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

Present by Kaiyue Li

Background

- The need for low-latency operations continues to grow
  - To make use of the advantage of next-gen storage, we need low latency messaging
- DCTCP no longer works due to its high tail latency at scale
  - We need micro-second level of tail latency

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

Issues of DCTCP

- Need to be deployed on each switch: hard to implement
- K (threshold) and g (gain) need to be fine-grained to perform well.
- Do not address the congestion build on on the host end for incast problem
- Network stack in kernel itself is a burden to latency

Swift

- Latency-based congestion control
- Use SNAP to avoid the kernel network stack
- Utilize NIC timestamp to collect latency
- Provide pacing functionality for cases when CWND < 1

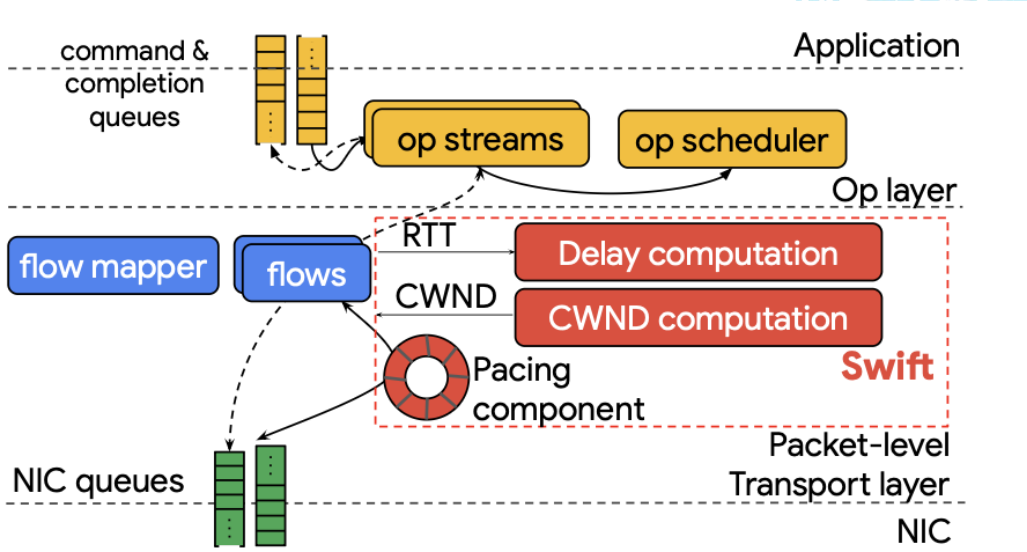


Figure 1: Swift as a packet-level congestion-control in the context of the Pony-Express architecture.

Design

- Requirements
  - Low latency with near zero loss and high throughput
  - End to end congestion control for both fabric (between hosts) and endpoint congestion
  - CPU efficient

Separate delays

To separate fabric and endpoint congestion, we need to separate the delays of a single RTT

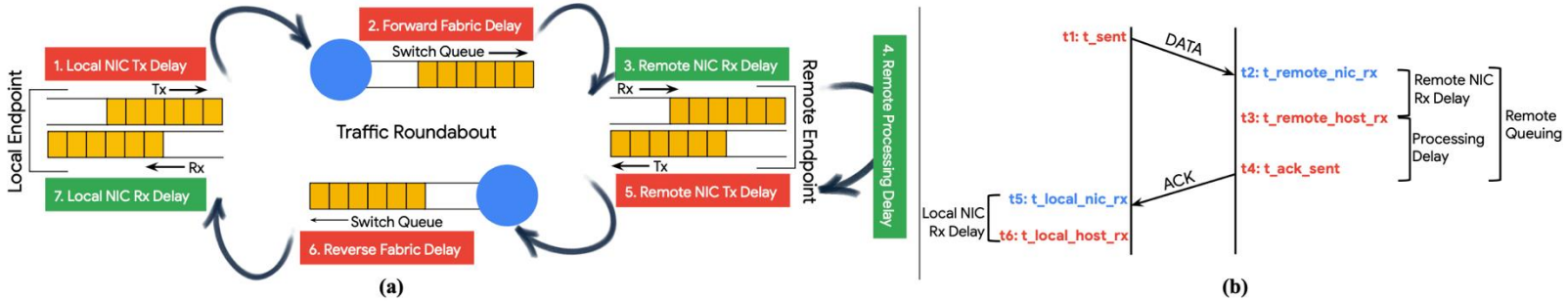
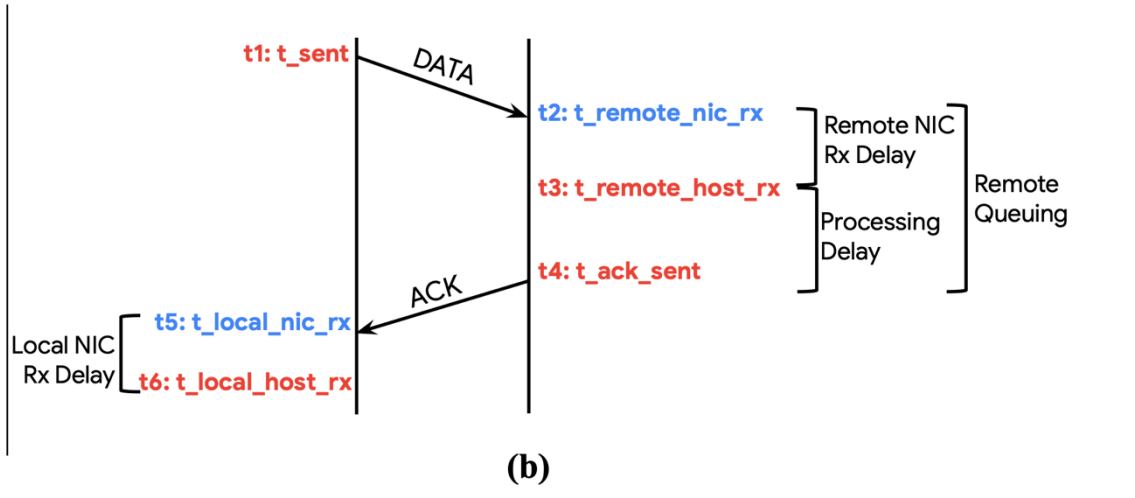


Figure 2: (a) Components of end-to-end RTT for a data packet and corresponding ACK packet. (b) Timestamps used to measure different delays (Hardware and software timestamps are shown in blue and red, respectively).

Swift delay

- Endpoint delay
  - Remote-queuing (t4- t2) + Local NIC RX delay (t6- t5)
- Fabric delay
  - RTT – endpoint delay = (t2 – t1)



Swift: algorithm

- AIMD style + react for every ACK
- cwnd increment at most a_i for entire RTT
- cwnd decrease at most by half
- Beta marks the reaction strength to delay

```
4  On Receiving ACK
5    retransmit_cnt ← 0
6    target_delay ← TargetDelay()                    ▷ See S3.5
7    if delay < target_delay then                    ▷ Additive Increase (AI)
8      if cwnd ≥ 1 then
9        cwnd ← cwnd + ai/cwnd · num_acked
10     else
11       cwnd ← cwnd + ai · num_acked
12   else                                            ▷ Multiplicative Decrease (MD)
13     if can_decrease then
14       cwnd ← max(1 − β · ((delay−target_delay)/delay),
           1 − max_mdf) · cwnd
```

Swift: Two window design

- fcwnd: tracks fabric congestion
- ecwnd: tracks end point congestion
- Effective window min (fcwnd, ecwnd)
- Tail latency improves by 2X

4 **On Receiving ACK**
5 $retransmit\_cnt \leftarrow 0$
6 $target\_delay \leftarrow$ TargetDelay() ▷ See S3.5
7 **if** $delay < target\_delay$ **then** ▷ Additive Increase (AI)
8 **if** $cwnd \geq 1$ **then**
9 $cwnd \leftarrow cwnd + \frac{ai}{cwnd} \cdot num\_acked$
10 **else**
11 $cwnd \leftarrow cwnd + ai \cdot num\_acked$
12 **else** ▷ Multiplicative Decrease (MD)
13 **if** $can\_decrease$ **then**
14 $cwnd \leftarrow \max(1 - \beta \cdot (\frac{delay - target\_delay}{delay}),$
$1 - max\_mdf) \cdot cwnd$

Swift: fine-grained pacing

- Use a timing wheel to implement a timing based sending where cwnd < 1 (more flows then BDP : maximum packet in flight)
- Cwnd = 0.5 -> 1 packet every 2 RTT using pacing_delay as the period of time waiting before sending a packet

$$\textbf{if } cwnd < 1 \textbf{ then}$$
$$\quad pacing\_delay \leftarrow \frac{rtt}{cwnd}$$

Calculate fabric target delay

- The target delay increases with the number of hops and the number of flows
- Number of hops could be known through TTL, (TTL – 1for every hop)
- Number of flows is not known but cwnd is inversely proportional to number of flows N, so we use $\frac{1}{\sqrt{cwnd}}$ to indicate N

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

where

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

Ensure coexistence between protocols

- Swift utilizes QoS to coexist with other protocols in data center scenario
- Assign dedicated QoS queues for swift protocol

Production result

- Compare Swift with GCN, Google's own version of DCTCP that reacts faster to congestion.
- Swift achieves low loss even at line-rate

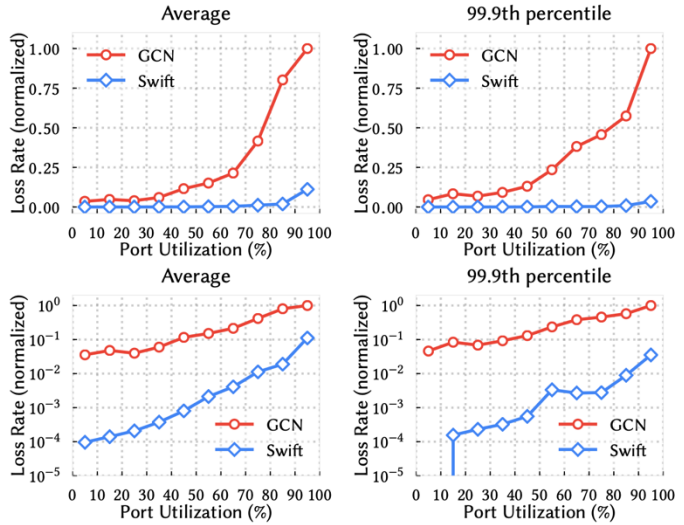


Figure 6: Edge (ToR to host) links: Average and 99.9p Swift/GCN loss rate (linear and log scale) vs. combined utilization, bucketed at 10% intervals. Loss rate is normalized to highest GCN loss rate. The near-vertical line in the log-scale plot is due to extremely small relative loss-rate.

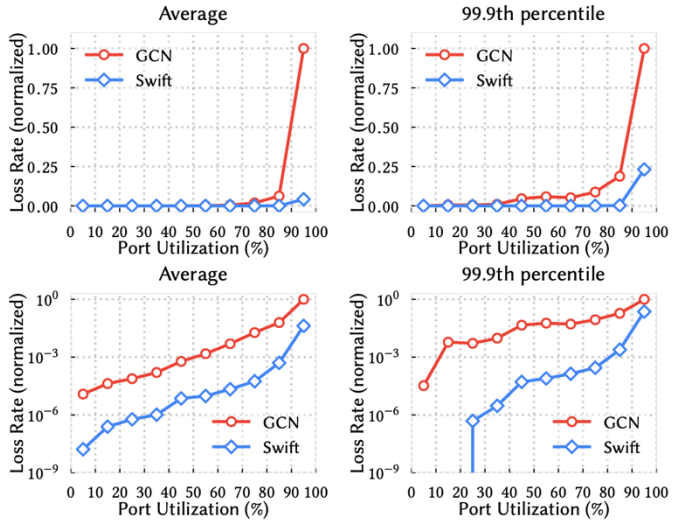


Figure 7: Fabric links: Average and 99.9p Swift/GCN loss rate Swift/GCN loss rate (linear and log scale) vs. combined utilization, bucketed at 10% intervals.

Production result

- Swift achieves low latency near the target
- Thus, the design requirements has been fulfilled

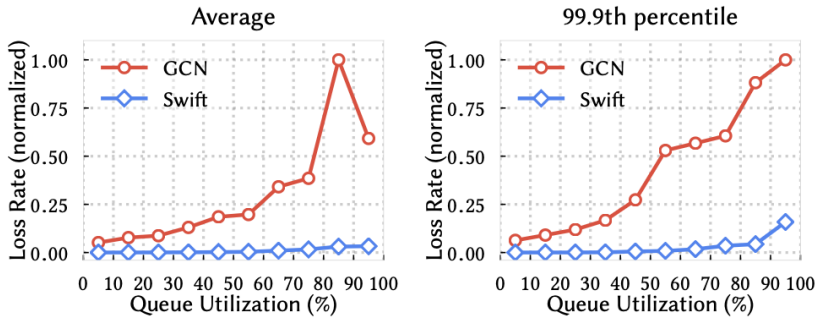


Figure 8: Average and 99.9th percentile loss rate vs. queue utilization.

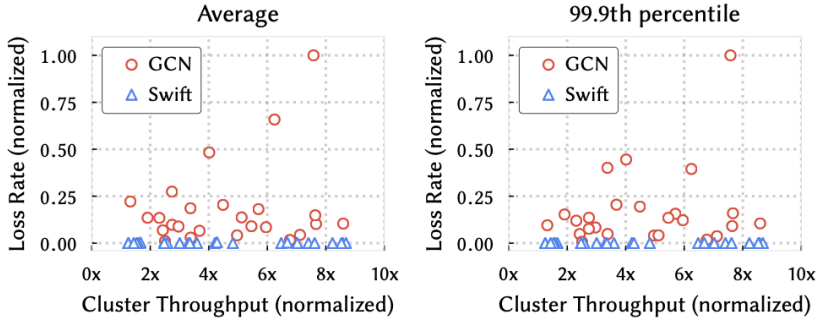


Figure 9: Edge (ToR to host) average and 99.9p loss rate vs. total Swift/GCN throughput in the cluster.

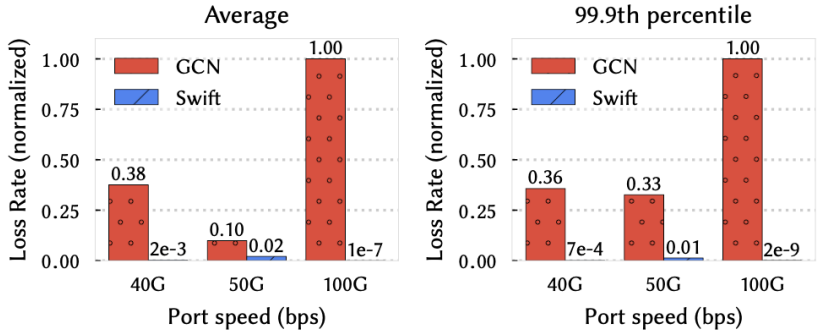


Figure 10: Average and 99.9p loss rate of highly-utilized (>90%) links in each switch group.

Production result

- Swift achieves low loss rate in QoS when coexists with other protocols even with lower priority queues

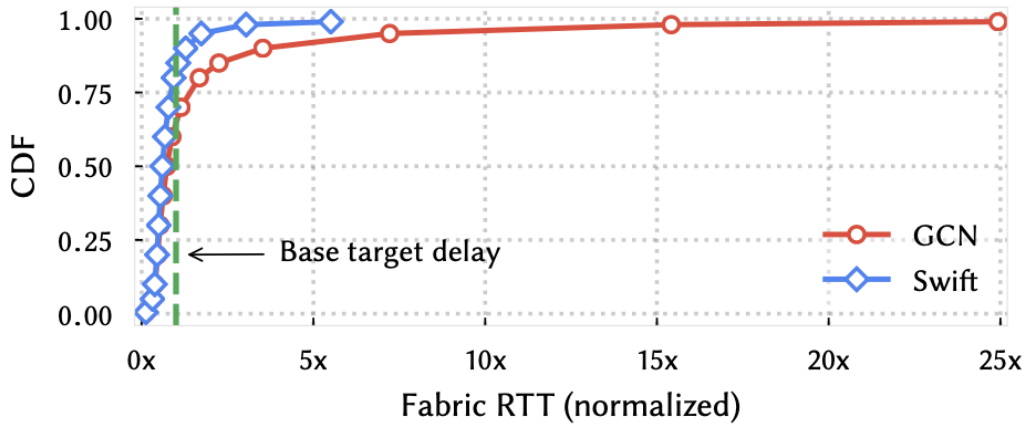


Figure 11: Fabric RTT: Swift controls fabric delay more tightly than GCN.

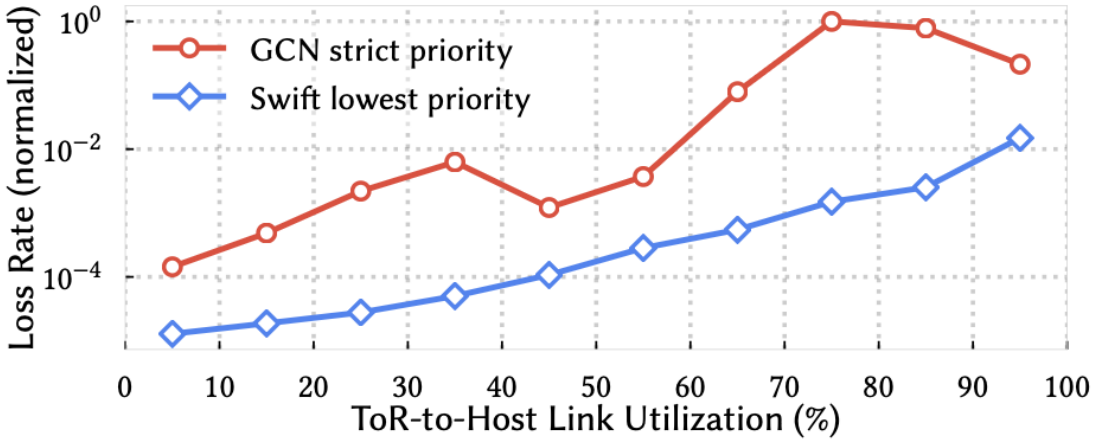


Figure 13: Average loss rate vs. port utilization for GCN traffic at strict scheduling priority and Swift at lower scheduling weight for ToR-to-host links. The highest GCN loss rate is normalized to 1.0.

Production result

- The separation of congestion is key its success and a great source of debugging

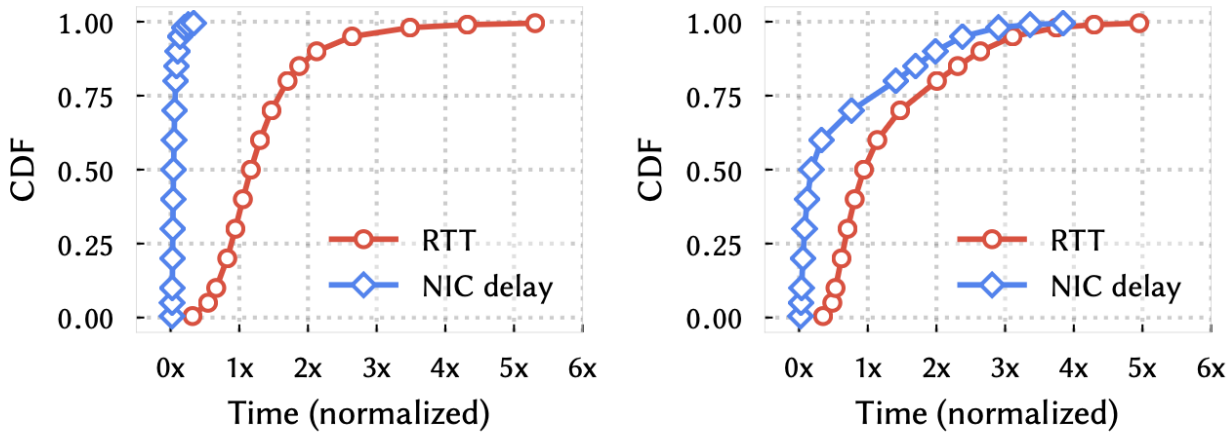


Figure 14: CDF of end-to-end packet RTT and NIC-Rx-queuing delay for the throughput-intensive cluster (left) and IOPS-intensive cluster (right).

Evaluate mechanism in swift: Effect of Target Delay

- Disabled dynamic target delay
- Only base target delay is used
- RTT closely tracked the base target delay
- Throughput saturated at 25 microseconds (easy to find a recommended base)

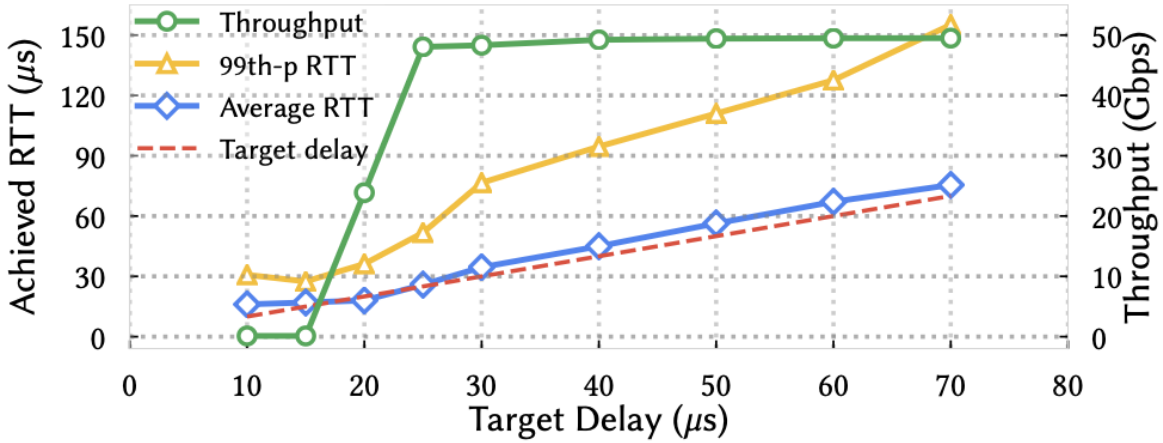
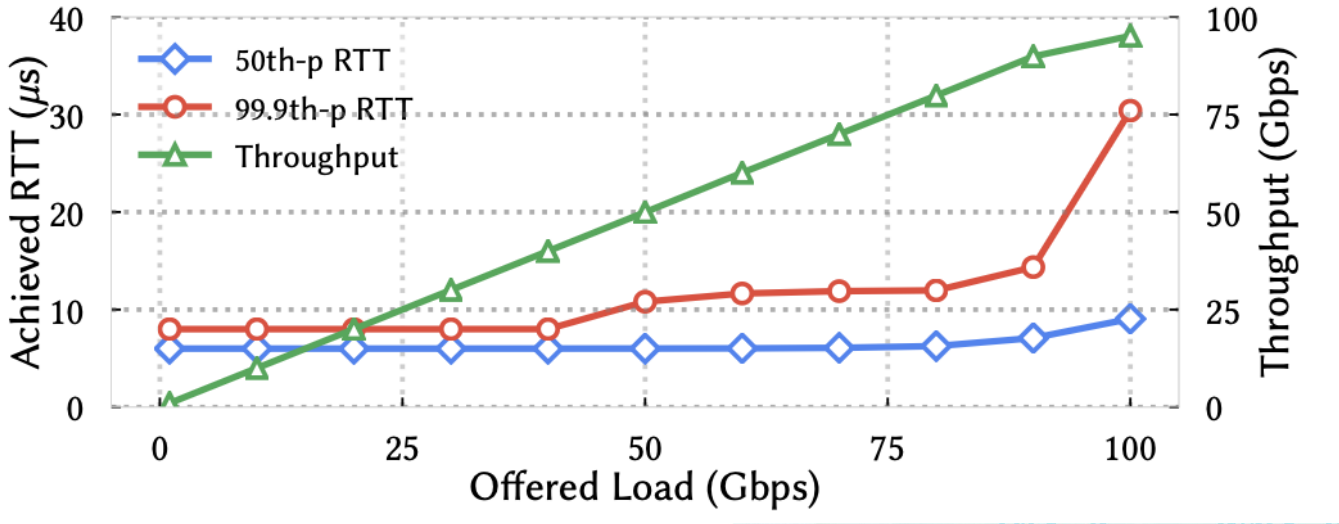


Figure 17: T_1: Achieved RTT and throughput vs. target delay, 100-flow incast.

Evaluate mechanism in swift: Low tail latency

- Only after 85% of offered load when the 99.9th latency begins to spike, but also within at most 3X higher than the unloaded RTT



Evaluate mechanism in swift: Effect of cwnd < 1 support

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.

Evaluate mechanism in swift: Congestion separation

- Swift-v0: a modified version that only uses one single target latency
- Performance downgrade when the separation is removed.

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.

Evaluate mechanism in swift: scaling and fairness

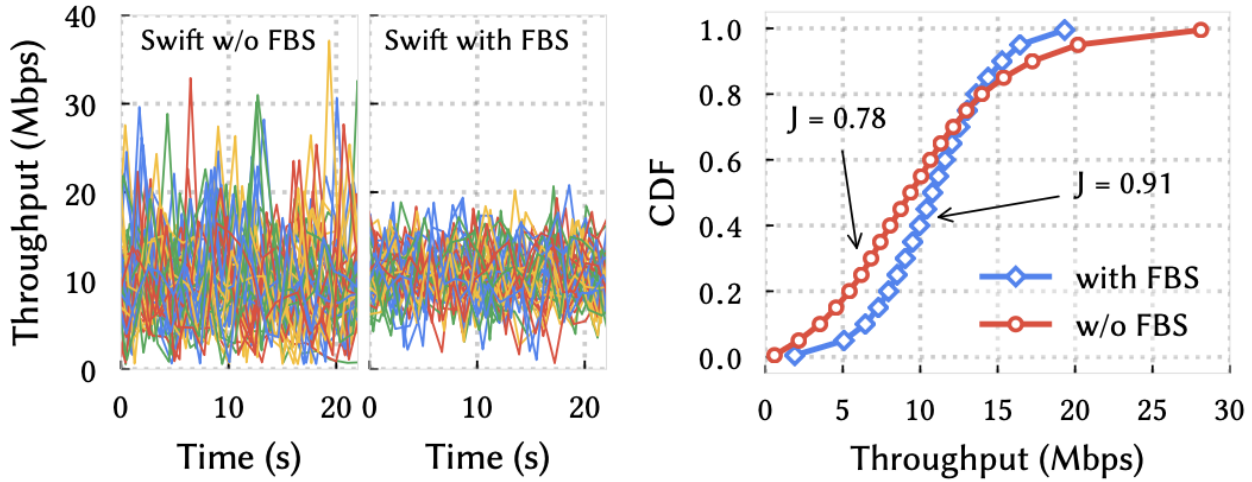


Figure 21: T_1: Throughput with and without flow-based scaling (FBS) for a 5000-to-1 incast. Jain's fairness index (J) shown is measured amongst all 5000 flows using a snapshot of flow rates.

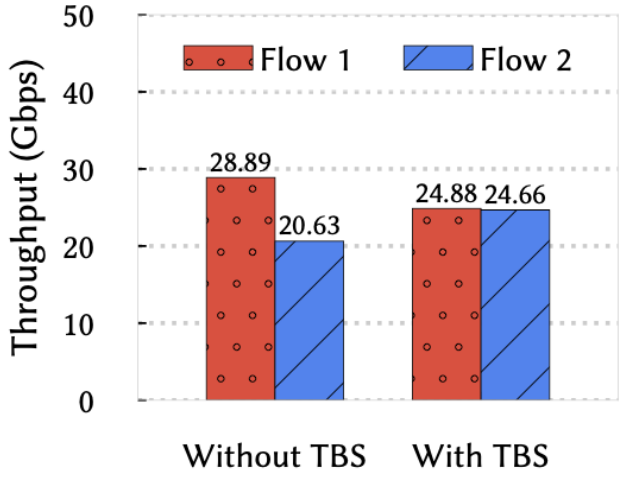


Figure 22: T_1: Throughput of two flows with different path lengths, with and without topology-base scaling (TBS).

Review of related work

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.

Conclusion

- Swift work well in data centers and provide tail latency of around 20 microseconds.
- Its success mainly rooted from its support for cwnd < 1, its separation of fabric and end-host latency, and its calculation of target latency that scales with fabric distance and flow numbers.
- To do better than 20 microseconds, we need new technologies.

Ask me questions orz

Thanks for listening