Swift: Delay is Simple and Effective for Congestion Control in the Datacenter

Gautam Kumar, Nandita Dukkipati, Keon Jang (MPI-SWS)*, Hassan M. G. Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat
Google LLC

ABSTRACT

We report on experiences with Swift congestion control in Google datacenters. Swift targets an end-to-end delay by using AIMD control, with pacing under extreme congestion. With accurate RTT measurement and care in reasoning about delay targets, we find this design is a foundation for excellent performance when network distances are well-known. Importantly, its simplicity helps us to meet operational challenges. Delay is easy to decompose into fabric and host components to separate concerns, and effortless to deploy and maintain as a congestion signal while the datacenter evolves. In large-scale testbed experiments, Swift delivers a tail latency of $<50\mu s$ for short RPCs, with near-zero packet drops, while sustaining ~100Gbps throughput per server. This is a tail of <3× the minimal latency at a load close to 100%. In production use in many different clusters, Swift achieves consistently low tail completion times for short RPCs, while providing high throughput for long RPCs. It has loss rates that are at least 10× lower than a DCTCP protocol, and handles O(10k) incasts that sharply degrade with DCTCP.

CCS CONCEPTS

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

KEYWORDS

Congestion Control, Performance Isolation, Datacenter Transport

ACM Reference Format:
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. 2020. Swift: Delay is Simple and Effective for Congestion Control in the Datacenter. In *Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication (SIGCOMM '20), August 10–14, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3387514.3406591

1 INTRODUCTION

The need for low-latency operations at datacenter scale continues to grow. A key driver is the disaggregation of storage, compute, and memory across the network for cost savings [7, 18, 20, 28, 40, 41, 45]. With disaggregation, low-latency messaging is needed to tap the potential of next-generation storage. For example, industry best practices call for $100\mu s$ access latency at 100k+ IOPS to use Flash effectively [30, 31]. Upcoming NVMe [55, 56] needs $10\mu s$ latency at 1M+ IOPS, else expensive servers sit idle while they wait for I/O [9]. Tight tail latency is also important because datacenter applications often use partition-aggregate communication patterns across many hosts [16]. For example, BigQuery [51], a query engine for Google Cloud, relies on a shuffle operation [11] with high IOPS per server [36]. Congestion control is thus a key enabler (or limiter) of system performance in the datacenter.

In this paper, we report on Swift congestion control that we use in Google datacenters. We found protocols such as DCTCP [1] inadequate because they commonly experience milliseconds of tail latency, especially at scale. Instead, Swift is an evolution of TIMELY [38] based on Google's production experience over the past five years. It is designed for excellent low-latency messaging performance at scale and to meet key operational needs: deploying and maintaining protocols while the datacenter is changing quickly due to technology trends; isolating the traffic of tenants in a shared fabric; efficient use of host CPU and NIC resources; and handling a range of traffic patterns including incast.

Swift is built on a foundation of hosts that independently adapt rates to a *target* end-to-end delay. We find that this design achieves high levels of performance when we accurately measure delay with NIC timestamps and carefully reason about targets, and has many other advantages. Delay corresponds well to the higher-level service-level objectives (SLOs) we seek to meet. It neatly decomposes into fabric and host portions to respond separately to different causes of congestion. In the datacenter, it is easy to adjust the delay target for different paths and competing flows. And using delay as a signal lets us deploy new generations of switches without concern for features or configuration because delay is always available, as with packet loss for classic TCP.

Compared to other work, Swift is notable for leveraging the simplicity and effectiveness of delay. Protocols such as DCTCP [1], PFC [49], DCQCN [59] and HPCC [34] use explicit feedback from switches to keep network queues short and RPC completion times low. They can provide good performance, but they do not help under large incasts and IOPS-intensive workloads. In particular, congestion build-up on hosts is a practical concern that is not addressed. Tight coordination with switches also complicates deployability and maintainability. Other protocols exemplified by pFabric [3], pHost [21], Homa [39], and NDP [23] explicitly schedule RPCs or flows based on their sizes or deadlines. They can deliver good performance for short RPCs, even at high load. Yet they are more complex to implement, deploy, and maintain, especially when there is coordination between switches and hosts. Moreover, they are not a fit for multi-tenant environments, in which a large RPC for one tenant may be of higher priority than a small RPC for another tenant.

We present results from using Swift for business-critical workloads that make up a substantial fraction of Google's network traffic. Swift allows us to maintain low end-host and switch queuing delays in clusters with diverse workloads, including bursty/incast

*The author contributed to this work while at Google.

SIGCOMM '20, August 10–14, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7955-7/20/08.
https://doi.org/10.1145/3387514.3406591

Media	Size	Access time	IOPS	Bandwidth
HDD	10-20TiB	>10ms	<100	120MB/s
Flash	<10TiB	~100μs	500k+	6GB/s
NVRAM	<1TiB	400ns	1M+	2GB/s per channel
DRAM	<1TiB	100ns	–	20GB/s per channel

Table 1: Single-device Storage characteristics

communication patterns. At the network level, Swift delivers high utilization, sustaining a per-server throughput close to 100Gbps (100% load) at the same time maintaining low delay and near-zero loss. As a reference point, the DCTCP loss rate is at least 10x higher from moderate to high load. At the application layer, Swift provides short RPC completion times for intensive storage and analytics workloads. For a demanding in-memory shuffle service [51, 57, 58], Swift achieves average latency close to the baseline delay for short transfers. By handling host congestion effectively, Swift sustains high IOPS even under incasts of O(10k) flows. We detail these results and more in the paper.

We draw several conclusions from our experiences. *First*, delay as a congestion signal has proven effective for excellent performance with a simplicity that has helped greatly with operational issues. In fact, Swift's design has been simplified from TIMELY, as it finds the use of an absolute target delay to be performant and robust. *Second*, it is important to respond to both fabric and host congestion. We initially underestimated congestion at hosts (as have most designs) but both forms matter across a range of latency-sensitive, IOPS-intensive, and byte-intensive workloads. Delay is readily decomposed for this purpose. *Third*, we must support a wide range of traffic patterns including large-scale incast. This range leads us to pace packets when there are more flows than the bandwidth-delay product (BDP) of the path, while using a window at higher flow rates for CPU efficiency.

This work does not raise any ethical issues.

2 MOTIVATION

The evolution of Swift was driven by trends in storage workloads, host networking stacks, and datacenter switches.

Storage Workloads. Storage is the dominant workload for our datacenter networks. It is the primary medium for communication across jobs, and disaggregation means that storage access crosses the network. Disk traffic is dominated by O(10) ms access latency rather than network latency, so carrying disk traffic does not require low-latency congestion control. But latency has become critical as cluster-wide storage systems have evolved to faster media (Table 1). Flash access latency is 100μs, making milliseconds of network latency unacceptable, and NVMe is even more demanding [25]. Tight network tail latency is a requirement because storage access touches multiple devices, and the overall latency for any single storage operation is dictated by the latency of the longest network operation. And in-memory filesystems, e.g., Octopus [35], require multiple round trips for transactions, which also stresses low fabric latency. Moreover, high throughput is needed. Cluster storage systems operate at petabyte scale. Demanding applications such as BigQuery run a shuffle workload on top of an in-memory filesystem [11]. For Swift, the need has been to continually tighten tail latency without sacrificing throughput.

Host Networking Stacks. The implementation of congestion control has undergone a wholesale change in the datacenter. Traditional

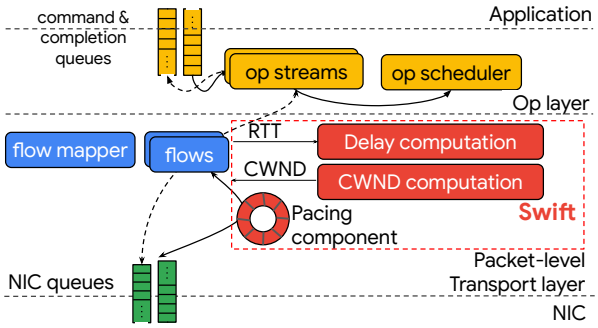

Figure 1: Swift as a packet-level congestion-control in the context of the Pony-Express architecture.

congestion control runs as part of the host operating system, e.g., Linux or Windows, and serves the general purpose use case. This setting is limited by its APIs, e.g., `sendmsg` and `setsockopt`, and is often expensive for innovation. Adding pacing in the kernel, for example, takes 10% of machine CPU [43, 44]. Newer stacks such as RDMA and NVMe are designed from the ground up for low-latency storage operations. To avoid operating system overheads, they are typically implemented in OS bypass stacks such as Snap [36] or offloaded to the NIC [6, 13]. Swift runs in Snap (as described shortly) and lets us design congestion control on a clean-slate with features such as NIC timestamps and fine-grained pacing. Snap also facilitated fast iterations. Swift also inspired a delay-based congestion control scheme to control the issue rate of RDMA operations based on precise timestamp measurements in the 1RMA [47] system. Additionally, addressing host congestion has become critical to maintain low end-to-end queuing. Increasing line-rates and IOPS-intensive workloads stress software/hardware per-packet processing resources; CPU, DRAM bandwidth, and PCIe bottlenecks build up queues in NICs and host stacks. For Swift, delay is decomposed to alleviate both fabric and host congestion.

Datacenter Switches. Our network has several generations of switches in the same fabric; heterogeneity is inevitable given advances in line rates from 10Gbps to 100Gbps and beyond. Tying congestion control deeply to switch internals poses a larger maintenance burden. For example, DCTCP relies on switches to mark packets with ECN when the queue size crosses a threshold. Selecting an appropriate threshold and maintaining the configurations as line speeds and buffer sizes vary is challenging at scale.[1] If we keep the same threshold (in bytes), then a 10× increase in link speed would mark packets 10× earlier in time even though the control loop is not 10× faster. If we grow the threshold, we must consider that absolute buffer size has increased with successive switch generations but is limited by chip area and has not kept pace with line-rates. Tuning is made more difficult because different switches have different ways of managing buffer, e.g., memory bank limitations. For Swift, we have found it easier to evolve delay targets at hosts as part of the march towards lower latency than to integrate with signals from switches.

3 SWIFT DESIGN & IMPLEMENTATION

In this section, we articulate how we settled on Swift and evolved the protocol over time. We avoided switch modifications to more

[1]The DCTCP authors provide a formula to compute marking thresholds [1] but echo our experience that care is needed in production networks.

readily support an evolving and heterogeneous cluster environment. We found existing end-host-based schemes such as DCTCP, D^3 [54], and D^2TCP [52] to be insufficient because of their inability to handle large scale incasts and congestion at hosts. Our observation was that a simple scheme around delay measurements could be sufficient. These high-level requirements guided the evolution of Swift:

(1) Provide low, tightly-bound network latency, near zero loss, and high throughput while scaling to a large datacenter across a range of workloads.
(2) Provide end-to-end congestion-control that manages congestion not only in the network fabric but also in the NIC, and on hosts, i.e., in software stacks. We call the latter endpoint (or host) congestion in this paper.
(3) Be highly CPU-efficient so as to not compromise an otherwise CPU-efficient OS bypass communication.

Latency, loss and throughput are traditional measures of congestion control. Low network latency reduces the completion times of short RPCs. Low loss is critical because loss adds significantly to latency of higher-level application transfer units, e.g., RPCs, especially at the tail. We find two additional measures that are also important in production—endpoint congestion and CPU efficiency.

The design we arrived at uses end-to-end RTT measurements to modulate a congestion window in packets, with an Additive-Increase Multiplicative-Decrease (AIMD) algorithm, with the goal of maintaining the delay around a ***target delay***. Swift decomposes the end-to-end RTT into NIC-to-NIC (fabric) and endpoint delay components to respond separately to congestion in the fabric versus at hosts/NICs.

Swift is implemented in *Pony Express*, a networking stack providing custom reliable transport instantiated in Snap [36]. It uses NIC as well as software timestamps for accurate RTT measurements. It uses Pony Express for CPU-efficient operation and low latency, and as an environment suited to features such as pacing. Figure 1 shows the placement of Swift in Pony Express. Pony Express provides command and completion queue API: applications submit commands to Pony Express, also known as "Ops" and receive completions. Ops map to network flows, and Swift manages the transfer rate of each flow. The overall algorithm is specified in Algorithm 1. We give details below by component.

Algorithm 1: Swift Reaction To Congestion

Parameters: ai: additive increment, β: multiplicative decrease constant, max_mdf: maximum multiplicative decrease factor
$cwnd\_prev \leftarrow cwnd$
$bool\ can\_decrease \leftarrow (now - t\_last\_decrease \geq rtt)$ ▷ Enforces MD once every RTT
On Receiving ACK
  $retransmit\_cnt \leftarrow 0$
  $target\_delay \leftarrow$ TargetDelay() ▷ See S3.5
  if $delay < target\_delay$ **then** ▷ Additive Increase (AI)
    if $cwnd \geq 1$ **then**
      $cwnd \leftarrow cwnd + \frac{ai}{cwnd} \cdot num\_acked$
    else
      $cwnd \leftarrow cwnd + ai \cdot num\_acked$
  else ▷ Multiplicative Decrease (MD)
    if $can\_decrease$ **then**
      $cwnd \leftarrow \max(1 - \beta \cdot (\frac{delay - target\_delay}{delay}), 1 - max\_mdf) \cdot cwnd$
On Retransmit Timeout
  $retransmit\_cnt \leftarrow retransmit\_cnt + 1$
  if $retransmit\_cnt \geq RETX\_RESET\_THRESHOLD$ **then**
    $cwnd \leftarrow min\_cwnd$
  else
    if $can\_decrease$ **then**
      $cwnd \leftarrow (1 - max\_mdf) \cdot cwnd$
On Fast Recovery
  $retransmit\_cnt \leftarrow 0$
  if $can\_decrease$ **then**
    $cwnd \leftarrow (1 - max\_mdf) \cdot cwnd$
$cwnd \leftarrow$ clamp$(min_cwnd, cwnd, max_cwnd)$ ▷ Enforce lower/upper bounds
if $cwnd \leq cwnd\_prev$ **then**
  $t\_last\_decrease \leftarrow now$
if $cwnd < 1$ **then**
  $pacing\_delay \leftarrow \frac{rtt}{cwnd}$
else
  $pacing\_delay \leftarrow 0$;
Output: $cwnd, pacing\_delay$

3.1 Using Delay to Signal Congestion

Delay is the primary congestion signal in Swift because it meets all our requirements. TIMELY noted that RTT can be measured precisely with modern hardware, and that it provides a multi-bit congestion signal, i.e., it encodes the extent of congestion and not only its presence. Swift further decomposes the end-to-end RTT to separate fabric from host issues; it has made delay measurements much more precise through a combination of timestamps in NIC hardware and in polling-based transport like Pony Express. We describe the delay components and how they are measured in Swift.

Component Delays of RTT

Figure 2(a) shows the components that make up an RTT, from locally sending a data packet to receiving the corresponding acknowledgement from a remote endpoint.

- ***Local NIC Tx Delay*** is the time the packet spends in the NIC Tx queue before it is emitted on the wire. When the networking stack uses a pull model [36], the host hands the packet to the NIC when the NIC is ready to send it, so this delay is negligible. Note that the host delay in this situation is not considered part of the packet layer; it is observed at higher layers.
- ***Forward Fabric Delay*** is the sum of the serialization, propagation and queuing delays for the data packet at switches between the source and destination. It also includes the NIC serialization delay.
- ***Remote NIC Rx Delay*** is the time the packet spends in the remote NIC queue before it is picked by the remote stack. This delay can be significant when the host is the bottleneck. For example, in the context of Snap [36], this delay can rise quickly when the packet processing capacity falls due to memory pressure and CPU scheduling.

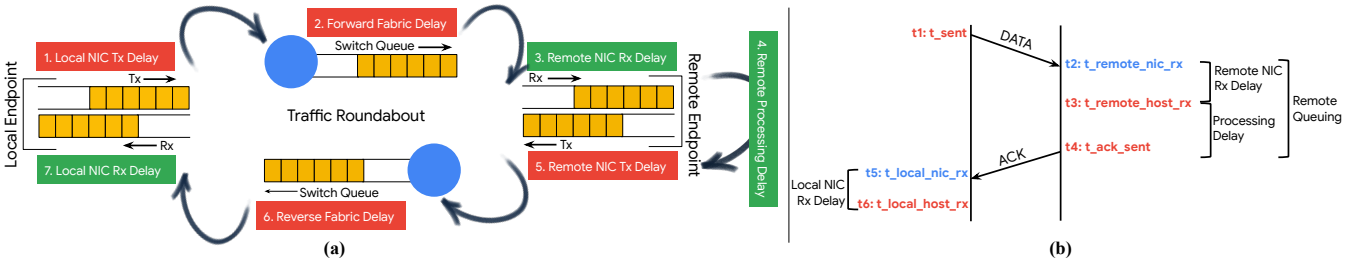

Figure 2: (a) Components of end-to-end RTT for a data packet and corresponding ACK packet. (b) Timestamps used to measure different delays (Hardware and software timestamps are shown in blue and red, respectively).

- ***Remote Processing Delay*** is the time for the stack to process the data packet and generate an ACK packet, including any explicit ACK delays.
- ***Remote NIC Tx Delay*** is the time spent by the ACK packet in the NIC Tx queue.
- ***Reverse Fabric Delay*** is the time taken by the ACK packet on the reverse path. Note that the forward and reverse paths may not be symmetric.
- ***Local NIC Rx Delay*** is the time spent by the ACK packet before it is processed by the stack to mark the delivery of the data packet.

The forward fabric delay is the primary indicator of network congestion, while the remote NIC Rx delay is the primary indicator of host congestion. Sender-based congestion control should not react to reverse path congestion since it has no direct control over it. If needed, ACKs can be prioritized by using a higher quality of service (QoS) class. However, we find the reverse path delays are well controlled in practice, since the reverse side traffic is also controlled by Swift.

Measuring Delays

Swift uses multiple NIC and host timestamps to separate the components of delay. Modern NICs widely support hardware timestamps that are accessible from host networking stacks, including Pony Express and kernel TCP. We describe the implementation in Snap [36]; details depend on the networking stack.

Figure 2(b) depicts the event time sequence. t_1 is the time a data packet is sent, as recorded by the stack. t_2 is when the packet lands on the remote NIC. It is available via the hardware timestamp that the NIC marks on the descriptor. t_3 is when the packet is processed by the stack, thus $t_3 - t_2$ is the time that the packet spends in the NIC queue. The key here is synchronizing the NIC clock (which provides t_2) and the host clock (which provides t_3). We use a simple linear extrapolation algorithm to translate the NIC timestamp to the host clock (Appendix A provides further details). t_4 is the time the ACK is ready to be sent out by the stack, and so $t_4 - t_3$ gives us the processing time.

We sum the NIC Rx delay and the processing time to obtain *remote-queuing*, and reflect this delay to the sender via a header on the ACK packet. The NIC Rx timestamp is appended locally to the packet descriptor and is not sent on the wire. In our experience, 4 bytes are enough for microsecond-level precision. Details of the packet format changes are in Appendix B. Finally, t_5 and t_6 are the corresponding receive timestamps for the ACK at the original sender. End-to-end RTT is $t_6 - t_1$.

3.2 Simple Target Delay Window Control

The core Swift algorithm is a simple AIMD controller based on whether the measured delay exceeds a target delay. We found simplicity to be a virtue as TIMELY evolved to Swift and removed some complexity, e.g., by using the difference between the RTT and target delay rather than the RTT gradient. Below we describe the algorithm based on a fixed target delay, then elucidate the end-host and fabric parts in §3.3, and dynamic scaling based on topology and load in §3.5.

The controller is triggered on receiving ACK packets. Swift reacts quickly to congestion by using ***instantaneous delay*** as opposed to minimum or low-pass filtered delay. In addition, Swift does not explicitly delay ACKs. Both choices mitigate staleness concerns in using delay as a congestion-signal [60]. Lines 4–14 in Algorithm 1 provide Swift's reaction on receiving an ACK; if the delay is less than the $target$, the $cwnd$ (measured in packets) is increased by $\frac{ai}{cwnd}$ (ai = *additive increment*), such that the cumulative increase over an RTT is equal to ai. Otherwise, the $cwnd$ is decreased multiplicatively, with the decrease depending on how far the delay is from the target, i.e., we use multiple bits of the delay signal for precise control. The multiplicative decrease is constrained to be once per RTT, so that Swift does not react to the same congestion event multiple times. We do this by checking against the time of the last $cwnd$ decrease. The initial value of $cwnd$ has little effect in our setting because Pony Express maintains long-lived flows.

3.3 Fabric vs. Endpoint Congestion

Many congestion control designs focus on the fabric as the network, and ignore host issues. We learned over time that host issues are important and need a different congestion response. To do so, we split the RTT into fabric delay due to links and switches, and end-host delay that happens in NIC and host networking stack. First, Swift computes *endpoint-delay* as the sum of *remote-queuing* (echoed in the ACK) and Local NIC Rx Delay (given by $t_6 - t_5$).[2] Then, Swift computes *fabric-delay* as *RTT* minus *endpoint-delay*.

Swift then uses two congestion windows, *fcwnd* to track fabric congestion, and *ecwnd* to track endpoint congestion. Both windows follow Algorithm 1 with a different *fabric-delay-target* and *endpoint-delay-target*. There is a slight difference in that we use Exponentially Weighted Moving Average (EWMA) filtering for the endpoint delay, given that endpoint delays are more noisy in our experience.

The effective congestion window is combined as min(*fcwnd*, *ecwnd*).[3] Note the similarity to how TCP uses the minimum of *cwnd* and receiver advertised window, where advertised window serves the role of *ecwnd*. In the context of Snap [36], delay is a

[2] We provide the reasoning behind including this delay in Appendix C.
[3] Both $cwnds$ are updated together and a ceiling value is used as a guard for the non-bottlenecked $cwnd$.

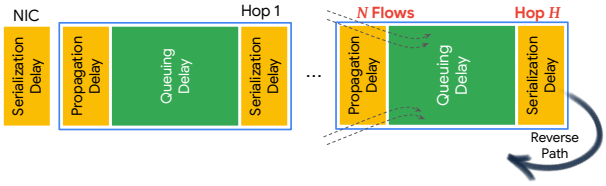

Figure 3: Target delay encapsulates both fixed and variable parts and is dynamically scaled based on topology and load.

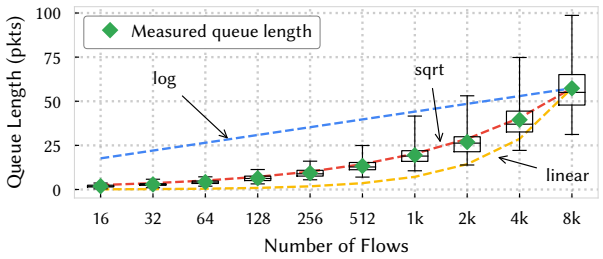

Figure 4: Average queue buildup with randomized flow arrival and perfect rate control grows as $O(\sqrt{N})$.

better measure of host congestion than advertised window. It is directly tied to all bottlenecks on the host, including CPU, memory, PCIe bandwidth, caching effects, thread scheduling, etc., whereas advertised window captures memory allocation (and very indirectly CPU bottlenecks). In addition, the advertised window is used for flow-control, i.e., to prevent a flow from over-running a buffer, and does not aim for fairness across flows when the host is the bottleneck.

Separating fabric and host congestion in the design of Swift had a huge impact in production, with the tail latency of most applications improving by 2×, and none suffering a regression. We give more production results in §4.

3.4 Large-Scale Incast

During deployment, we ran into applications that relied on extremely large incasts, with thousands of flows destined to a single host simultaneously. In this scenario, when number of flows exceed the path BDP, even a congestion window of one is too high to prevent overload. To handle such cases, we augmented Swift to allow the congestion window to fall below one packet down to a minimum of 0.001 packets. This case needs special handling of the increment update (Lines 7–11 of Algorithm 1). To implement a fractional congestion window, we translate it to an inter-packet delay of $\frac{RTT}{cwnd}$ (Lines 29–32 of Algorithm 1) that the sender uses to pace packets into the network. For example, a *cwnd* of 0.5 results in sending a packet after a delay of $2 \times RTT$. The pacing is implemented using a Timing Wheel [43]. Results from production (§4) show pacing is critical to maintain low latency and loss at scale.

While conventional wisdom is that always-on pacing is beneficial in terms of smooth traffic and lower losses, we found that pacing packets for moderate or higher flow rates did not provide better performance than an ACK-clocked window. Moreover, pacing packets is not CPU efficient compared to ACK clocking. Beyond the CPU cost of pacing data packets, added CPU is consumed on the receiver due to reduced opportunities for ACK coalescing, and on the sender due to a corresponding increase in the number of ACKs. TIMELY used rate control but did not suffer from these problems because it paced 64KB chunks, which allowed for efficient use of CPU. But for a Snap transport that operates in MTU-sized units, pacing is mostly not necessary for performance, nor is it CPU-efficient. In Swift, we finesse this issue by normally using ACK-clocked congestion window and shifting to pacing when the *cwnd* falls below 1.

3.5 Scaling the Fabric Target Delay

So far, we have described Swift with a fixed target delay. Here, we describe how to scale the target fabric delay (henceforth referred to as target delay) to the latency of paths that are longer or heavily loaded.

Target delay encapsulates both the fixed and the variable parts of the fabric delay, as shown in Figure 3. The base portion of target delay consists of delays incurred for a single hop network with a small number of flows: propagation delay, serialization delay in NIC and switch (which depends on link speed), queuing delay for a small number of flows, measurement errors from software and hardware timestamps, as well as any unaccounted delays in network, e.g., resulting from QoS scheduling. On top of this base, we scale target delay based on topology and load.

Topology-based Scaling: While using a single target delay that is high enough to cover propagation and serialization delays across the datacenter diameter gives us good overall throughput, it comes at the cost of building larger queues for traffic that takes shorter paths, e.g., intra-top-of-rack (intra-ToR) network, plus some RTT unfairness. Instead, we want to use smaller targets for flows with shorter paths to improve performance.

Measuring the minimum path delay is not simple in the Internet, as shown by prior work [12]. For datacenters, the topology is known and network distance is bounded. Given this environment, we translate the network path for a flow to a target delay by using a fixed base delay plus a fixed per-hop delay. We measure the forward-path hop count by subtracting the received IP TTL (Time-To-Live) values from known starting TTL, and reflect it back in the ACK-header. This design works with multi-path forwarding, though in practice we find it sufficient to use a single path at a time per flow.

Flow-based Scaling: We also scale the target delay with the number of competing flows. The target must provide enough headroom for Swift to fully utilize the bottleneck. We find that the queue size and hence the target required to saturate the bottleneck link increases with the number of competing flows. For intuition, consider a link with N flows that are rate-limited to exactly their fair share but have random start times. Then queuing happens only when packets from different flows come together by chance. Figure 4 shows the simulation results—the average queue length grows as $O(\sqrt{N})$. This behavior can be modeled as a bounded random walk and the average queue size, like the distance from the starting point, grows as $O(\sqrt{N})$. Results from buffer sizing work [5] that show required buffer space for TCP *reduces* with number of flows by $O(\sqrt{N})$ may seem contradictory but it is modeling different aspects. The buffer sizing work is modeling the *variation* in window as governed by AIMD. Using Central Limit Theorem, Reference [5] shows that the variation in total window is reduced by $O(\sqrt{N})$ for large number of unsynchronized flows.[4] In Swift, we model the amount of queuing that would happen naturally due to random

[4] Another way to reason: given N flows, each flow's window fluctuation is $O(\frac{1}{N} \times BDP)$; chance of fluctuations coinciding is $O(\sqrt{N})$. Thus, average fluctuation of total window is $O(\frac{\sqrt{N}}{N}) = O(\frac{1}{\sqrt{N}})$.

Parameter	Description
$base\_target$	base target delay
$\hbar$	per hop scaling factor
fs_max_cwnd	max cwnd for target scaling
fs_min_cwnd	min cwnd for target scaling
fs_range	max scaling range

Table 2: Parameters for target delay scaling.

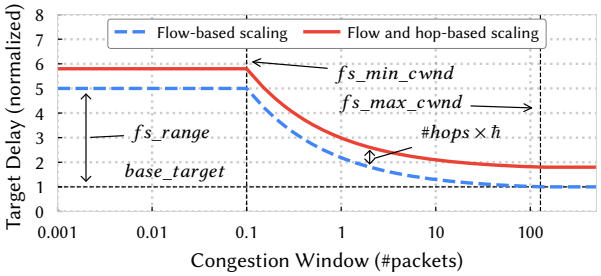

Figure 5: Example fabric delay target curve vs congestion window.

chance of collision between flows but not because of bandwidth overestimation, and factor it into the queuing headroom to avoid overreaction.

Since the sender does not know the number of flows at the bottleneck, we need another means to adjust the target. We rely on the fact that *cwnd* is inversely proportional to the number of flows when Swift has converged to its fair-share. So we adjust the target in proportion to $1/\sqrt{cwnd}$, i.e., the target delay grows as *cwnd* becomes smaller. As well as lowering the queuing when there are few flows, this method improves fairness: it speeds slow flows with a larger target, and slows fast flows with a smaller target. We find this convergence bias especially useful when flows have congestion windows less than one.

Overall Scaling: Combining topology and flow scaling, and using the notation of Table 2, we arrive at the formula for target delay:

$$t = base\_target + \#hops \times \hbar + max(0, min(\frac{\alpha}{\sqrt{fcwnd}} + \beta, fs\_range)),$$

where

$$\alpha = \frac{fs\_range}{\frac{1}{\sqrt{fs\_min\_cwnd}} - \frac{1}{\sqrt{fs\_max\_cwnd}}}, \quad \beta = -\frac{\alpha}{\sqrt{fs\_max\_cwnd}}.$$

Figure 5 provides an example to show the relationship between target delay, *cwnd*, and flow scaling parameters. $base\_target$ is the minimum target delay required to provide 100% utilization in a one hop network with a small number of flows. fs_range specifies the *additional* target on top of *base* that is progressively reduced over the *cwnd* range $[fs\_min\_cwnd, fs\_max\_cwnd]$. $\hbar$ is the per-hop scaling factor. These three parameters determine the slope of the scaling curve and the *cwnd* range to which it is applied.

A steeper curve improves fairness and convergence by making it more likely that slower flows increment their rates. However, it also increases queuing in the network. To precisely model queuing due to random collisions, the slope of the curve is a function of number of flows and link capacity. We approximate the ideal curve by selecting a tolerable delay over base target, and a *cwnd* range over which to apply scaling. Defining a scaling range lets us use a steeper curve for smaller *cwnd*, representing the high congestion regime.

We note that the interdependence of *cwnd* and target delay is for the small *cwnd* regime. It does not pose a problem in practice because the target delay and *cwnd* adjustments are opposite in direction, e.g., when *cwnd* grows, the queuing delay increases which reduces target delay, and vice versa. The result is to more quickly equalize the queuing and target delay than with a static target, which gives faster convergence and better fairness.

3.6 Loss Recovery and ACKs

We discuss the details of generating acknowledgments and recovering lost packets because they impact delay-based congestion control.

Loss Recovery Happily, we have needed to invest minimal effort in loss recovery for good tail latency because Swift keeps packet losses low. Like TCP, packet losses are detected in Swift with two main mechanisms: selective acknowledgements (SACK) for fast recovery, and a retransmission timer to ensure data delivery in the absence of acknowledgments from the receiver. SACK is implemented using a simple sequence number bitmap. When a packet is detected as lost via a hole in the bitmap, it is retransmitted, and the congestion window is reduced multiplicatively. In addition, a retransmission timeout (RTO) is maintained on a per-flow basis computed using exponentially weighted moving average over the end-to-end RTT. To adapt quickly to potentially severe congestion, the congestion window is reduced by the maximum multiplicative factor on an RTO. We have not needed to draw on other mechanisms well-known from TCP loss recovery, e.g., References [4, 10, 19, 37].

ACKs Swift does not explicitly delay ACKs to react more quickly to congestion. Note that ACK coalescing will still occur if multiple packets arrive together, e.g., they are processed in a batch by Snap. We also take care not to delay ACKs in the case of bi-directional traffic, as would happen if we piggyback the ACK on a reverse data packet that is paced. Instead, we decouple data and ACK packets for paced flows—an incoming data packet generates a pure ACK sent immediately to unblock the remote end, while a reverse data packet respects any pacing-delay that is imposed on it.

3.7 Coexistence via QoS

In a shared production deployment, there are multiple congestion control algorithms. WAN flows operate with a different congestion control algorithm than datacenter flows optimized for latency. Customers configure cloud VMs with the congestion control of their choice. And UDP-based traffic uses application-level rate control logic. It is essential that Swift traffic be able to coexist with various forms of congestion control without adverse competition for switch buffers, otherwise its latency may be inflated and its throughput reduced.

As a pragmatic solution, we leverage QoS features. Switches have ~10 QoS queues per port [32] that can share the buffer space across ports based on usage. We reserve a subset of QoS queues for Swift traffic and give them a share of the link capacity via weighted-fair-queuing. By using larger scheduler weights for higher priority traffic, we are able to handle tenants with different traffic classes. While this simple arrangement does not completely isolate Swift traffic, we show it provides enough separation for excellent performance (§4.3).

4 TAKEAWAYS FROM PRODUCTION

We deployed Swift in production at Google over the course of four years. It supports traffic at large scale from applications with a range of needs including:

- HDD and SSD reads and writes that serve the storage needs of many Google applications. While there are many small reads and writes, this traffic is generally throughput-intensive (also referred to as byte-intensive in this paper).
- An in-memory key-value store used by several Google applications, which is latency-sensitive.
- An in-memory filesystem used for BigQuery shuffle, which is IOPS-intensive and can have a large degree of fan-in.

4.1 Measurement Methodology

Data reported in this section is taken fleet-wide, except where we call out specific clusters, for a period of one week. This data covers a very wide range of workloads, scale, and utilization. We draw our conclusions from three types of data:

- **Switch statistics** tell us the link utilization and loss rates. We compute them over 30-second intervals using per-port *output-packets* and *output-discards* counters collected from production switches. We do not include loss due to routing failures or packet corruption.
- **Host round-trip times**. NIC-to-NIC probers with NIC hardware timestamps give us fabric RTT data for both Swift and non-Swift traffic. End-to-end packet RTT, including host and fabric parts, is measured in Swift.
- **Application metrics**, where available, highlight the impact of Swift on applications.

Our main point of comparison for Swift is DCTCP-style congestion control. While there are many other proposed congestion control protocols for the datacenter, they are more complex and not readily available at fleetwide scale. In contrast, DCTCP is a well-known reference point, for which we report on a Google version called **GCN** that has a faster response to congestion by scaling its multiplicative decrease based on the current ECN mark rate rather than an EWMA, and disabling delayed ACKs when receiving ECN-marked data. GCN has also been thoroughly tuned at scale. Thus, we believe GCN serves as a more stringent comparison for Swift than DCTCP. The GCN results in this section are with our kernel production deployment; we also provide results for GCN instantiated in PonyExpress [36] for comparison in §5.3.

For confidentiality purposes, we normalize the absolute loss, latency and throughput numbers from our production deployments; in §5 we report absolute numbers for Swift performance from testbed experiments.

4.2 Performance At Scale

We report combinations of latency/loss and throughput/utilization because it is essential to deliver on all of them at once—it is easy but not useful to optimize one metric at the expense of another. A key difficulty is how to perform a comparative fleetwide evaluation as we cannot easily shift between these protocols to provide clean A/B data at fleet scale. Rather, both Swift and GCN must run together in various mixes in the same cluster. We report Swift/GCN loss rates versus total *port* utilization as well as individual Swift/GCN utilizations (which we call *queue* utilizations) to show that Swift delivers excellent performance.

Takeaway: Swift achieves low loss even at line-rate

One of the biggest improvements as we moved traffic to Swift was the reduction in packet loss. We compare Swift and GCN loss rates versus the combined link utilization of Swift and GCN. The loss rate is the lost packets divided by the sent packets for a type of traffic (Swift or GCN). The combined link utilization is the sum

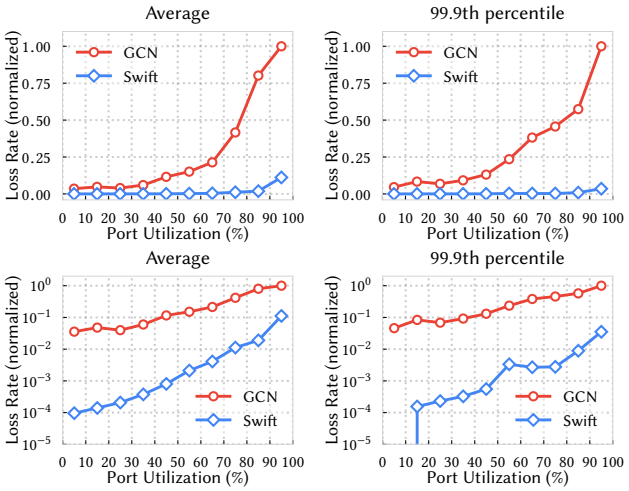

Figure 6: Edge (ToR to host) links: Average and 99.9p Swift/GCN loss rate (linear and log scale) vs. combined utilization, bucketed at 10% intervals. Loss rate is normalized to highest GCN loss rate. The near-vertical line in the log-scale plot is due to extremely small relative loss-rate.

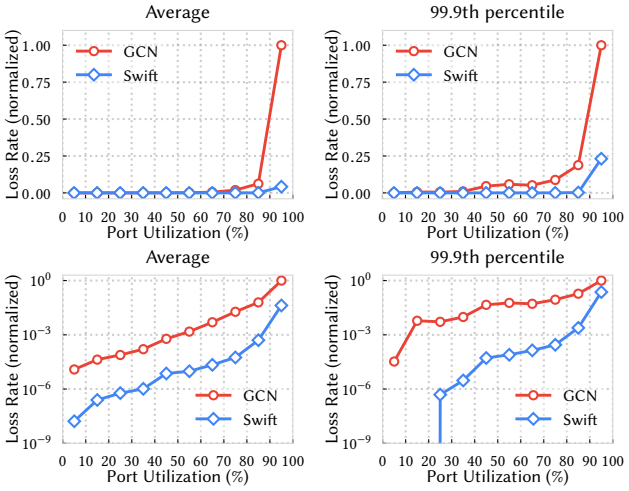

Figure 7: Fabric links: Average and 99.9p Swift/GCN loss rate Swift/GCN loss rate (linear and log scale) vs. combined utilization, bucketed at 10% intervals.

of Swift/GCN bits on the wire divided by the link capacity. We use switch counters for both metrics. We normalize the highest GCN loss rate in each plot to 1.0.

Figure 6 shows links at the edge, i.e., ToR to host. We see in the log-scale plot that Swift provides 2+ orders of magnitude lower average and 99.9th-percentile loss rates than GCN across a range of combined utilization. While the data is normalized due to confidentiality, we note that GCN losses have been an operational challenge, even at lower utilization levels. Swift continues to provide very low average and tight tail loss for heavily utilized (>90%) links at the edge, while GCN does not.

Figure 7 shows the same plot for fabric links in the core of the datacenter. We see the same trends. Note that not all loss in production is due to congestion control, e.g., link flaps cause loss before the switch converges. This impact is evident at high utilization in the fabric, especially for 99.9p loss, where there is a slight uptick in Swift loss rate.

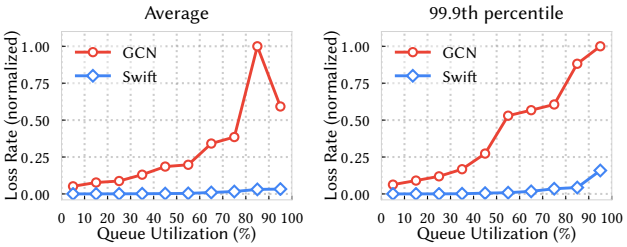

Figure 8: Average and 99.9th percentile loss rate vs. queue utilization.

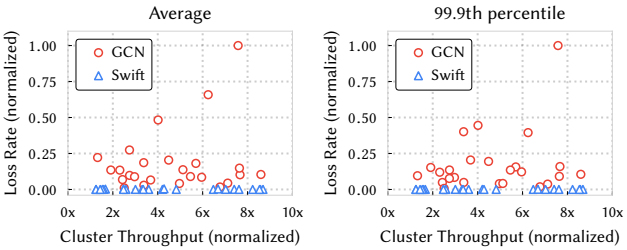

Figure 9: Edge (ToR to host) average and 99.9p loss rate vs. total Swift/GCN throughput in the cluster.

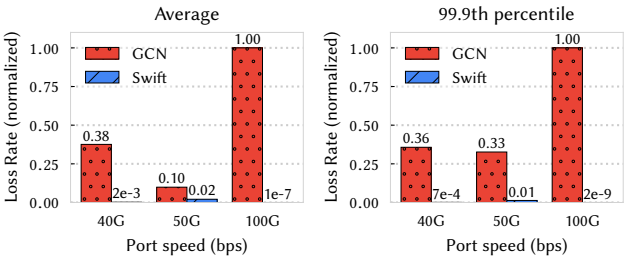

Figure 10: Average and 99.9p loss rate of highly-utilized (>90%) links in each switch group.

As a check, Figure 8 shows Swift/GCN loss versus the Swift/GCN *queue utilization*.[5] and see the same behavior. This gives us confidence that low Swift loss rates are sustained even when all the traffic on the link is from Swift.

We plot aggregate cluster throughput versus the loss rate in Figure 9. We pick 25 clusters at Google and plot the loss rate for Swift/GCN separately against the total Swift/GCN throughput in the cluster. We report edge links only for brevity. We see that Swift consistently delivers low loss even at extreme tails at scale, while GCN is much more variable. In our experience, the low loss rates of Swift are a direct outcome of its prompt reaction to congestion as detected by its target delay, as well as scaling to large incasts. Given the extremely small loss rate both at the edge and in the fabric, the end-to-end retransmission rate for Swift is also very low, which is consistent with our choice not to invest heavily in loss recovery.

Swift's performance improvements hold for a range of link speeds. We show results for commodity NIC link speeds from 40 to 100Gbps. Figure 10 compares the average and 99.9p loss rate for edge (ToR to host) links at high utilization (>90%). At near line-rate utilization, even the 99.9th-p loss rate for Swift is much smaller than GCN.

Takeaway: Swift achieves low latency near the target

We now turn to latency. Figure 11 shows NIC-to-NIC round trip time (or fabric RTT) across our datacenters as measured by NIC

[5]Note that there is a spike in the [80-90%] loss rate bucket for GCN. We verified this spike accurately measures our production traffic, and we are investigating plausible causes.

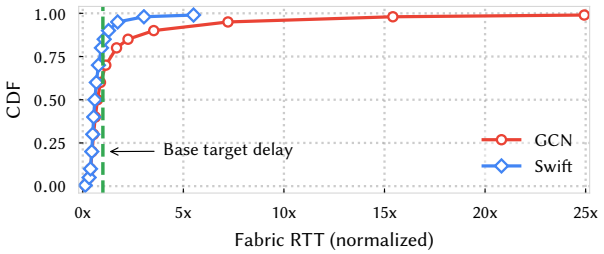

Figure 11: Fabric RTT: Swift controls fabric delay more tightly than GCN.

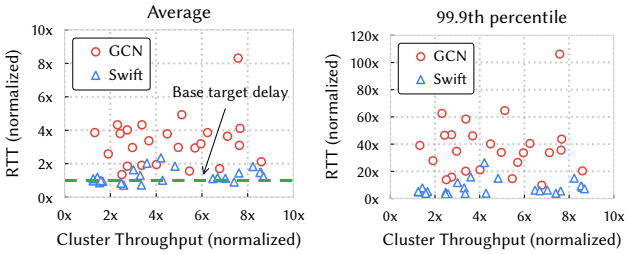

Figure 12: Cluster Swift/GCN Throughput vs. Average RTT. The dashed line is the *base target delay* (normalized to 1).

timestamps. We can see that Swift is able to maintain the average fabric round-trip around the configured target delay, and controls tail latency much better than non-delay based GCN. We normalize the numbers w.r.t the *base target delay*.

In Figure 12, we show that Swift achieves average RTTs close to target delay at large scale and across our clusters. The average RTT roughly matches the base target delay used in our deployments. This behavior has proven to be extremely useful as we tuned the target delay over the course of our full deployment—early Swift deployments used **2×** the base target delay than our current configuration. This change was done incrementally and carefully, ensuring we did not cause regression in application performance by decreasing the bandwidth applications get as we navigate the latency-throughput tradeoff. §5.1 details the experiments that guided us in setting the base target delay in production.

We note that in some clusters the average RTT is above the base target, though it stays below the upper bound of the target with topology and flow scaling. We spot-checked a few clusters and found that the average is drivenup by the tail RTT for two reasons. First, external factors impact the delay observed for Swift. The main factor is a large amount of traffic on high QoS classes outside of Swift control. These factors are inevitable in a large, heterogeneous, shared infrastructure. Second, the tail RTT is also driven up by heavy incast workloads that trigger flow-based scaling of target delay as described in §3.5.

Thus, Swift achieves near line-rate throughput while maintaining its promise of low loss and low latency.

Swift outperforms GCN for three main reasons. First, Swift rapidly reduces its *cwnd* to below one under extreme congestion, when number of flows exceed the network bandwidth-delay product. Second, Swift alleviates congestion at end-hosts in addition to congestion in the fabric. Finally, Swift predictably bounds end-to-end delay regardless of intermediate link rates and buffer sizes, which is a difficult goal to attain with GCN thresholds.

4.3 Use of Shared Infrastructure

Recall that we share network links with non-Swift traffic. To do so we separate traffic using QoS classes that share link bandwidth

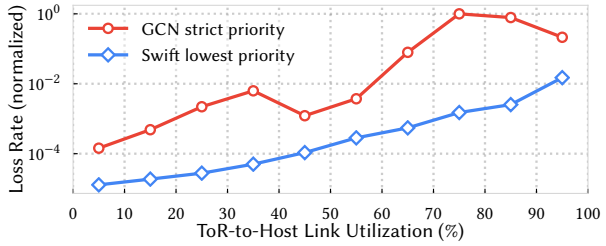

Figure 13: Average loss rate vs. port utilization for GCN traffic at strict scheduling priority and Swift at lower scheduling weight for ToR-to-host links. The highest GCN loss rate is normalized to 1.0.

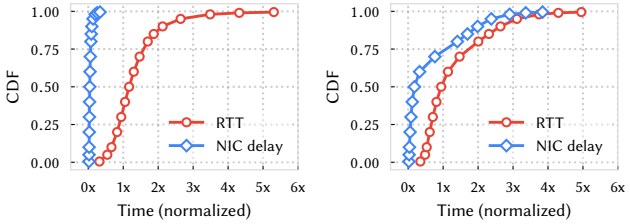

Figure 14: CDF of end-to-end packet RTT and NIC-Rx-queuing delay for the throughput-intensive cluster (left) and IOPS-intensive cluster (right).

using weighted-fair queuing. In our experience, this arrangement lets Swift achieve low latency even in cases when a good portion of the traffic is controlled by loss or via GCN. We see in Figure 11 that Swift latency is substantially smaller than the GCN latencies when both queues have the same scheduling priority. Similarly, the cluster-scale measurements in Figure 12 show an order of magnitude lower latency for Swift than GCN.

To stress the isolation mechanism, we compare the packet loss versus port utilization for unequal scheduling priority: GCN running with the advantage of strict priority scheduling and Swift running at its lowest weight. Figure 13 shows that Swift controls the queueing much better than GCN even though it has less preferred access to link bandwidth.

4.4 Fabric and Host Congestion

The response of Swift to host as well as fabric congestion has been key to its success in production. In a shared environment, we often have a mix of IOPS-intensive applications that stress hosts, and throughput-intensive applications that stress the fabric. While most cluster congestion is in the fabric, host congestion is not rare, and if we did not respond to it separately then IOPS-intensive flows could unfairly degrade co-located throughput-intensive flows.

We choose two clusters to show how the fabric and host both contribute to end-to-end packet RTT. One cluster is dominated by IOPS-intensive tasks that are end-host queuing dominated, while the other carries predominantly large storage RPCs, which are typically network queuing dominated. For each cluster, we split the end-to-end packet RTT as measured in Swift (t_6 - t_1 in Figure 2) to obtain fabric and host components. We show these components for the two clusters in Figures 14. There is a clear distinction. The host delays are small and tight for the throughput-intensive cluster, but can contribute as much as the fabric delays in the IOPS-intensive cluster.

Splitting the RTT into fabric and host components has also been invaluable for debugging. When a network problem is reported in production, the first step is typically to determine whether the culprit is the fabric or the host. By looking at log data for fabric

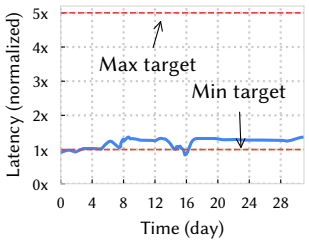

Figure 15: Small Op latency in an in-memory filesystem.

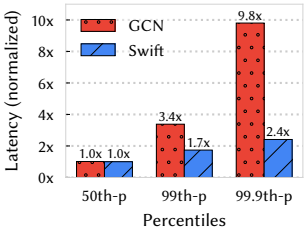

Figure 16: Op latency in an SSD storage system.

versus endpoint congestion in aggregate also tells us whether a cluster is congested mostly by the fabric or end host engines.

4.5 Application Performance

It is important that Swift support IOPS-intensive and throughput-intensive applications, latency sensitive workloads with short Ops, and allow them to run well together; coexistence facilitates service deployment in a shared network environment.

In-memory BigQuery Shuffle. Swift supports a disaggregated, in-memory filesystem for BigQuery shuffle [11] built atop Snap [36]. What ultimately matters to a memory-based file system is how quickly the workload completes, as measured by the IOPS, and the completion time of small Ops. Swift's ability to simultaneously control network delays and host congestion at scale is thus invaluable for this application. Figure 15 shows that the Op completion time closely follows the Swift's target delay.[6] The separate treatment of fabric and host congestion in Swift was key in enabling this application to meet its access latency SLOs in all clusters. In addition, the team informed us that the ability to keep network latencies small provided meaningful backpressure to them. They recently rolled out a change to handle application-level queuing better, which improved the tail-latency by 7×; a change which would not have been possible with GCN.

Storage. Parts of Storage traffic are also served over Swift, with throughput as the primary metric of goodness. We provide results from a load test provided to us by the SSD-storage team that does 16kB reads. Swift achieves 4× lower 99.9th-p application latency vs. GCN as shown in Figure 16. Additionally, Swift achieves ~7% higher IOPS with a 100% success rate in Op-completions, while the losses in GCN resulted in 1.7% of its Ops failing due to deadlines being exceeded.

4.6 Production Experience

We briefly summarize production experiences that may be of interest.

Swift's ability to operate at near line-rate with near zero loss has caused confusion at times, since other teams are not used to this level of performance at scale. On two occasions, bugs were incorrectly filed for monitoring failures because highly-utilized links reported zero loss. Both cases were quickly attributed to Swift; the use of QoS classes separation made this attribution easy.

Swift's extremely low delays has also met with skepticism that it may be unnecessarily sacrificing throughput, thereby reducing application performance. It is relatively easy to measure how well congestion-control is mitigating congestion, but it is much harder to measure how well it is utilizing available bandwidth given that

[6] High IOPS is not shown here; Reference [36] shows that in some intervals a single Snap instance is serving upwards of 5M IOPS.

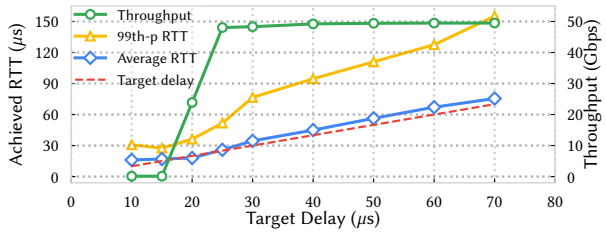

Figure 17: T_1: Achieved RTT and throughput vs. target delay, 100-flow incast.

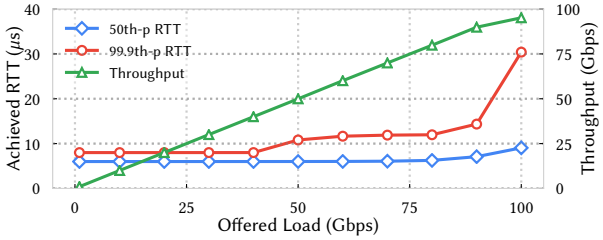

Figure 18: T_2: Achieved RTT and throughput vs. per-machine offered load. Total load varies from 500Gbps to ~50Tbps.

Metric	Swift w/o $cwnd < 1$	Swift
Throughput	8.7Gbps	49.5Gbps
Loss rate	28.7%	0.0003%
Average RTT	2027.4μs	110.2μs

Table 3: T_1: Throughput, loss rate and average RTT for 5000-to-1 incast with and without $cwnd < 1$ support.

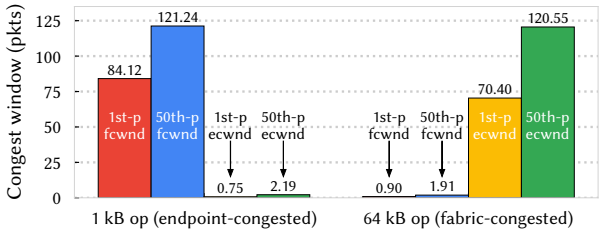

Figure 19: T_1: Fabric ($fcwnd$) and Endpoint ($ecwnd$) congestion windows for a 100-to-1 incast with 1kB and 64kB writes.

traffic may be application-limited or bottlenecked at end-hosts. To address this concern we rolled out experimental and more aggressive versions of Swift to subsets of the fleet. These versions prioritized throughput at the cost of increased queuing, including raising the target delay, disabling pacing and $cwnd < 1$ mode, raising the window floor and shortening timeouts. We observed increased RTT and loss, but with no increase in throughput or improvement in application performance. This result corroborates that Swift is not throttling traffic to keep latency and loss low.

To see that the Swift implementation is efficient, we checked its CPU usage in a diverse, heavy workload. It accounted for ~2.6% of Pony Express CPU. Of this, 1.4% is for ACK-processing (which includes code not specific to Swift) and 0.31% is spent to support timestamping and delay measurement.

We also found it simple to tune Swift's target delay in production, in contrast with challenges in tuning ECN thresholds. ECN thresholds requires switch configuration updates, which is an ongoing task due to evolving line-rates and topologies, and complicated by heterogeneity, e.g., switches with a mix of line rates. In comparison, target delay is controlled at the hosts and has a clear end-to-end interpretation.

5 EXPERIMENTAL RESULTS

We present results from controlled experiments to evaluate the mechanisms in Swift. Our experiments are from two testbeds with benchmarking data (and no production traffic) running on them:

- T_1 has 60 machines with 50Gbps NICs. We use it for incast scenarios and for experiments relating to fairness, target delay scaling and fabric vs. endpoint congestion.
- T_2 is a larger testbed with ~500 machines and 100Gbps NICs. We use it for larger experiments with all-to-all traffic patterns.

5.1 Effect of Target Delay

Target delay is the key control parameter in Swift; we report on how it affects performance. First, we look at how well the Swift protocol can match measured RTT to the specified *base target delay*. In T_1, we set up 10 sender machines with 10 flows per sender, each pushing 64kB write RMA operations to a single receiver machine as quickly as possible. We vary the base target delay from 15μs to 70μs. We disable flow and topology scaling (§3.5) for this experiment to highlight the impact of base target delay. Figure 17 plots the achieved RTT. We see it closely tracks the configured base target delay.

Next, we look at how target delay affects throughput. The base delay must at a minimum cover the propagation and NIC/switch serialization delays, along with measurement inaccuracies. Beyond this minimum, a higher target allows for more queuing. We want the target delay to be low to reduce latency but high enough to maximize network throughput. Figure 17 also shows how the throughput varies with the target delay. Initially the target is too low and Swift throttles throughput. After a narrow transition region, throughput remains saturated above a target of ~25μs. This clean behavior makes it easy to find a good target in practice. In our initial deployments, we used a liberal target to first ensure throughput. We then trimmed it to achieve lower latency, while ensuring no reduction in throughput.

Note that needing a 25μs target in this experiment is not a limitation of Swift but of the networking stack. Appendix D shows an experiment (from a prototype that reduces host CPU interference) where we lowered the target delay to 15μs and still sustained line-rate throughput of 100Gbps.

5.2 Throughput/Latency Curves

Given a target delay, we sweep the offered load to characterize the operating points of throughput vs. latency. To do this we use T_2 with a uniform random traffic pattern: each client selects one machine from the remote racks at random for a 64kB write RMA operation. We set the target delay to 25μs. We vary the offered load by varying the interval over which we issue operations, and measure the fabric RTT with NIC timestamps.

We see in Figure 18 that throughput increases with little rise in RTT until we exceed 80% of the line-rate. Even then, the rise of median RTT is modest and the tail RTT diverges slowly from the median all the way through full load. At a load close to 100%, Swift is able to maintain the 99.9th-p RTT to be <50μs at an aggregate load close to 50Tbps. We note that the tail RTT is at most 3× higher than the baseline unloaded RTT.

5.3 Large-scale Incast

Swift supports $cwnd < 1$ with pacing to handle large-scale incast, which is an important datacenter workload. In T_1, we start 100 flows

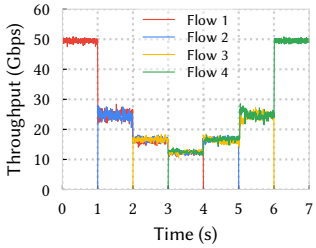

Figure 20: T_1: Throughput of four flows shows fairness achieved by Swift.

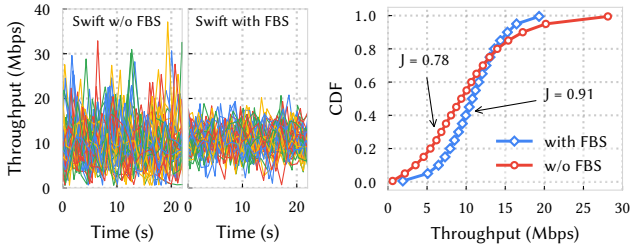

Figure 21: T_1: Throughput with and without flow-based scaling (FBS) for a 5000-to-1 incast. Jain's fairness index (J) shown is measured amongst all 5000 flows using a snapshot of flow rates.

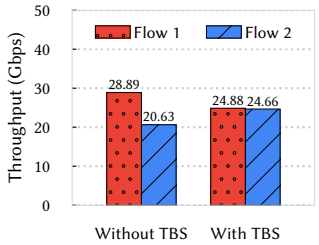

Figure 22: T_1: Throughput of two flows with different path lengths, with and without topology-base scaling (TBS).

from each of 50 machines to a single destination machine, representing a 5000-to-1 incast. We measure the achieved throughput, loss rate and average RTT for runs with and without the support for $cwnd < 1$.

We see in Table 3 that Swift achieves line-rate throughput with low latency and almost zero loss—excellent performance for a 5000-to-1 incast. Conversely, without support for $cwnd < 1$ the protocol degrades to high latency and loss that drives down throughput. In comparison, GCN running in Pony Express achieves a 5.1% loss-rate and 8.67× average RTT even for a smaller scale 1000-to-1 incast.

5.4 Endpoint Congestion

As discussed before, endpoint congestion has become increasingly important with rising link rates and advent of IOPS-intensive applications. To show how Swift differentiates and handles both fabric and endpoint congestion, we use 20 machines in T_1 each with 5 flows incast to a single destination machine. We use 1kB writes for an IOPS-intensive workload and 64kB writes for a byte-intensive workload. We plot the fabric and endpoint congestion windows, $fcwnd$ and $ecwnd$, showing the median and tail values in Figure 19. For the tail, we use the 1st-percentile since lower congestion windows represent smaller sending rates and, hence, larger RPC latencies. We see a clear distinction as the workload shifts: the IOPS-intensive case is limited by the endpoint window, while the byte-intensive case is limited by the fabric window.

Decomposing the end-to-end RTT into fabric and endpoint components lets Swift to craft different responses to congestion at network and hosts. To show this strategy has a performance benefit, we compare Swift to a version (Swift-v0) that uses a single target for end-to-end delay. We setup two concurrent incasts, one for the fabric and one for endpoint. Three machines in T_1, each with 5 flows, send traffic to another receiver machine in T_1. Two of the three senders perform large (64kB) writes to create fabric congestion, and the remaining one performs small (64B) writes to stress the endpoint. We show the results in Table 4. We observe that Swift (fabric base target of 50μs and engine target of 100μs) achieves throughput close to line rate. Swift-v0 with an end-to-end target of 100μs achieves much lower throughput. This happens because the machines performing large writes inappropriately reduce their congestion windows due to congestion at the receiver. We can increase the target for Swift-v0 to increase throughput, but it comes at the cost of increased RTT. Swift-v0 with 200μs target achieves line-rate throughput but observes higher average and 99th-p RTT.

5.5 Flow Fairness

Flow and topology scaling help Swift to achieve a fair allocation of bandwidth across flows, regardless of whether the flows have different path lengths.

Configuration	Throughput	Average RTT	99th-p RTT
Swift	48.7Gbps	129.2μs	175.1μs
Swift-v0, 100μs target	41.6Gbps	118.3μs	154.4μs
Swift-v0, 150μs target	44.9Gbps	157.6μs	203.8μs
Swift-v0, 200μs target	49.5Gbps	184.9μs	252.7μs

Table 4: T_1: Throughput, average and tail RTT for Swift and Swift-v0 that uses different target delays without decomposing fabric and endpoint congestion.

Flows with same path lengths: First, we show how Swift converges to fair-share as flows arrive and depart. In Figure 20, we start with a single flow between a pair of machines. Keeping the destination machine the same, we incrementally add one more flow from a different source machine and then start tearing down the flows one by one. We see the flow allocations are tight and fair.

While fairness across a few flows is essential, fairness across thousands of flows is imperative at scale. For this, we use 50 machines in T_1, each with 100 flows sending to a single destination machine for a total of 5000 flows competing for the same bottleneck link. We plot the throughput over time, CDF of flow rates, and Jain's fairness index [26] in Figure 21. We randomly sample 50 flows to keep the throughput plot legible, but the CDF and Jain's fairness index is measured across all 5000 flows. Even though the per-flow fair-share rate is only 10Mbps on a 50Gbps link, Swift achieves good fairness with a Jain's fairness index of 0.91. The impact of flow-based scaling is evident, as it tightens the throughput range for an extremely demanding workload.

Flows with different path lengths (RTT Fairness): Swift scales the target delay for a flow based on network path length. This not only reduces latency for shorter paths, but provides fairness irrespective of the base RTT for a flow. To show how well this mechanism works, we use two flows destined to the same machine, one sent from the same rack and the other sent from a remote rack. We plot the throughput of each flow without and then with topology-based scaling. The results in Figure 22 show a marked improvement to fair throughput levels.

6 RELATED WORK

Swift is inspired by a large body of work for datacenter congestion control (CC) summarized in Table 5. We summarize the work below.

ECN-based: DCTCP [1], a trailblazer for datacenter CC and the main comparison point for Swift, uses ECN for rate control. Hull [2] uses phantom queues, and D^{2}TCP [52] is a deadline-aware protocol that also uses ECN. When number of flows exceed network BDP,

Congestion Control Category	Simplicity/Deployability	NIC/Endhost Support	Support in Switches	Robust to Traffic Patterns	Congestion Handled
ECN based: DCTCP, D^2TCP, HULL	Complex ECN Tuning/Deployment	Not Required	ECN, HULL Phantom-Q	Incast Issues	Fabric Only
Explicit Feedback: XCP, RCP, DCQCN, HPCC, D^3	Complex Scheme/Deployment	Required for HPCC, DCQCN	Required for XCP, RCP, D^3, HPCC	Incast Issues (not HPCC)	Fabric Only
Receiver/Credit Based: Homa, NDP, pHost, ExpressPass	Not Universally Deployable	Not Required	Needed for NDP, ExpressPass	Work Well	ToR Downlink not ExpressPass, NDP
Packet Scheduling: pFabric, QJump, PDQ, Karuna, FastPass	Not Deployable As Is	Not Required	Needed for PDQ, pFabric	Incast Issues, Specificity	Fabric Only
Swift	Simple, Wide Deployment at Scale	NIC TimeStamps	None	Works Well	Fabric and Endhost

Table 5: The focal point of Swift is simplicity and ease of deployment while providing excellent end-to-end performance.

ECN-based schemes cannot match sending rate to bottleneck bandwidth. In addition to not handling congestion at hosts, tuning ECN thresholds in heterogeneous networks is prone to bugs.

Explicit Network Feedback including INT: XCP [27], RCP [17], and D^3 [54] rely on switches to feedback rates to end-hosts. DCQCN [59] relies on ECN and combines elements of DCTCP and QCN [48] to control congestion. HPCC [34] relies on in-network telemetry (INT) to obtain precise load information and for rate control. Deployability of these schemes is a challenge, especially in heterogeneous datacenters, as they require coordinated switch, NIC, and end-host support. Swift uses an intuitive delay framework that does not need switch support, though we note that it can easily incorporate INT as it becomes more widely available. In particular, INT can measure per-hop sojourn times to provide a more accurate breakdown of delay.

Credit-based: pHost [21], NDP [23], Homa [39] and ExpressPass [15], rely on the receiver end-host issuing credit packets. While they show tremendous improvement in reducing flow completion time (FCT), Homa and pHost assume that congestion is at ToR downlinks. In practice, congestion can happen in the fabric which can be over-subscribed [46]. Additionally, ExpressPass and NDP require switch modifications. Swift handles end-to-end congestion without requiring new support. Schemes like Homa can be layered atop Swift to issue grants based on *cwnd*.

Congestion Control via Packet Scheduling: pFabric [3] achieves near-optimal FCT through the usage of QoS queues. However, this framework does not support multi-tenant environments in which a large RPC for one tenant may be of higher priority than a small RPC for another tenant. QJUMP [22] requires priorities to be specified manually on a per-application basis. Karuna [14] requires a global calculation. PDQ [24] requires switches to maintain per-flow state. FastPass [42] places scheduling logic in a central scheduler.

Delay-based Schemes: Swift builds upon TIMELY [38] and DX[33], which championed the use of one-way queuing delay as a signal for congestion control. Swift advances over TIMELY [38] include: decoupling fabric and host congestion; using a simple target end-to-end delay instead of RTT gradients; scaling the target based on load and topology; handling extreme incast; and measuring RTT precisely even in the presence of ACK coalescing. In retrospect, we appreciate how aspects of Swift's design have addressed challenges in using delay as a signal that were called out by Zhu et al. [60] (as detailed in Appendix E).

7 CONCLUSION AND FUTURE DIRECTIONS

Congestion control has increasingly adopted complex constructs to generalize to a range of applications and workloads. These efforts often require coordinated changes across switches, hosts, centralized entities and applications, limiting adoption and, paradoxically, generality. In this paper, we report on our multi-year experience with congestion control in production at Google. After some false starts, we realized that simplicity remains a virtue when choosing congestion signals. We settled on delay as the simplest actionable feedback. Very low base latency in the datacenter provides the opportunity to quickly react to both network and end host dynamics. However, doing so requires high fidelity measurement of delay and decomposition of such measures into meaningful constituent components. Both requirements have historically been hard to achieve though it is worth noting that most earlier attempts focused on wide-area deployments. By leveraging NIC hardware timestamps and rapid reaction to congestion in the protocol stack, we show that delay can be both simple to use and extremely effective. Swift achieves ~30us tail latency while maintaining near 100% utilization.

While we feel we are tantalizingly close, we see multiple opportunities to improve on Swift. We believe it is competitive with the best centralized or in-network credit-based/explicit feedback schemes, but this remains to be shown. We also view Swift's algorithm as transport-agnostic, including existing TCP stacks and Cloud virtualization stacks. We believe that delay is useful for controlling higher-level operations such as RPC rate, with the opportunity to perform fine-grained load balancing and timeouts. Finally, while we have substantially improved on predictable latency relative to the state of the art, supporting <10us latency for short transfers will require new techniques as target transfer times approach the actual propagation time in datacenters.

ACKNOWLEDGMENTS

We would like to thank Neal Cardwell, Steven Gribble, Jeff Mogul, the anonymous SIGCOMM reviewers and our shepherd, Yibo Zhu, for providing valuable feedback. Swift is a multi-year effort at Google that benefited from an ecosystem of support and innovation, from RoCE to Pony Express. We thank the Pony Express and Storage production, and support teams at Google for their contributions to the work, including but not limited to, Inho Cho, Yi Cui, Qiaobin Fu, Bill Veraka, Larry Greenfield, Sean Bauer, Michael Marty, Marc de Kruijf, Nicholas Kidd, Milo Martin and Joel Scherplez. Manya Ghobadi, Emily Blem, Vinh The Lam, Philip Wells and Ashish Naik contributed to the work in the early days of Swift.

REFERENCES

[1] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data Center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 Conference (SIGCOMM âĂŹ10)*. Association for Computing Machinery, New York, NY, USA, 63âĂŞ74. https://doi.org/10.1145/1851182.1851192

[2] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. 2012. Less is More: Trading a Little Bandwidth for Ultra-low Latency in the Data Center. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*. USENIX Association, Berkeley, CA, USA, 19–19. http://dl.acm.org/citation.cfm?id=2228298.2228324

[3] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. 2013. pFabric: Minimal Near-optimal Datacenter Transport. In *Proceedings of the ACM SIGCOMM 2013 Conference (SIGCOMM '13)*. ACM, New York, NY, USA, 435–446. https://doi.org/10.1145/2486001.2486031

[4] M. Allman, K. Avrachenkov, U. Ayesta, J. Blanton, and P. Hurtig. 2010. *Early Retransmit for TCP and Stream Control Transmission Protocol (SCTP)*. RFC 5827. RFC Editor. http://www.rfc-editor.org/rfc/rfc5827.txt http://www.rfc-editor.org/rfc/rfc5827.txt.

[5] Guido Appenzeller, Isaac Keslassy, and Nick McKeown. 2004. Sizing Router Buffers. In *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM âĂŹ04)*. Association for Computing Machinery, New York, NY, USA, 281âĂŞ292. https://doi.org/10.1145/1015467.1015499

[6] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. 2020. Enabling Programmable Transport Protocols in High-Speed NICs. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 93–109. https://www.usenix.org/conference/nsdi20/presentation/arashloo

[7] Krste Asanović. 2014. FireBox: A Hardware Building Block for 2020 Warehouse-Scale Computers. In *12th USENIX Conference on File and Storage Technologies*. USENIX Association, Santa Clara, CA.

[8] Wei Bai, Kai Chen, Li Chen, Changhoon Kim, and Haitao Wu. 2016. Enabling ECN over Generic Packet Scheduling. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies (CoNEXT âĂŹ16)*. Association for Computing Machinery, New York, NY, USA, 191âĂŞ204. https://doi.org/10.1145/2999572.2999575

[9] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. 2017. Attack of the Killer Microseconds. *Commun. ACM* 60, 4 (March 2017), 48âĂŞ54. https://doi.org/10.1145/3015146

[10] E. Blanton and M. Allman. 2004. *Using TCP Duplicate Selective Acknowledgement (DSACKs) and Stream Control Transmission Protocol (SCTP) Duplicate Transmission Sequence Numbers (TSNs) to Detect Spurious Retransmissions*. RFC 3708. RFC Editor.

[11] Google Cloud Blog. 2018. How Distributed Shuffle improves scalability and performance in Cloud Dataflow pipelines. (2018). https://cloud.google.com/blog/products/data-analytics/how-distributed-shuffle-improves-scalability-and-performance-cloud-dataflow-pipelines

[12] Lawrence S. Brakmo, Sean W. OâĂŹMalley, and Larry L. Peterson. 1994. TCP Vegas: New Techniques for Congestion Detection and Avoidance. *SIGCOMM Comput. Commun. Rev.* 24, 4 (Oct. 1994), 24âĂŞ35. https://doi.org/10.1145/190809.190317

[13] Chelsio Communications. 2020. Chelsio TCP Offload Engine. https://www.chelsio.com/nic/tcp-offload-engine/. (2020). Accessed: 2020-02-02.

[14] Li Chen, Kai Chen, Wei Bai, and Mohammad Alizadeh. 2016. Scheduling Mix-flows in Commodity Datacenters with Karuna. In *Proceedings of the ACM SIGCOMM 2016 Conference (SIGCOMM '16)*. ACM, New York, NY, USA, 174–187. https://doi.org/10.1145/2934872.2934888

[15] Inho Cho, Keon Jang, and Dongsu Han. 2017. Credit-Scheduled Delay-Bounded Congestion Control for Datacenters. In *Proceedings of the ACM SIGCOMM 2017 Conference (SIGCOMM '17)*. ACM, New York, NY, USA, 239–252.

[16] Jeffrey Dean and Luiz André Barroso. 2013. The Tail at Scale. *Commun. ACM* 56, 2 (Feb. 2013), 74âĂŞ80. https://doi.org/10.1145/2408776.2408794

[17] Nandita Dukkipati and Nick McKeown. 2006. Why Flow-Completion Time is the Right Metric for Congestion Control. *SIGCOMM Comput. Commun. Rev.* 36, 1 (Jan. 2006), 59âĂŞ62. https://doi.org/10.1145/1111322.1111336

[18] Paolo Faraboschi, Kimberly Keeton, Tim Marsland, and Dejan Milojicic. 2015. Beyond Processor-centric Operating Systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. USENIX Association, Kartause Ittingen, Switzerland, 1–7. https://www.usenix.org/conference/hotos15/workshop-program/presentation/faraboschi

[19] S. Floyd, J. Mahdavi, M. Mathis, and M. Podolsky. 2000. *An Extension to the Selective Acknowledgement (SACK) Option for TCP*. RFC 2883. RFC Editor.

[20] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Network Requirements for Resource Disaggregation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 249–264. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gao

[21] Peter X. Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2015. pHost: Distributed Near-optimal Datacenter Transport over Commodity Network Fabric. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT '15)*. ACM, New York, NY, USA, Article 1, 12 pages. https://doi.org/10.1145/2716281.2836086

[22] Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert N. M. Watson, Andrew W. Moore, Steven Hand, and Jon Crowcroft. 2015. Queues Don't Matter When You Can JUMP Them!. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 1–14. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/grosvenor

[23] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichik, and Marcin Mojcik. 2017. Re-architecting Datacenter Networks and Stacks for Low Latency and High Performance. In *Proceedings of the ACM SIGCOMM 2017 Conference (SIGCOMM '17)*. ACM, New York, NY, USA, 29–42.

[24] Chi-Yao Hong, Matthew Caesar, and P. Brighten Godfrey. 2012. Finishing Flows Quickly with Preemptive Scheduling. In *Proceedings of the ACM SIGCOMM 2012 Conference (SIGCOMM '12)*. ACM, New York, NY, USA, 127–138. https://doi.org/10.1145/2342356.2342389

[25] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *CoRR* abs/1903.05714 (2019), 1–61. arXiv:1903.05714 http://arxiv.org/abs/1903.05714

[26] Raj Jain, Dah Ming Chiu, and Hawe WR. 1984. A Quantitative Measure Of Fairness And Discrimination For Resource Allocation In Shared Computer Systems. (09 1984), 37 pages.

[27] Dina Katabi, Mark Handley, and Charlie Rohrs. 2002. Congestion Control for High Bandwidth-Delay Product Networks. In *Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM âĂŹ02)*. Association for Computing Machinery, New York, NY, USA, 89âĂŞ102. https://doi.org/10.1145/633025.633035

[28] K. Katrinis, D. Syrivelis, D. Pnevmatikatos, G. Zervas, D. Theodoropoulos, I. Koutsopoulos, K. Hasharoni, D. Raho, C. Pinto, F. Espina, S. Lopez-Buedo, Q. Chen, M. Nemirovsky, D. Roca, H. Klos, and T. Berends. 2016. Rack-scale disaggregated cloud data centers: The dReDBox project vision. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*. IEEE, Dresden, Germany, 690–695.

[29] Changhoon Kim, Parag Bhide, Ed Doe, Hugh Holbrook, Anoop Ghanwani, Dan Daly, Mukesh Hira, and Bruce Davie. 2016. InâĂŘband Network Telemetry (INT). https://p4.org/assets/INT-current-spec.pdf. (2016). Accessed: 2020-01-13.

[30] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. 2016. Flash Storage Disaggregation. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys âĂŹ16)*. Association for Computing Machinery, New York, NY, USA, Article Article 29, 15 pages. https://doi.org/10.1145/2901318.2901337

[31] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. 2017. ReFlex: Remote Flash Local Flash. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS âĂŹ17)*. Association for Computing Machinery, New York, NY, USA, 345âĂŞ359. https://doi.org/10.1145/3037697.3037732

[32] Gautam Kumar, Srikanth Kandula, Peter Bodik, and Ishai Menache. 2013. Virtualizing Traffic Shapers for Practical Resource Allocation. In *Presented as part of the 5th USENIX Workshop on Hot Topics in Cloud Computing*. USENIX, San Jose, CA, 1–6. https://www.usenix.org/conference/hotcloud13/workshop-program/presentations/Kumar

[33] C. Lee, C. Park, K. Jang, S. Moon, and D. Han. 2017. DX: Latency-Based Congestion Control for Datacenters. *IEEE/ACM Transactions on Networking* 25, 1 (Feb 2017), 335–348. https://doi.org/10.1109/TNET.2016.2587286

[34] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and et al. 2019. HPCC: High Precision Congestion Control. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM âĂŹ19)*. Association for Computing Machinery, New York, NY, USA, 44âĂŞ58. https://doi.org/10.1145/3341302.3342085

[35] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. 2017. Octopus: an RDMA-enabled Distributed Persistent Memory File System. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 773–785. https://www.usenix.org/conference/atc17/technical-sessions/presentation/lu

[36] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, and et al. 2019. Snap: A Microkernel Approach to Host Networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP âĂŹ19)*. Association for Computing Machinery, New York, NY, USA, 399âĂŞ413. https://doi.org/10.1145/3341301.3359657

[37] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. 1996. *TCP Selective Acknowledgment Options*. RFC 2018. RFC Editor.

[38] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. 2015. TIMELY: RTT-based Congestion Control for the Datacenter. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*

(SIGCOMM '15). ACM, New York, NY, USA, 537–550. https://doi.org/10.1145/2785956.2787510

[39] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. 2018. Homa: A Receiver-driven Low-latency Transport Protocol Using Network Priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*. ACM, New York, NY, USA, 221–235. https://doi.org/10.1145/3230543.3230564

[40] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. 2015. Latency-Tolerant Software Distributed Shared Memory. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. USENIX Association, Santa Clara, CA, 291–305. https://www.usenix.org/conference/atc15/technical-session/presentation/nelson

[41] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, et al. 2015. The RAMCloud Storage System. *ACM Transactions on Computer Systems (TOCS)* 33, 3 (2015), 7.

[42] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. 2014. Fastpass: A Centralized "Zero-queue" Datacenter Network. In *Proceedings of the ACM SIGCOMM 2014 Conference (SIGCOMM '14)*. ACM, New York, NY, USA, 307–318. https://doi.org/10.1145/2619239.2626309

[43] Ahmed Saeed, Nandita Dukkipati, Vytautas Valancius, Vinh The Lam, Carlo Contavalli, and Amin Vahdat. 2017. Carousel: Scalable Traffic Shaping at End Hosts. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM âĂŹ17)*. Association for Computing Machinery, New York, NY, USA, 404âĂŞ417. https://doi.org/10.1145/3098822.3098852

[44] Ahmed Saeed, Yimeng Zhao, Nandita Dukkipati, Ellen Zegura, Mostafa Ammar, Khaled Harras, and Amin Vahdat. 2019. Eiffel: Efficient and Flexible Software Packet Scheduling. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 17–32. https://www.usenix.org/conference/nsdi19/presentation/saeed

[45] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2018. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 69–87. https://www.usenix.org/conference/osdi18/presentation/shan

[46] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, and et al. 2015. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in GoogleâĂŹs Datacenter Network. *SIGCOMM Comput. Commun. Rev.* 45, 4 (Aug. 2015), 183âĂŞ197. https://doi.org/10.1145/2829988.2787508

[47] 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. 2020. 1RMA: Re-envisioning Remote Memory Access for Multi-tenant Datacenters. In *Proceedings of the 2020 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '20)*. ACM, New York, NY, USA, to appear.

[48] IEEE Std. 2010. IEEE 802.11Qau. Congestion notification. (2010).

[49] IEEE Std. 2011. IEEE. 802.11Qbb. Priority based flow control. (2011).

[50] Mohit P. Tahiliani, Vishal Misra, and K. K. Ramakrishnan. 2019. A Principled Look at the Utility of Feedback in Congestion Control. In *Proceedings of the 2019 Workshop on Buffer Sizing (BS âĂŹ19)*. Association for Computing Machinery, New York, NY, USA, Article Article 8, 5 pages. https://doi.org/10.1145/3375235.3375243

[51] Jordan Tigani and Siddartha Naidu. 2014. *Google BigQuery Analytics*. Wiley, Indianapolis, IN, USA.

[52] Balajee Vamanan, Jahangir Hasan, and T.N. Vijaykumar. 2012. Deadline-aware Datacenter TCP (D2TCP). In *Proceedings of the ACM SIGCOMM 2012 Conference (SIGCOMM '12)*. ACM, New York, NY, USA, 115–126. https://doi.org/10.1145/2342356.2342388

[53] Washington State Department of Transportation. 2020. What is a roundabout? https://www.wsdot.wa.gov/Safety/roundabouts/BasicFacts.htm. (2020). Accessed: 2020-01-13.

[54] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. 2011. Better Never Than Late: Meeting Deadlines in Datacenter Networks. In *Proceedings of the ACM SIGCOMM 2011 Conference (SIGCOMM '11)*. ACM, New York, NY, USA, 50–61. https://doi.org/10.1145/2018436.2018443

[55] Jian Xu and Steven Swanson. 2016. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. USENIX Association, Santa Clara, CA, 323–338. https://www.usenix.org/conference/fast16/technical-sessions/presentation/xu

[56] Jian Yang, Joseph Izraelevitz, and Steven Swanson. 2019. Orion: A Distributed File System for Non-Volatile Main Memory and RDMA-Capable Networks. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. USENIX Association, Boston, MA, 221–234. https://www.usenix.org/conference/fast19/presentation/yang

[57] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. USENIX, San Jose, CA, 15–28. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia

[58] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloudâĂŹ10)*. USENIX Association, USA, 10.

[59] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. 2015. Congestion Control for Large-Scale RDMA Deployments. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '15)*. ACM, New York, NY, USA, 523–536. https://doi.org/10.1145/2785956.2787484

[60] Yibo Zhu, Monia Ghobadi, Vishal Misra, and Jitendra Padhye. 2016. ECN or Delay: Lessons Learnt from Analysis of DCQCN and TIMELY. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies (CoNEXT âĂŹ16)*. Association for Computing Machinery, New York, NY, USA, 313âĂŞ327. https://doi.org/10.1145/2999572.2999593

Appendices are supporting material that have not been peer-reviewed.

A CONVERSION BETWEEN HOST AND NIC CLOCKS

Some delay computations, e.g., NIC-Rx-queuing delay, require a combination of NIC HW and host SW clocks. Swift uses a simple linear-extrapolation approach to convert the incoming HW clock into a host-clock value to compute such delays:

$$host\_clock = ratio \cdot nic\_clock + offset$$

We update the $ratio$ and $offset$ periodically. The algorithm is simple: we read the nic_clock, then the $host\_clock$ and also maintain the previous set of readings as $last\_nic\_clock$ and $last\_host\_clock$. The $ratio$ and $offset$ can then be updated as:

$$ratio = \frac{host\_clock - last\_host\_clock}{nic\_clock - last\_nic\_clock}$$
$$offset = host\_clock - ratio \cdot nic\_clock$$

B PACKET FORMAT

Figure 23 shows the format that Swift uses, which consumes 4 bytes to reflect back remote-side queuing delay. In addition, 1 byte is used for to echo back forward-side hop-count by computing the difference between the initial TTL and observed TTL on the incoming packet.

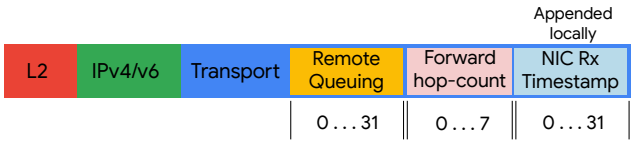

Figure 23: Packet format changes for Swift.

C DELAYS WITH SHARED RX/TX SCHEDULING

Traditionally, links are bidirectional, i.e., incoming traffic doesn't affect outgoing traffic on a link. Our deployment of Swift in Snap had a unique challenge since Rx and Tx scheduling is shared in Snap. While counter-intuitive, Swift design addresses this by factoring local NIC queue buildup as part of endpoint congestion. The insight is taken from traffic-roundabouts which also have shared Rx and Tx scheduling; factoring local NIC-Rx delay emulates yield-at-entry as the right-of-way when looked at it as a roundabout [53]. In other words, if a machine's NIC-Rx queue builds up, it should prioritize clearing those packets before trying to inject more packets (through Tx) in the network.

D EXPERIMENT WITH TARGET DELAY

In our experience, a nice property of delay is that as low latency networking stacks advance to avoid interference from host CPUs, Swift continues to work well just with a simple knob of target delay. In Figure 24, we show results from a testbed implementation of Swift in a prototype stack where it is able to achieve near line-rate throughput (~100Gbps) even at 15μs RTT.

E DELAY OR ECN

Zhu et al. [60] called out challenges in using delay as a congestion-signal; we appreciate how some aspects of Swift's design ended up addressing these challenges.

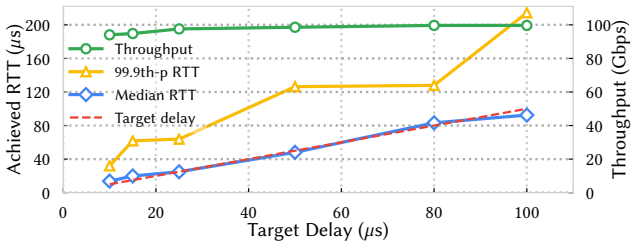

Figure 24: Achieved RTT vs. target delay, 160-flow incast.

- The authors show that a fluid model of TIMELY does not have a unique fixed point because of reliance on the *gradient-based control*. Swift rather uses a target delay (vs. gradient-based control) and in this way, doesn't suffer from multiple fixed-points at convergence. The authors echo our experience and provide a version called, Patched TIMELY, that also gets rid of the delay-gradient.
- The authors rightly note that delay lags behind ECN in that modern switches mark ECN at packet-egress while delay, implicitly, measures congestion when the packet arrives at the bottleneck switch. As discussed in §3.2 and shown in experiments §5, Swift uses a low target delay and does not delay ACKs, and hence largely mitigates this concern. Testimony to this is our experience with Swift achieving low latency and losses at scale at Google.
- The authors provide an important result (Theorem 6 in Reference [60]): purely relying on end-to-end delay measurements for congestion control can provide *either fairness or fixed-delay but not both*. The reason is that if the delay is controlled to a fixed value, the algorithm is agnostic of the number of flows making the system of equations inconsistent. Swift resolves this by not using the same target-delay for all flows and instead scales target delay as explained in §3.5. For example, flow based scaling varies target delay as function of congestion window and converges to a single fixed point. As shown in Figure 21, this greatly improves fairness especially under large scale incast.
- The authors in Reference [50] argue that end-to-end delay is an ambiguous signal in that a flow may traverse a wide variety of link speeds across a number of hops. We believe while this can be a valid concern for the Internet traffic, this concern is much less applicable to a datacenter CC like Swift where paths are known and the link speeds don't vary as widely as in the Internet. Swift uses topology-based scaling to account for different hop-counts across flows.
- In addition, delay has a few important operational advantages. First, delay evolves naturally as networks become faster, and tuning it at scale is simpler than ECN especially for production environments with multiple QoS classes—it has been shown that ECN is problematic in such scenario and sojourn time is more robust [8]. Second, delay has a multi-bit nature as a congestion signal; it provides visibility into the extent of congestion, unlike ECN which only signals whether congestion exists or not. That said, we look forward to integrating multi-bit ECN signals like sojourn-time [8] and INT [29] in Swift's framework.