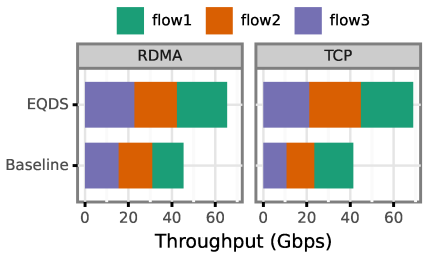


Figure 7: Permutation throughput in the T2 testbed (BlueField-2 hosts).

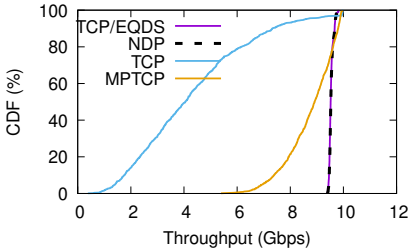


Figure 8: Simulation: permutation throughput in a 1024-node Fat Tree.

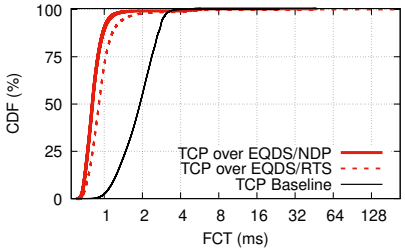


Figure 9: FCT: 1MB flows from random servers, closed loop (T1 testbed)

# 8 Evaluation

Datacenter networks have been shown to suffer from a number of pathologies including reduced throughput due to flow collisions, high loss or delay due to incast, inflated latency for RPC-style traffic and a dependency on compatible congestion control mechanisms for network sharing. EQDS's main goal is to enable the deployment of known solutions to all these problems in actual networks, and to make the resulting performance available to unmodified host stacks.

The first part of our evaluation examines whether EQDS helps mitigate these known pathologies and can boost performance for regular datacenter applications. To ensure the results are not specific to one host stack or deployment model and do not depend on network support, we experiment with unmodified TCP/IP (Linux) and RDMA NIC stacks (BlueField 2, CX4), run EQDS both on the host and on two smart NICs, and use both legacy and trimming-enabled networks.

We find that EQDS helps boost throughput of unmodified TCP/IP and RDMA stacks by up to 30-40%, mitigating the effects of collisions in a permutation traffic matrix (§8.1) and for NVMe over Fabric traffic. It achieves near-perfect incast behaviour with very small in-network queues both with trimming (our testbed and simulation) and RTS (on Amazon EC2), halving the latency and doubling the throughput of a micro-service based social network application benchmark on busy networks, as well as speeding up memcached by 10-30x (§8.2). Finally, we show that EQDS helps conflicting upper-layer congestion controllers nicely share the underlying network without any in-network support (§8.3).

In the second part we seek to understand whether EQDS introduces problems of its own. The biggest concerns are host overheads such as EQDS software's memory and CPU costs, its ability to handle high link speeds and dependence on specific hardware for performance.

In our evaluation we used two small-scale testbeds: T1 is a testbed we used for TCP/IP tests (10 servers) and T2 for mixed RDMA and TCP/IP tests (6 servers). Both testbeds used leaf-spine topologies and support trimming, but can also be configured with drop-tail/ECN; a detailed testbed description is provided in Appendix B. To test behaviour at scale and behaviour over legacy networks we use a large scale deployment in Amazon EC2 as well as simulation.

## 8.1 Improving throughput

TCP and RDMA require in-order packet delivery to function well, so datacenters switches hash the packet 5-tuple to select one of many equal-length paths to the destination. However, when the number of flows is small and the flows are high bandwidth, such placement can randomly place multiple flows on the same link causing congestion. Prior work has shown that flow collisions degrade performance in folded Clos topologies [11, 38] by up to 60% in the worst-case where a permutation traffic matrix is used, where each host sends to and receives from one other host. EQDS's per packet multipath should avoid such performance loss, at the cost of performing reordering in the receiving EQIF. We have run permutation experiments for RDMA in testbed T2 and for TCP in testbeds T1 and T2 as well as in simulation at larger scale.

Figure 7 shows that EQDS successfully spreads single flow TCP and RDMA traffic over all the available paths without causing reordering problems. TCP and RDMA flows running over EQDS achieve 22Gbps on average (maximum 24Gbps) in a permutation traffic matrix, compared to an average of 12-14Gbps without EQDS.

Our simulation results in Figure 8 explore behaviour at larger scale (FatTree with k=16, 1024 servers, 10Gbps NICs). Flow collisions hurt TCP badly, yielding only 40% of the network capacity. Multipath TCP[38], a variant of TCP that spreads traffic over multiple subflows (8 in our experiment) fares better with mean utilization close to 90%. NDP's packet spraying enables it to achieve near-optimal throughput. Finally, TCP over EQDS benefits from packet spraying but avoids the costs of reordering which is handled by EQDS, achieving similar performance to NDP.

Permutation traffic matrices highlight worst-case behaviour, with infinite flows and the smallest number of flows possible. Do collisions actually matter for other traffic matrices, for shorter flows, and for real applications? We examine this next.

**1MB flows, random traffic matrix.** On our testbed we run a workload where each server downloads a 1MB object from another randomly chosen server in a closed-loop, mimicking a storage workload over TCP. We measure flow completion times and plot them in Figure 9. EQDS lowers median and 95th percentile completion times by 2.4x and 2x respectively.

We also ran the same experiment with EQDS using request-

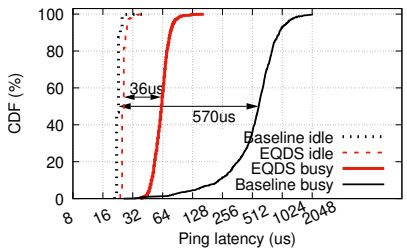


Figure 10: Ping latency: target is idle or busy with 9 incoming TCPs (T1 testbed).

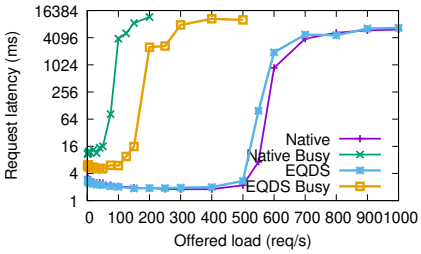


Figure 11: DeathStarBench in the T1 testbed: request latency

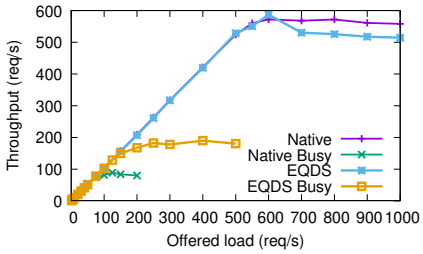


Figure 12: DeathStarBench in the T1 testbed: throughput

to-send and without trimming support in the switches. The flow completion times (FCT) for this setup are, as expected, slightly larger than the NDP-based implementation due to the additional RTT needed at the beginning of transfers. Still, EQDS/RTS halves the completion time of TCP compared to baseline, both in the median and at 95%. At 99% all variants have similar FCTs, and beyond that EQDS can take longer due to missing optimizations in the code for short flows.

**NVMeOF.** We run a disaggregated storage service using a Storage Performance Development Kit (SPDK) NVMeOF-RDMA target with a null block device (i.e. no storage access) to stress the network subsystem. The NVMeOF-RDMA protocol is target-driven, with the server reading or writing to memory buffers in the client after coordinating the access via rendezvous. This involves both latency-sensitive operations for control and throughput-hungry operations for data.
We run the NVMeOF targets and clients on separate ToRs. Each client accesses the targets round-robin, using the SPDK `perf` utility to generate the workload. Figure 13 shows the throughput of random writes and reads of 64 KB blocks while varying the queue depth of the NVMeOF target. EQDS with its multipath support increases both peak read and write throughput and reduces standard deviation. Note that EQDS requires deeper NVMeOF queues to achieve higher throughput due to the added latency of our BlueField setup (§8.5).

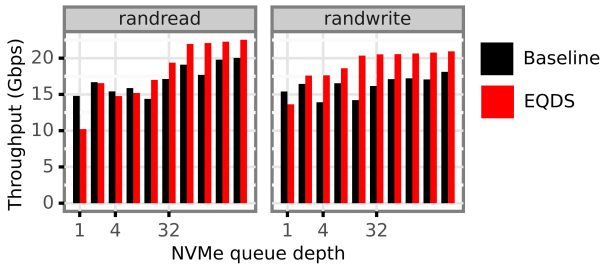


Figure 13: NVMeOF in the T2 testbed (BlueField-2).

## 8.2 Improving application latency

In deployed networks, there is a strict tradeoff between supporting many-to-one incast traffic gracefully, typically by provisioning large shared buffers in the network, and the latency of request-response applications such as micro-services or in-memory key-value storage (e.g. memcached). EQDS solves this trade-off by moving the buffering to the sending hosts, promising to achieve both low latency and good incast behaviour simultaneously.

**Many to one traffic.** To understand the baseline behaviour, we run large many to one workloads (850 iperf senders to the same receiver) in both simulation and on Amazon EC2 VMs.

We use htsim simulation to understand the behaviour of TCP NewReno when the destination link runs at 10Gbps, and we vary the size of bottleneck switch buffer. Figure 14 is a CDF of the flows' mean throughput. When the buffer is small (approx. 1 packet per flow), there is a large variance of throughput, with many flows starved. As we increase the buffer to around 100 packets for each flow, TCP can share capacity much more evenly, and there is no starvation. We also show the result when the switches implement fair-queuing, which further reduces variance and reduces buffer needs.

We run the same workload over EQDS, with the EQIF per-destination buffer set to 100 packets and a 15 packet buffer at the bottleneck. EQDS perfectly shares the bottleneck capacity, emulating a fair queue at the bottleneck link.

While our simulations show that TCP many-to-one requires significant buffering to work well without EQDS, what is the actual behaviour in production datacenters? We rented VMs on Amazon EC2 (m5.8xlarge for the receiver, and m5.xlarge for all others, 10Gbps link speeds) and deployed the Linux kernel version of EQDS, configured to run in RTS mode, as EC2 does not currently implement trimming.

We ran the same workload, with 850 hosts running iperf to one receiver and plot the results in Figure 15. Note the linear $x$ axis for this plot: many to one traffic in EC2 works fairly well, similar to the simulated fair-queuing results. TCP running over EQDS in EC2 achieves almost perfect sharing and 4% better throughput than the baseline, matching our simulations.

To achieve such good sharing, it appears that EC2 uses large buffers.We measure these buffers by pinging the same destination from an idle VM before and during the incast. Figure 16 is a CDF of ping latency. EC2 ping latency increases 163x from 55$\mu s$ to 9ms during the 10Gbps incast, indicating that a buffer of around 11MB exists in this network. In contrast, EQDS achieves similar many-to-one throughput with only a 21$\mu s$ increase in ping latency.

Is this behaviour specific to EC2 or the RTS backend? We

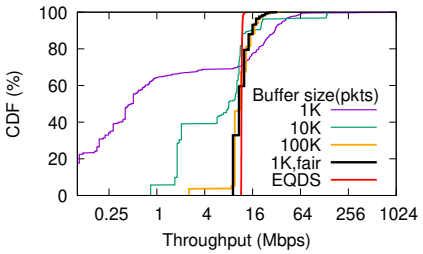


Figure 14: Simulation: 850 to 1 traffic, varying size of bottleneck buffer.

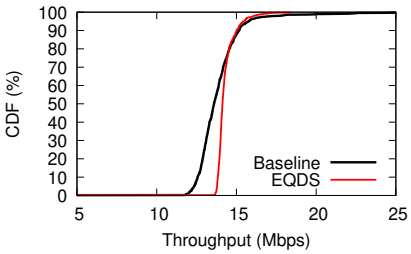


Figure 15: Amazon EC2: 850 to 1 traffic, kernel EQDS, RTS mode

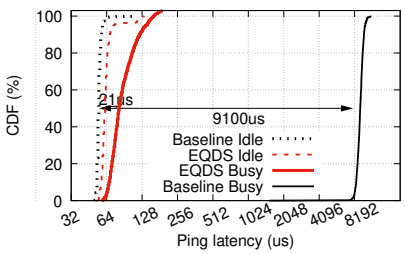


Figure 16: Ping latency (EC2): target is idle or busy with 850 incoming TCPs.

ran a similar experiment in our testbed using trimming instead of RTS. Figure 10 shows the results. When the destination is busy, EQDS with trimming results in a 36$\mu s$ increase in ping RTT, similar to RTS on EC2. Without EQDS, the increase is around 500$\mu s$ because our switch buffers for TCP are 1MB (less than EC2) and the link speed is 25Gbps. In summary, EC2 appears to be using large buffers to cope with incast. How does this affect latency sensitive traffic?

**Memcached.** We installed `memcached` on our EC2 destination VM and used `memcslap`, a benchmarking client, to measure server performance by issuing 1000 GET/PUT operations in a closed-loop manner. The mean request latency for an idle memcached server in EC2 is 700us. When running over EQDS, the same request takes around 900us due to the additional kernel processing EQDS does and the use of RTS.

When the destination is busy with 100 iperf clients sending to it, the mean request latency over EQDS increases 3x to 3ms, mostly due to sharing bandwidth with iperf, compared to a 140x increase to 100ms without EQDS due to large buffers. With 850 iperf clients sending, memcached over EQDS has a mean request latency of 23ms due to sharing bandwidth with 8.5 times more flows, compared to 400ms over native EC2. Overall, EQDS reduces memcached request latency for a busy EC2 server by 20 to 30x compared to the baseline.

**Micro-service apps.** We ran the latency-sensitive social network application of DeathStarBench [13] over kernel EQDS. We distributed the micro-service nodes to ensure that most requests are not local and must use the network, and used `wrk` to generate requests in a closed-loop manner.

We tested two scenarios: one where the network was idle, and one where high-throughput traffic going to the same hosts saturated the links filling the switch buffers. We note that the social network application does not generate much traffic – 100Mbps peak – but is latency sensitive so we expect to see an impact of longer network latencies in our "busy" scenario.

Figures 11 and 12 show the results. On an idle network social network requests take ∼2ms to complete, with little difference between the baseline and EQDS. Our deployment can sustain a request load of about 500 requests per second after which DSB saturates the CPU. When the network is busy, however, EQDS reduces request latency by 50% and can sustain double the baseline request load.

## 8.3 Sharing the network

EQDS allows fine-grained host sharing policies without in-network support; we examine some interesting scenarios.

**Non-responsive traffic competing with TCP.** Consider the scenario in Figure 17 where a receiver in our testbed receives data from 4 TCP senders and then a UDP sender starts sending at line-rate (emulated with iperf3). As the UDP flow does not respond to congestion, the testbed network run in legacy mode allows the UDP flow to use nearly 25Gbps of bandwidth, starving the TCP flows. With EQDS (in trimming mode in this experiment), the UDP sender is throttled at the sending EQIF which enforces perfect ingress sharing, without any need for fair-queuing in the network.

**Congestion controllers being nice.** In fact, EQDS can enable **any** combination of existing congestion controllers to co-exist fairly without any in-network support. In figure 18 we show the sharing results when two senders send to the same receiver, with one sender using Cubic[17] and the other using the congestion control algorithms shown on the $x$ axis label. We used all the congestion control implementations available in the Linux kernel, as well as the Mellanox DCQCN implementation. EQDS is able to fairly share the receiver's link without in-network support for almost all congestion controllers, in contrast to the status quo where latency-based schemes such as BBR[4] are starved by loss-based ones (e.g. Cubic). One exception are the Vegas[2] controllers that cannot utilize the 25Gbps link over EQDS when running alone, and that receive a smaller share when competing against Cubic. This appears to be due to Vegas measuring a small base RTT and stopping the window increase before it reaches a BDP.

The amount of buffering in the EQDS TX queue depends, as expected, on the congestion control algorithm. BBR is best, with an average latency of 243$\mu s$ when sending at line rate, compared to DCTCP (K=16) at 435$\mu s$ and Cubic at 1315$\mu s$.

**RDMA versus TCP.** To see how EQDS aids effective co-existence between different host stacks, we run a single TCP and $n$ RDMA flows to the same destination in T2 (BlueField NICs). Figure 19 shows the bandwidth distribution among competing flows as the number of RDMA flows varies from zero to four. Without EQDS, TCP fills the switch buffers while DCQCN causes RDMA to back off, with some RDMA

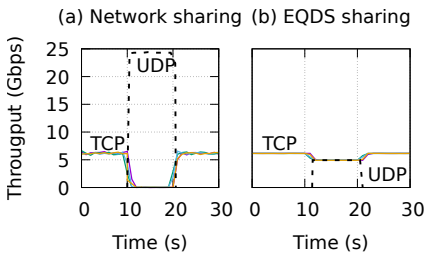


Figure 17: Bandwidth sharing (5 to 1): one non-responsive UDP sender (testbed T1)

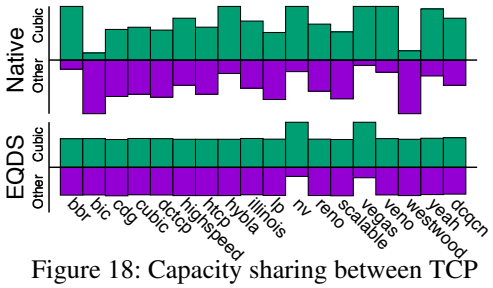


Figure 18: Capacity sharing between TCP variants, with and without EQDS (testbed T1)

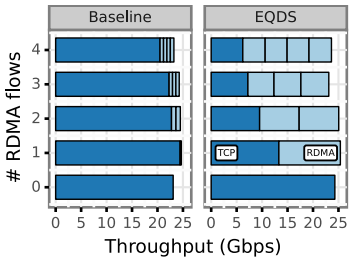


Figure 19: RDMA and TCP network sharing (testbed T2)

flows being starved. With EQDS, data is buffered at the sending hosts and the bandwidth allocation is determined by the receiver, protecting RDMA and sharing the bandwidth fairly.

**NVMeOF in parallel with a TCP shuffle.** To understand sharing beyond a single bottleneck link, and to also test the CX4 RDMA stack over EQDS, we run the NVMeOF RDMA benchmark (§8.1) in parallel with TCP traffic emulating a MapReduce shuffle operation on the T2 testbed (CX4 hosts). Specifically, three nodes on one ToR run both iperf senders and NVMeOF targets. These three send to three nodes on another ToR running both iperf receivers and NVMeOF clients that perform random reads.

We observe that with EQDS the TCP shuffle alone is about 30% faster than without EQDS (26.1 Gbps vs 19.6 Gbps), in line with the previous experiments. When both TCP and RDMA run concurrently without EQDS, the shuffle throughput drops to 17.6 Gbps and NVMeOF drops to 2.56 Gbps – far from the optimal fair share. Under EQDS, shuffle and NVMeOF achieve 15.8 Gbps and 10 Gbps respectively, which is both a fairer allocation and higher overall throughput.

## 8.4 EQDS in legacy networks

To investigate EQDS with an oversubscribed core, we interconnect two ToRs in our T1 testbed with two 25Gbps spine links. Fair queuing is enabled. One ToR hosts three servers (2 at 100Gbps, 1 at 10Gbps); the other ToR hosts eight clients (mix of 10 and 25Gbps links). Each client connects to one server and continuously requests a 1MB object in a closed-loop, creating a new TCP connection each time. Clients are equally balanced across servers.

We increase the number of active clients from 1 (core utilization ≈50%) up to 8 (400% core over-subscription). We plot the median and 99% FCTs for baseline TCP, TCP-over-EQDS with trimming and TCP-over-EQDS using RTS. When the core is lightly loaded, EQDS's packet-level load balancing gives slightly smaller median (Figure 20) and 2x to 5x smaller 99th percentile FCT (Figure 21). As we add more clients and the core becomes overloaded, EQDS ends up behaving similarly to the native baseline, while RTS is slightly slower than baseline as it requires a large amount of buffering. For EQDS with trimming, as the core gets busier the trim rate increases and each packet can be re-sent multiple times. While this does not affect FCT, it is undesirable; it would be better to reduce the pull rate when core congestion is detected.

**Legacy "elephant" traffic.** To understand how legacy traffic coexists with EQDS, we use the same configuration but with two clients downloading 1MB objects from two servers, and create a long-lasting iperf3 flow to another client. We vary the throughput for this legacy flow from 0 to 25Gbps and measure the FCTs of the other flows. Figure 22 shows how flow collisions on the spine between native 1MB flows and the elephant flow affects FCT: the median grows by 2.5x when the elephant flow fully occupies the spine link. With EQDS, packet spraying helps mitigate the effects of the elephant flow, and the increase in FCT is modest. A future implementation of load-aware routing should improve things further.

## 8.5 Host processing evaluation for EQDS

Our experiments so far have looked at how EQDS can help existing stacks, and used 25Gbps (our testbeds) and 10Gbps link speeds (EC2, simulation). We now examine the performance and overheads of our two EQDS implementations and evaluate how they perform at higher speeds. We tested using several configurations:

**Setup 1.** DPDK on the host, both with native transports and underneath the VM stack, where EQDS takes the role of the software switch used in virtualization.

**Setup 2.** DPDK on an SoC-based SmartNIC (Broadcom Stingray PS225 or Mellanox BlueField 2) with support for legacy TCP and RDMA traffic.

**Setup 3.** DPDK on the host, processing RDMA traffic to and from a Mellanox CX4 NIC.

**Setup 4.** Our Linux Kernel 5.4 implementation, with EQDS kernel module running underneath the TCP/IP stack.

The DPDK implementation is the most versatile and performs best, though this depends on the higher level stack and the setup used (Figure 23). The best performance is given by `eqdsperf` in setup 1 using the zero-copy native EQDS transport. Between two 2.5GHz Xeon Silver 4215 machines with Intel Columbiaville NICs, eqdsperf achieves 40Gbps with 1.5KB packets and 100Gbps with 4KB packets. The bottleneck is the sender; with two senders to the same receiver, the link saturates with 3KB packets.

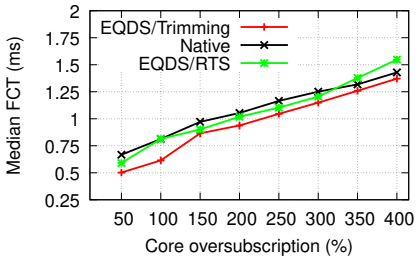


Figure 20: Median FCT for 1MB flows (oversubscribed core).

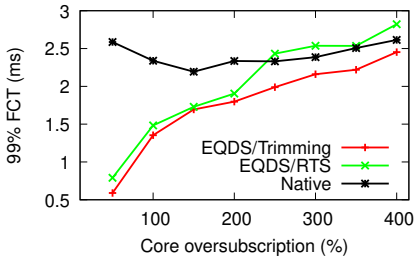


Figure 21: 99% FCT for 1MB flows (oversubscribed core).

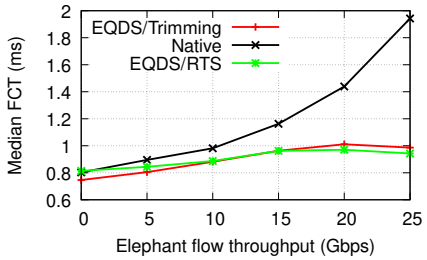


Figure 22: Median FCT with an elephant legacy flow.

| Setup | Higher stack | Throughput (Gbps) Link | 1.5KB/9KB | Latency ping(us) |
|---|---|---|---|---|
| (1) Host | eqdsperf | 100 | 40/98 | 15(+1) |
| (1) Host | Linux VM | 100(40) | 27/55 | 18(+1) |
| (2)PS225 | Linux | 25 | 23.6/24.1 | 31(+12) |
| (2)PS225 | RDMA | 25 | 23.5/24.1 | 18(+12) |
| (2)Blue2 | Linux | 100 | 23/49 | 22(+16) |
| (2)Blue2 | RDMA | 100 | 23/49 | 22(+16) |
| (3)Host/CX4 | RDMA | 10 | 9.6/9.6 | 13(+9) |

Figure 23: EQDS DPDK performance

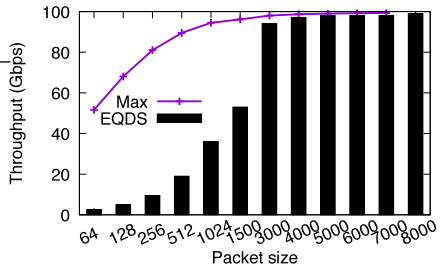


Figure 24: `eqdsperf` throughput (setup 1)

| Stack | MTU | TSO off GRO off | TSO on GRO on |
|---|---|---|---|
| Linux | *1.5KB* | 17Gbps | 40Gbps |
| EQDS | *1.5KB* | 5Gbps | 27Gbps |
| Linux | *9KB* | 40Gbps | 60Gbps |
| EQDS | *9KB* | 28Gbps | 55Gbps |

Figure 25: EQDS Kernel performance (setup 4)

In Setup 1, when we run EQDS underneath Linux VMs in KVM using vhost-user networking, it achieves 27Gbps with 1.5KB packets compared to 40Gbps for the baseline (testpmd). The cost here for both testpmd and EQDS comes from the packet copy from the guest to the host.

In Setup 2 we run EQDS on both the Broadcom PS225 SmartNIC and Mellanox's BlueField 2 NIC. Our EQIF uses one NIC core for each of the two links and one for credit pacing. On the Stingray, EQDS saturates the link with a single core when the MTU is 1.5KB or larger. On the PS225 the results are similar with EQDS reaching 50Gbps with an 8KB MTU. We note that SmartNIC ARM cores are weaker than x86 cores and memory bandwidth seems limited; reaching 100Gbps may require more offloading.

In Setup 3 we divert RDMA traffic from a Mellanox ConnectX-4 Lx NIC via a host core. This is not ideal as it requires an extra round trip across the PCIe bus, but demonstrates RDMA-over-EQDS even without a smart NIC. Since the CX4 PCIe bandwidth is 56Gbps, we limit the link bandwidth to 10Gbps to avoid a PCIe bottleneck. With EQDS, the bandwidth of a single RDMA flow between two RDMA nodes remains the same as with baseline RDMA (Table 23), but the additional PCIe round-trip increases median latency by 9 µs on an unloaded system and by 14 µs under load.

In Setup 4 we run EQDS in a kernel module. Figure 25 shows TCP performance with and without EQDS, and explores the effect of TCP offloads. Our kernel implementation can reach 55Gbps (one core) with jumbograms, or 27Gbps with a 1.5KB MTU. TCP's dependence on offload support is clear. EQDS increases CPU utilization at the sender by 2% and at the receiver by 5% when running an iperf at 55Gbps.

**Tunnel setup overhead.** When EQDS tunnels are already set up, EQDS only adds a few $\mu s$ of latency, as shown in Fig. 23. When a new host is contacted, however, a tunnel is setup dynamically using a zero-RTT approach. We measure this by tearing down tunnels, then sending a series of pings; the difference between the first ping time and subsequent ones is around 20$\mu s$. This setup latency is due to memory initialization costs for the tunnel data structures.

**EQDS memory consumption.** How does EQDS memory consumption scale with the size of the datacenter? There are three categories of memory to consider.

*A Sending EQIF* buffers packets awaiting transmission. For TCP, we use 100 packets per destination by default, but we find that if needed this can be reduced as low as four packets per destination when sending to many receivers. For RDMA, the upper marking threshold is set to 375 packets.

*The receiving EQIF* has a reorder buffer per sender. In the worst case with faulty links, the buffer can reach the EQDS window which is between 50 packets (DPDK) and 150 packets (kernel). In practice this is limited to less than 10 packets.

*In-flight packets* are buffered at the sender pending retransmission, but the pending buffer size is limited by the BDP. We verify this by starting iperfs to an increasingly larger number of receivers. The total number of in-flight packets across all tunnels saturates at around 400 packets (NIC ring size plus one BDP) regardless of the number of receivers.

With 1.5GB of DRAM, EQDS can buffer packets for 10,000 active destination; this fits in the RAM of the SmartNICs we used (8-16GB) as well as the hosts. In practice both the number of active destinations and actual buffer utilization are smaller; on EC2 the receiver's memory usage for EQDS did not exceed 100MB with 850 senders.

## 9 Related work

Tunneling is widely used in datacenters for security isolation of traffic (e.g. VXLAN and GRE). These protocols, however, do not offer any performance isolation between tenants. Rate limiting is typically used in public clouds, but this offers limited isolation, especially for incast workloads.

Numerous research works examined the performance sharing problem [40, 34, 35]. Seawall [40] and ElasticSwitch [35] use rate-limiting between each pair of hosts in a datacenter to manage sharing, and use centrally computed weights to achieve fair sharing among different tenants. FairCloud [34] discusses fundamental tradeoffs in sharing cloud networks. This line of works relies on rate limiting and needs large in-network queues to cope with incast; it also doesn't improve utilization of multipath networks.

Virtualized congestion control works such as VCC [6] and AC/DC [19] propose ways of deploying new TCP congestion variants without VM changes. Both keep per-flow congestion state in the hypervisor, applying rate-limiting techniques (such as receive window reduction) to force the VM stack to reduce its rate; both show how DCTCP can be deployed in this way. OnRamp [3] is a recent proposal that aims to improve the fast start phase of transport protocols by tracking fine-grained RTT measurements per flow, in the hypervisor, and then stopping packets from entering the network when latency increases above a certain threshold. This line of work does not allow multipath transmission of TCP packets or other network-specific enhancements, and still requires large in-network buffers to cope with incast.

EQDS takes the next logical step over this line of prior work: it completely decouples host stacks from the network via edge queues, thus supporting multiple higher layer transports (e.g. TCP, RDMA or native). EQDS does not do rate limiting (because it can build large queues), but uses a receiver-driven control loop instead, ensuring that network queue depths are kept as low as possible. Any sharing outcome for clouds (e.g., [34]) can be configured using the primitives EQDS provides to higher layers. The key benefit of EQDS is decoupling the higher layer transports from lower layer implementations; this enables regular TCP run over packet-sprayed networks, among others.

Fastpass [33], pHost [14], Aeolus [20], NDP [18], Homa [31], qJump [15] are techniques that show how one can operate a network at high load and small queues. In-band network telemetry [32] provides accurate congestion information to endpoints that can be used in over-subscribed topologies (e.g. HPCC [25]). In contrast to all these transport or congestion control protocols, EQDS is a congestion tunneling layer that is meant to allow other transports to operate on top; it can use the mechanisms proposed by any of these or other, yet-to-be-invented, mechanisms to drive packet pacing and ensure that the network core is used efficiently, but it also allows to deploy them transparently to user applications. Our implementation implements both NDP [18] and a request-to-send variant that does not require trimming.

New transport stacks like PonyExpress [27] or eRPC [23] show the benefits of kernel bypass to support novel APIs, but they face an uphill battle in deployment. EQDS allows such transports to be deployed without changes to the network.

## 10 Conclusions and next steps

EQDS is a new network layer that provides strong performance isolation among co-existing transports by pulling the shared queues out of the data center network core and moving them to the edge. More importantly, it introduces a new datagram service abstraction which fully decouples the transport services above it from the network implementation underneath, thus enabling independent evolution of these layers while ensuring future compatibility among them.

EQDS is designed for gradual adoption. From the application perspective, RDMA and TCP EQIFs allow seamless deployment of EQDS without any application modifications. In the longer term we envision the emergence of application protocols implemented atop and optimized for EQDS-native APIs, thereby taking full advantage of the improved visibility into the network performance that they offer.

From the network infrastructure perspective, one can use our software-only implementations in Linux kernel, SmartNIC or/and host that allow the benefits of EQDS isolation to be reaped at low performance cost, and can also run without switch support for packet trimming.

The NIC implementation is ideal from a deployment point of view, as it is cleanly separated from the host software, which can use offloading support as before. Our smart NIC setup, however, adds a little unnecessary latency (12-16us) and is limited to 25Gbps per core at 1.5KB MTU. These limitations should be fixable with NIC ASIC support for EQDS.

There are many examples of existing ASIC and FPGA implementations of connected transport protocols, ranging from RDMA [9, 21, 22, 41], to TCP offload engines [30, 12, 39]. An EQDS NIC would build upon these works, and its hardware version might actually be simpler: keeping state for each connected endpoint rather than per-flow means more connections could be stored in on-chip memory, and its relaxed ordering and reliability guarantees allow implementing a simpler state machine. By offloading control packet processing, we expect to shorten the EQDS control loop delay significantly, and reducing its jitter, thus minimizing switch buffer usage. Furthermore, offloading connection setup and teardown logic would allow low latency small flows to get close to native performance.

**Acknowledgements.** The authors thank Mihai Brodschi for implementing the UDP stack over EQDS, and our shepherd and the anonymous reviewers for their feedback. We thank Intel, Broadcom and Nvidia for providing hardware for testing. This work was partly funded by CORNET, a research grant of the European Research Council (no. 758815).

## References


[1] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. "Data Center TCP (DCTCP)". In: *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2010.

[2] L.S. Brakmo and L.L. Peterson. "TCP Vegas: end to end congestion avoidance on a global Internet". In: *IEEE Journal on Selected Areas in Communications* 13.8 (1995), pp. 1465–1480.

[3] "Breaking the Transience-Equilibrium Nexus: A New Approach to Datacenter Packet Transport". In: *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, Apr. 2021.

[4] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. "BBR: Congestion-Based Congestion Control". In: *ACM Queue* 14, September-October (2016), pp. 20–53.

[5] Yanpei Chen, Rean Griffith, Junda Liu, Randy H. Katz, and Anthony D. Joseph. "Understanding TCP Incast Throughput Collapse in Datacenter Networks". In: *Workshop on Research on Enterprise Networking (WREN)*. ACM, 2009.

[6] Bryce Cronkite-Ratcliff, Aran Bergman, Shay Vargaftik, Madhusudhan Ravi, Nick McKeown, Ittai Abraham, and Isaac Keslassy. "Virtualized Congestion Control". In: *SIGCOMM '16*. Association for Computing Machinery, 2016, pp. 230–243.

[7] Steve Deering. *Watching the waist of the internet hourglass*. ICNP plenary. 1998.

[8] DPDK. *DPDK Porgrammer's Guide » Switch Representation within DPDK Applications*. 2019. URL: https://doc.dpdk.org/guides-19.11/prog_guide/switch_representation.html.

[9] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. M. Merritt, E. Gronke, and C. Dodd. "The Virtual Interface Architecture". In: *IEEE Micro* 18.2 (Mar. 1998), pp. 66–76.

[10] Haggai Eran. *libconntrack-cm – Connection Tracking for RDMA CM*. 2022. URL: https://github.com/acsl-technion/libconntrack-cm.

[11] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. "Hedera: Dynamic Flow Scheduling for Data Center Networks". In: *Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2010.

[12] W. Feng, P. Balaji, C. Baron, L.N. Bhuyan, and D.K. Panda. "Performance characterization of a 10-Gigabit Ethernet TOE". In: *13th Symposium on High Performance Interconnects (HOTI'05)*. 2005, pp. 58–63.

[13] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayantara Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Yuan He, and Christina Delimitrou. "An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud and Edge Systems". In: *Proceedings of the Twenty Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Apr. 2019.

[14] Peter X. Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. "pHost: Distributed Near-optimal Datacenter Transport over Commodity Network Fabric". In: *Conference on Emerging Networking Experiments and Technologies (CoNEXT)*. ACM, 2015.

[15] Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert N. M. Watson, Andrew W. Moore, Steven Hand, and Jon Crowcroft. "Queues Don't Matter When You Can JUMP Them!" In: *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, May 2015, pp. 1–14.

[16] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. "RDMA over Commodity Ethernet at Scale". In: *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2016.

[17] Sangtae Ha, Injong Rhee, and Lisong Xu. "CUBIC: A New TCP-Friendly High-Speed TCP Variant". In: *SIGOPS Oper. Syst. Rev.* 42.5 (July 2008), pp. 64–74.

[18] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. "Re-architecting Datacenter Networks and Stacks for Low Latency and High Performance". In: *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2017.

[19] Keqiang He, Eric Rozner, Kanak Agarwal, Yu (Jason) Gu, Wes Felter, John Carter, and Aditya Akella. "AC/DC TCP: Virtual Congestion Control Enforcement for Datacenter Networks". In: *SIGCOMM '16*. Association for Computing Machinery, 2016, pp. 244–257.

[20] Shuihai Hu, Wei Bai, Gaoxiong Zeng, Zilong Wang, Baochen Qiao, Kai Chen, Kun Tan, and Yi Wang. "Aeolus: A Building Block for Proactive Transport in Datacenters". In: *SIGCOMM '20*. Association for Computing Machinery, 2020, pp. 422–434.

[21] InfiniBand Trade Association (IBTA). *About InfiniBand*. (Accessed: May 2021). URL: https://www.infinibandta.org/about-infiniband/.

[22] InfiniBand Trade Association (IBTA). *The RoCE Initiative*. (Accessed: May 2021). URL: https://www.infinibandta.org/roce-initiative/.

[23] Anuj Kalia, Michael Kaminsky, and David G. Andersen. "Datacenter RPCs Can Be General and Fast". In: *NSDI'19*. USENIX Association, 2019, pp. 1–16.

[24] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan M. G. Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. "Swift: Delay is Simple and Effective for Congestion Control in the Datacenter". In: *SIGCOMM '20*. Association for Computing Machinery, 2020, pp. 514–528.

[25] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. "HPCC: High Precision Congestion Control". In: *SIGCOMM '19*. Association for Computing Machinery, 2019, pp. 44–58.

[26] Yuanwei Lu, Guo Chen, Bojie Li, Kun Tan, Yongqiang Xiong, Peng Cheng, Jiansong Zhang, Enhong Chen, and Thomas Moscibroda. "Multi-Path Transport for RDMA in Datacenters". In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Apr. 2018, pp. 357–371.

[27] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. "Snap: A Microkernel Approach to Host Networking". In: *SOSP '19*. Association for Computing Machinery, 2019, pp. 399–413.

[28] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. "TIMELY: RTT-Based Congestion Control for the Datacenter". In: *SIGCOMM '15*. Association for Computing Machinery, 2015, pp. 537–550.

[29] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. "Revisiting Network Support for RDMA". In: *SIGCOMM '18*. Association for Computing Machinery, 2018, pp. 313–326.

[30] Jeffrey C. Mogul. "TCP Offload is a Dumb Idea Whose Time Has Come". In: *HOTOS'03*. USENIX Association, 2003, pp. 5–5.

[31] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. "Homa: A Receiver-driven Low-latency Transport Protocol Using Network Priorities". In: *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2018.

[32] p4.org Applications Working Group. *In-band Network Telemetry (INT) Dataplane Specification*. 2020. URL: https://github.com/p4lang/p4-applications/blob/master/docs/INT_v2_1.pdf.

[33] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. "Fastpass: A Centralized "Zero-queue" Datacenter Network". In: *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2014.

[34] Lucian Popa, Gautam Kumar, Mosharaf Chowdhury, Arvind Krishnamurthy, Sylvia Ratnasamy, and Ion Stoica. "FairCloud: Sharing the Network in Cloud Computing". In: *SIGCOMM '12*. Association for Computing Machinery, 2012, pp. 187–198.

[35] Lucian Popa, Praveen Yalagandula, Sujata Banerjee, Jeffrey C. Mogul, Yoshio Turner, and Jose Renato Santos. "ElasticSwitch: Practical Work-Conserving Bandwidth Guarantees for Cloud Computing". In: *SIGCOMM '13*. Association for Computing Machinery, 2013, pp. 351–362.

[36] R. Ludwig and A. Gurtov. *RFC4015: The Eifel Response Algorithm for TCP*. 2003. URL: https://tools.ietf.org/html/rfc4015.

[37] R. Ludwig and M. Meyer. *RFC3522: The Eifel Detection Algorithm for TCP*. 2003. URL: https://tools.ietf.org/html/rfc3522.

[38] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. "Improving Datacenter Performance and Robustness with Multipath TCP". In: *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2010.

[39] Mario Ruiz, David Sidler, Gustavo Sutter, Gustavo Alonso, and Sergio López-Buedo. "Limago: An FPGA-Based Open-Source 100 GbE TCP/IP Stack". In: *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. 2019, pp. 286–292.

[40] Alan Shieh, Srikanth Kandula, Albert Greenberg, Changhoon Kim, and Bikas Saha. "Sharing the Data Center Network". In: *NSDI'11*. USENIX Association, 2011, pp. 309–322.

[41] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarni, and Gustavo Alonso. "StRoM: Smart Remote Memory". In: *EuroSys '20*. Association for Computing Machinery, 2020.

[42] Arjun Singhvi, Aditya Akella, Dan Gibson, Thomas F. Wenisch, Monica Wong-Chan, Sean Clark, Milo M. K. Martin, Moray McLaren, Prashant Chandra, Rob Cauble, Hassan M. G. Wassel, Behnam Montazeri, Simon L. Sabato, Joel Scherpelz, and Amin Vahdat. "1RMA: Re-Envisioning Remote Memory Access for Multi-Tenant Datacenters". In: *SIGCOMM '20*. Association for Computing Machinery, 2020, pp. 708–721.

[43] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. "Programmable Packet Scheduling at Line Rate". In: *SIGCOMM '16*. Association for Computing Machinery, 2016, pp. 44–57.

[44] Haitao Wu, Zhenqian Feng, Chuanxiong Guo, and Yongguang Zhang. "ICTCP: Incast Congestion Control for TCP in Data-Center Networks". In: *IEEE/ACM Trans. Netw.* 21.2 (Apr. 2013), pp. 345–358.

## Appendix A. Protocol state machines

The state machines for our protocol implementation are shown in Figures 26 (the sender) and 27 (for the receiver).

The aim is for the two endpoints to self-synchronize. The sender sets the SYN flag on all data packets sent until it receives a SYN-ACK (or SYN-NACK or SYN-SACK) packet from the receiver. As a window of SYN packets can arrive out-of-order, the lowest ten bits of the sequence number are always zero in the first SYN packet and the upper sequence number bits in SYN packets indicate an epoch number.

The receiver state machine is very simple, with just two states, `closed` and `established`. As SYNs from the initial window can arrive out of order, an arriving SYN causes the receiver to move to established state and set the initial sequence number to be that from the SYN with the lowest ten bits cleared. Any subsequent SYNs with the same epoch number are treated as normal data packets except the SYN flag is set in their ACKs.

When data arrives at the sender it chooses a random epoch number, starts sending data with SYN set, and moves to SYN-SENT state. It then moves to *established* state on receiving a SYN-ACK with the correct epoch number from the receiver.

The epoch number is needed because either end can drop state at any point. If a sender drops state very early in a connection, then immediately tries to re-establish a tunnel, old SYN packets may still be in flight. The epoch number allows the sender and receiver to agree which is the new connection.

If the sender retains the old epoch number, it simply chooses a greater epoch for the new connection. The receiver then accepts the new SYN as re-establishing the connection seamlessly. However, if the sender has no state, it chooses a random epoch number. If this random epoch number is greater it will be accepted, but if it is lesser, the receiver concludes it is old and replies with a SYN-ACK advertising its current epoch. If the sender receives such a mismatched greater epoch in a SYN-ACK, it concludes the setup has failed, chooses a new epoch greater than that advertised by the receiver, and resends its initial window of SYNs. This new attempt will then always succeed.

It is possible that SYN-ACKs from the previous connection are still in flight when the new connection is attempted. If these have a lower epoch (the common case) they are ignored; if they have a greater epoch they trigger the re-sync process as described above.

If a receiver has outstanding data in its reorder queue when a connected reestablishes, it releases this data to the host out-of-order as it can no longer guarantee the sequence space holes will be filled. This is expected to be very rare in practice as endpoints will not normally drop state with unacked data in transit.

The sender responds to any indication of an unhealthy tunnel by closing it and re-establishing a new tunnel, possibly after a timeout. A large number of retransmit timeouts implies that there is likely a network connectivity issue affecting the tunnel. Similarly, receipt of a RST indicates the receiver is in the `closed` state, and prompts re-establishment of the tunnel if there are packets in the EQIF TX queue.

The key benefit of this self-synchronizing design is that either endpoint can drop state without reliably informing its peer. This gives EQDS implementations a lot of freedom in managing per-tunnel memory, allowing lightly used tunnels to be dropped in memory pressure scenarios. While these ability is not used in our software implementations, we expect that future ASIC implementations of EQDS in NICs will make full use of these features.

## Appendix B. Details on the experiment setup.

Our T1 (TCP/IP) testbed has 10 endpoints:

- 8 Linux kernel endpoints emulated on four servers with Intel Xeon Silver 4215 CPUs @ 2.50GHz (8 cores, 16 hyperthreads), 128GB of RAM and a dual-port 100Gbps Intel Columbiaville NIC each (running in Setup 1, with EQDS on the host cores either as part of the kernel or as a DPDK process).

- 4 Broadcom endpoints emulated by two servers with Intel Xeo E5-2650L v2 CPUs @ 1.70GHz, 32GB of RAM (10 cores, 20 hyperthreads), each with a dual-port 25Gbps

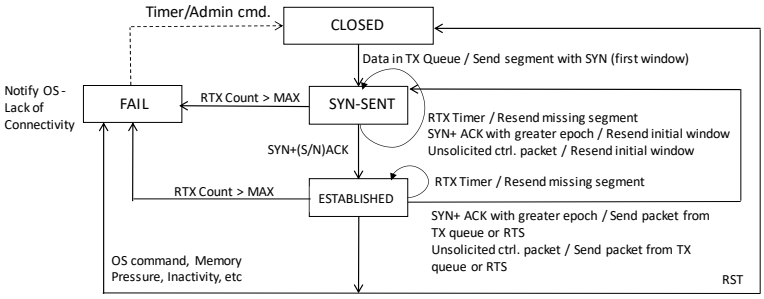


Figure 26: EQDS sender state machine

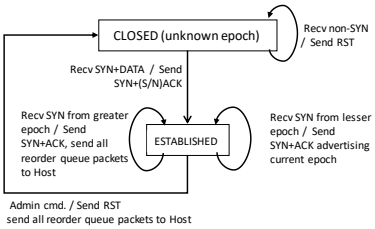


Figure 27: EQDS receiver state machine

Broadcom Stingray NIC (EQDS is running on the Arm cores of the Stingray, in Setup 2).

Each endpoint is connected via a 25Gbps link to a 64-port Tofino 1 switch; we downgraded the Columbiaville NICs to 25Gbps to match the speed of the Stingray NICs. We implemented packet trimming in the switch via a combination of ingress meters and cloning sessions. We used the four pipelines in this switch to emulate four ToR switches; these are connected via DAC cables to two spine switches emulated by another 32-port Tofino (one pipeline per emulated switch). Switch buffers are set to 15 packets (125KB) for EQDS and 200 packets (2MB) for TCP. Cross-sectional bandwidth is 200Gbps (slight over-subscription).

We run the workload in our Linux machines with and without EQDS, and measure performance using `ping`, `iperf` and `iperf3` for UDP tests. Our kernel and DPDK stacks interoperate and at 25Gbps have similar performance, so we omit a performance breakdown.

Our T2 (RDMA) testbed has six endpoints, each with a 2.1GHz Intel Xeon E5-2620 v2 CPU. The hosts are connected to an NVIDIA Spectrum SN3700 switch, configured using loopback cables as a 2-tier Clos topology with 40 Gbps bisectional bandwidth. All links are configured to 10 Gbps with 4 KiB MTU. Each of six ports of the switch is connected either to one of three dual-port 8-core NVIDIA BlueField-2 Smart NIC (two of which clock at 2.5 GHz and one at 2 GHz), or to an NVIDIA Mellanox ConnectX-4 Lx.

We implement trimming in the SN3700 by mirroring and truncating dropped frames, then sending them to a dedicated loopback port where a P4 program redirects them back to the destination port and appropriate queue. When trimming is enabled, switch buffers are set to 60 kB.

When using ConnectX-4 Lx NICs, EQDS runs on the host CPU instead of the SmartNIC, and traffic between the host network stack and EQDS uses NIC loopback.

The baseline RDMA performance for SmartNICs is measured by configuring the SmartNICs to forward traffic between the host and the network in hardware, without going through the ARM cores. For NICs, the host network stack uses the RDMA NIC directly.

**Buffer settings for TCP EQIFs.** Our experiments show that TCP/IP runs smoothly over EQDS with our default settings (sending EQIFs buffer up to 100 packets per destination), but does this change with other buffer sizes? Larger buffers simply result in more EQDS sender-side buffering for TCP Cubic, and do not affect performance at all. What about smaller buffers?

When the send buffer is at least one BDP (30 packets in our testbed), all TCP variants achieve line-rate; below that the throughput depends on the congestion controller. DCTCP maintains full utilization with K=16, while BBR needs half that to reach line rate..

**DCQCN parameter settings for the RDMA EQIF.** Our RDMA EQIF performs flow control using the NIC's DCQCN implementation, but it has a shorter control loop than a baseline DCQCN setup: it sends CNPs directly to the sender NIC instead of simply marking packets and waiting for the receiver to send CNPs. This allows more aggressive DCQCN parameters to be used.

We explored the DCQCN parameter space, varying the EQIF's probabilistic marking thresholds ($K_{low}$/$K_{high}$), active increase rate ($R_{AI}$) and rate increase/decrease timers ($R_d$/$R_i$). Our goal was to be able to stop the RDMA sender quickly during large incasts (1Mbps per sender) without dropping packets, and to be able to resume at line rate (i.e. have enough buffering) when an incast subsides (available rate is 25Gbps). The resulting parameters, shown below, are as aggressive as possible without under or overflowing the EQIF queue.

| | $K_{low}$ | $K_{high}$ | $R_d$ | $R_i$ | $R_{AI}$ |
|---|---|---|---|---|---|
| Baseline | 150 kB | 1500 kB | 4 µsec | 300 µsec | 5 Mbps |
| EQDS(BF2) | 72 kB | 584 kB | 4 µsec | 750 µsec | 5 Mbps |
| EQDS(Cx4) | 150 kB | 1500 kB | 4 µsec | 128 µsec | 50 Mbps |