Paper-Reading Report

Paper:

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

Reporter: Senj Lee

Date: 21st / June / 2024

Introduction

Today's Demand

实践中，由于存储技术的进步，如 Flash、NVMe、PCIe 等下一代存储对延迟的要求愈发严格，因为等待 I/O 时服务器的闲置会导致极大的浪费。

Other works' weakness

- ECN-based Protocol：
  - DCTCP: 在大规模数据中心中对尾部延迟的处理不佳
  - PFC/DCQCN：在 incast 较大时/IOPS 密集型场景中的作用有限
  - 这些基于交换机的 ECN 标记的协议对交换机的依赖性比较强
- RPC/Flow/DDL-based Protocols：
  - 在交换机与主机间协调时，实现、部署、维护等操作过于复杂
  - 不适合多用户环境

Advantages of Swift

- 使用延迟作为拥塞信号，因而部署简单，能够适应快速变化的数据中心的技术
- 像 TIMELY 一样高效利用 CPU 和 NIC 资源，保证对 CPU 的低占用率
- 有效处理 large-scale incast 等流量模式，即使在 $\mathcal{O}(10k)$ 规模的流量情况下也能保持高 IOPS
- 将延迟分解为主机延迟（host delay）和结构延迟（fabric delay），分别对不同的拥塞原因进行测量、响应、优化
- 在不同工作负载的集群中都能维持较低的排队延迟水平，提供较高的利用率，并且保证接近零的丢包率

Motivation

Storage Workloads

存储是我们数据中心网络的主要工作负载。磁盘流量主要由 $\mathcal{O}(10)$ ms 访问延迟而非网络延迟决定，因此承载磁盘流量不需要低延迟拥塞控制。但是，随着集群范围的存储系统向更快的介质发展，延迟变得至关重要：

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

任何单个存储操作的总体延迟都由最长网络操作的延迟决定，因此保持低延迟（尤其是低尾部延迟）至关重要。

Host Networking Stacks

数据中心中的网络栈与常规 Linux 等操作系统的网络栈大不相同，RDMA、NVMe 为了避免 OS/CPU 开销，通常在 OS bypass stacks 如 Snap、NIC 中实现，而 Swift 正是在 Snap 中运行——Snap 提供了 NIC 时间戳和细粒度调速等功能。

随着线路速率和 IOPS 密集型工作负载的发展，给数据包处理的软硬件带来巨大压力，因此对不同软硬件带来的延迟需要进行仔细地考量，从而优化之。

Datacenter Switches

数据中心的交换机设备大相径庭，产生的异构性不可避免，因此与交换机深度绑定的协议（DCTCP-like Protocols）对维护来说是极大的负荷。

而 Swift 使用延迟作为拥塞控制信号，在主机上改进 delay targets 比从交换机信号（ECN-echo）中学习更加方便。

Design & Implementation

Component Delays of RTT

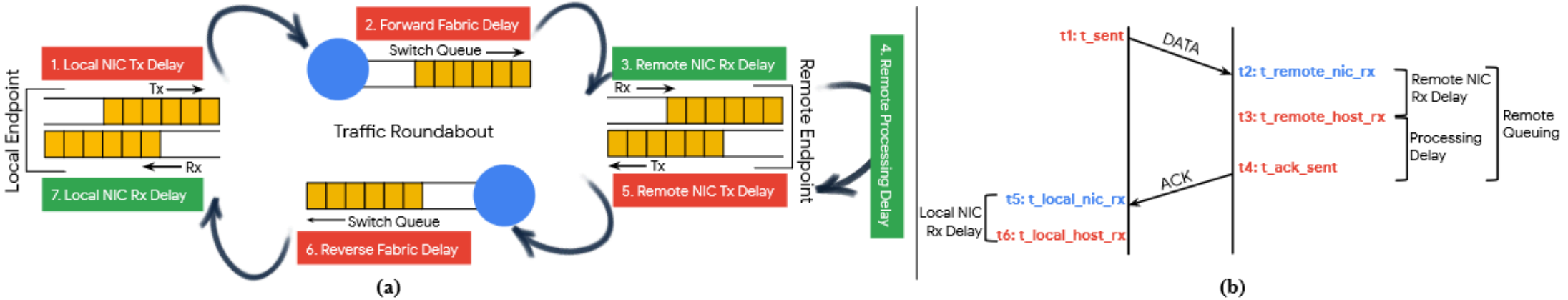

Figure 2: (a) Components of end-to-end RTT for a data packet and corresponding ACK packet. (b) Timestamps used to measure different delays (Hardware and software timestamps are shown in blue and red, respectively).

- **Local NIC Tx Delay**: packet spends in the NIC Tx queue before it is emitted on the wire.
- **Forward Fabric Delay**⭐: the sum of the serialization, propagation and queuing delays.
- **Remote Processing Delay**
- **Reverse Fabric Delay**: the time taken by the ACK packet on the reverse path.

Target Delay Window Control (I)

Advanced than TIMELY

> 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**.

To mitigate staleness concerns in using delay as a congestion signal: (对过时性之担忧)

- use **instantaneous delay** as opposed to minimum or low-pass filtered delay.
- do not explicitly delay ACKs.

Target Delay Window Control (II)

On Receiving ACK

```
retransmit_cnt = 0
target_delay = TargetDelay()
if delay < target_delay then # Additive Increase (AI)
    if cwnd >= 1 then
        cwnd = cwnd + ai / cwnd*num_acked
    else
        cwnd = cwnd + ai * num_acked
else  # Multiplicative Decrease (MD)
    if can_decrease then
        cwnd = max(1 – β*((delay–target_delay)/delay), 1 – max_mdf) * cwnd
```

- AI：so that the cumulative increase over an RTT is equal to *ai* .
- MD: the decrease depending on how far the delay is from the target.
  - MD is constrained to be one per RTT, so that Swift does not react to the same congestion event multiple times.

Fabric vs. Endpoint Congestion

Swift 中将 RTT 进行拆分：链路和交换机造成的结构延迟（fabric delay），以及 NIC 和主机网络堆栈中发生的终端主机延迟（end-point host delay）。

这就是为什么前文说 Swift 使用 RTT 的差来进行拥塞控制：

- $\text{end\_point\_delay} = \text{remote\_queuing} + \text{local\_NIC\_Rx\_delay}$；
- $\text{fabric\_delay} = RTT - \text{end\_point\_delay}$.

Swift 使用两个拥塞窗口，fcwnd 跟踪结构拥塞，ecwnd 跟踪端点拥塞。两个窗口都遵循前文中收到 ACK 的响应的算法，具有不同的结构延迟目标和端点延迟目标。因此有效的拥塞窗口选择 $\min(fcwnd, ecwnd)$ 。如此分离两种不同延迟的设计，使得 Swift 在绝大多数的应用场景中尾部延迟优化了 2 倍之多。

Scaling the Fabric Target Delay to Letency of paths (I)

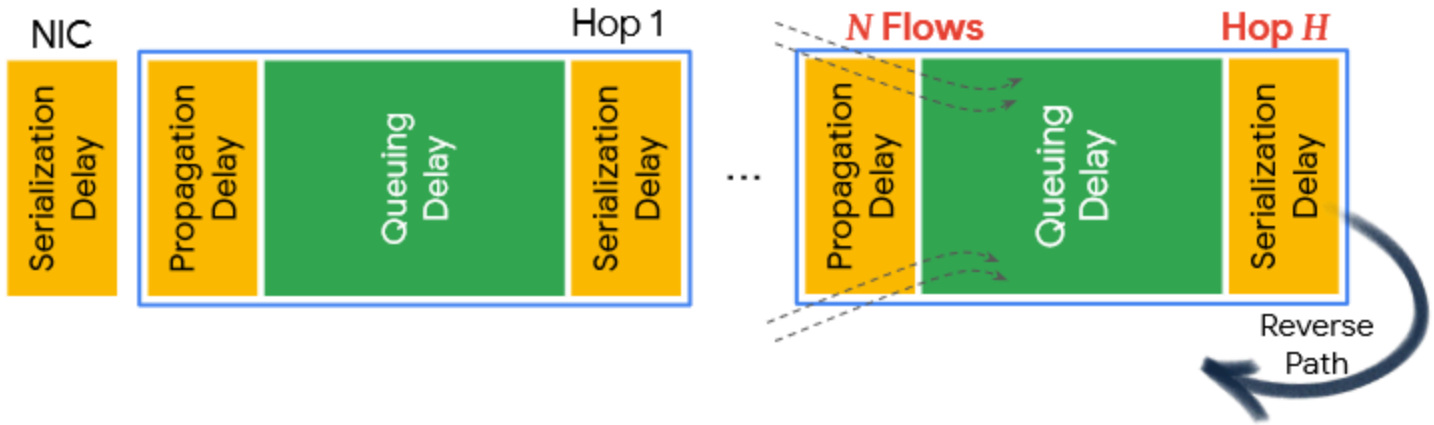

Figure 3: Target delay encapsulates both fixed and variable parts and is dynamically scaled based on topology and load.

Fabric Target Delay 包含 NIC 和交换机中串行化延迟、排队延迟、传播延迟等多种成分，因此根据拓扑结构和负载对其进行缩放：

Scaling the Fabric Target Delay to Letency of paths (II)

Topology-based Scaling

目标是对路径较短的流控制较小的目标延迟，而 DC 中网络拓扑固定，因此通过使用固定的基本延迟加上固定的每跳延迟，将流的网络路径转换为目标延迟：

- 从已知的起始TTL中减去接收到的IP 数据包中 TTL 的值来测量前向路径跳数，并将其写回ACK头中。

Flow-based Scaling

平均队列长度、缓冲区占用以 $\mathcal{O}(\sqrt{N})$ 的规模增长，N 是竞争流的数量。

当 Swift 收敛到其公平份额时，cwnd 与流的数量成反比。因此，将目标延迟按比例调整为 $\frac{1}{\sqrt{cwnd}}$，即目标延迟随着 cwnd 变小而增加。这种方法**除了在流很少的情况下降低排队率外，还提高了公平性**：它用较大的目标延迟来加速慢速流，用较小的目标延迟来减缓快速流。

Scaling the Fabric Target Delay to Letency of paths (III)

Overall Scaling

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

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

- $\#hops \times \hbar$ 是跳数与逐跳放缩因子之积，其含义正是对 Topology-based Scaling 的考量；
- $\max()$ 则是由 cwnd **与流的数量成反比** 这一规律得出。

Solution for Large-Scale Incast

```
if cwnd <= cwnd_prev then
    t_last_decrease = now
if cwnd < 1 then
    pacing_delay = rtt / cwnd
else
    pacing_delay = 0
```

- Swift 允许拥塞窗口 *cwnd* 小于 1，从而有效应对大规模 incast 问题；
- *cwnd* = 0.5，意味着推迟 2*RTT 后发送数据包；这样的 pacing-adjusted 对于保持 low latency 和 loss 是非常有用的；
- ❓为什么论文中说：*But for a Snap transport that operates in MTU-sized units, pacing is mostly not necessary for performance, nor is it CPU-efficient.*

Loss Recovery and ACKs

Swift 的丢包率很低，其与传统 TCP 的数据包丢失检测机制相同：

- Selective ACK for fast recovery, and
- retransmission timer to ensure data delivery in the absence of ACKs from the receiver.

同样如果发生重传超时，意味着链路严重拥塞，因此通过 MD 机制来快速缩小拥塞窗口。

Swift 不使用显式的 delay ACK 来实现连续确认，而是对收到的数据包有节奏地立即发送 pure ACK ，从而解除对远程端的阻塞。

? Role of QoS

Performance: Swift vs. GCN

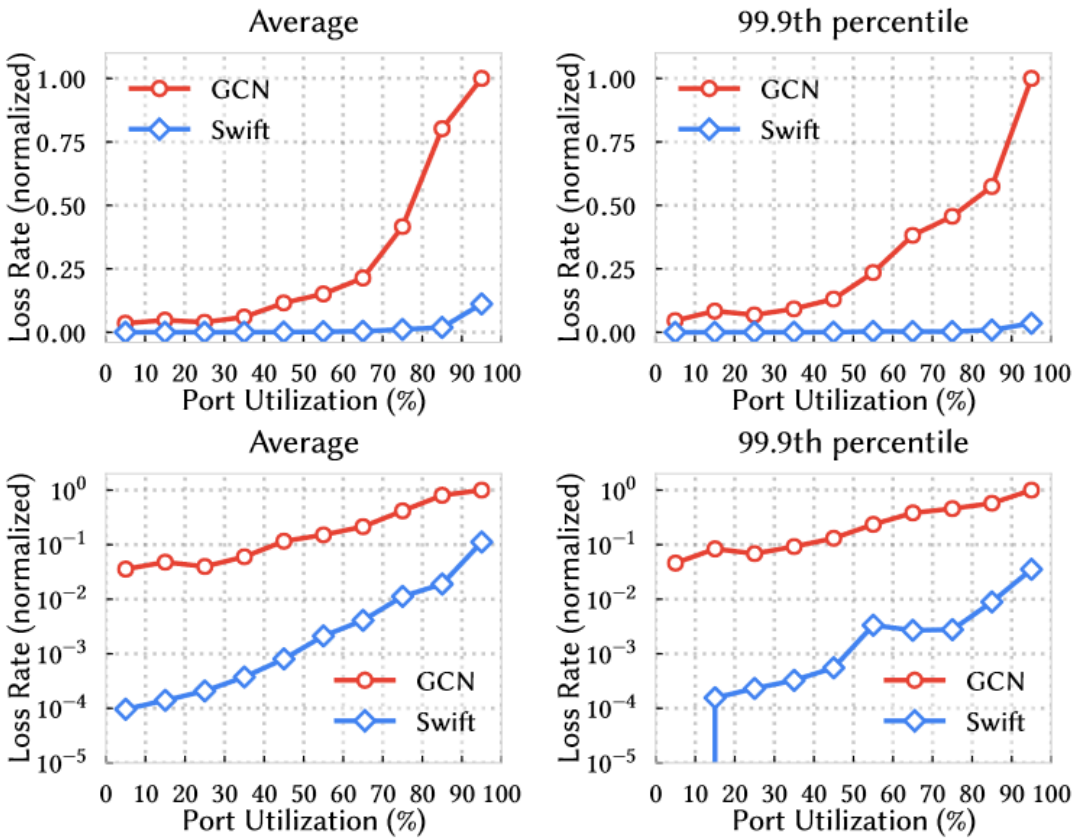

Figure 6: Edge (ToR to host) links: Average and 99.9p Swift/GCN loss rate (linear and log scale) vs. combined utilization, bucketed at 10% intervals. Loss rate is normalized to highest GCN loss rate. The near-vertical line in the log-scale plot is due to extremely small relative loss-rate.

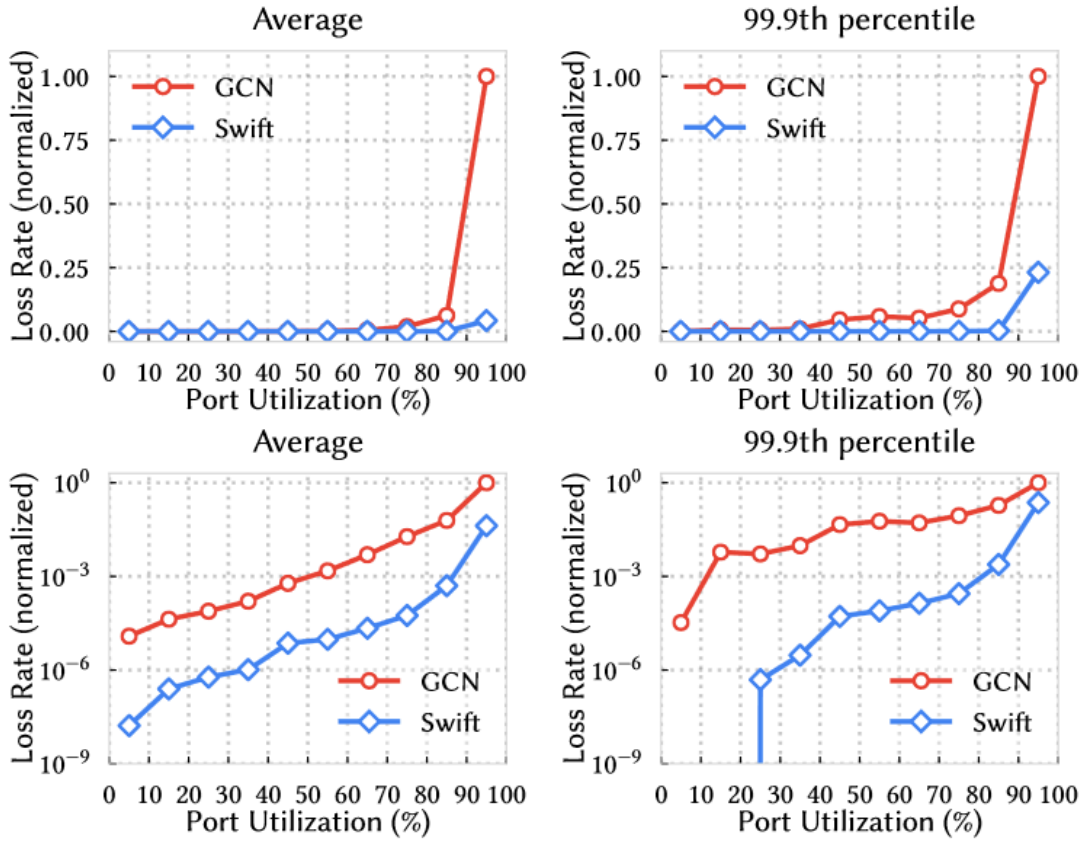

Figure 7: Fabric links: Average and 99.9p Swift/GCN loss rate Swift/GCN loss rate (linear and log scale) vs. combined utilization, bucketed at 10% intervals.

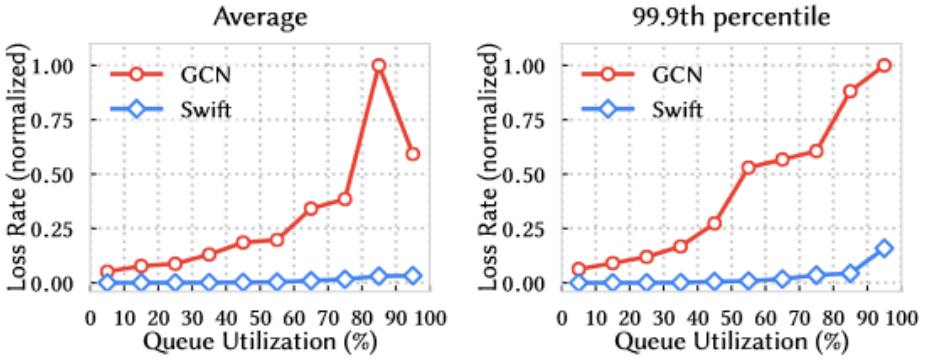

Figure 8: Average and 99.9th percentile loss rate vs. queue utilization.

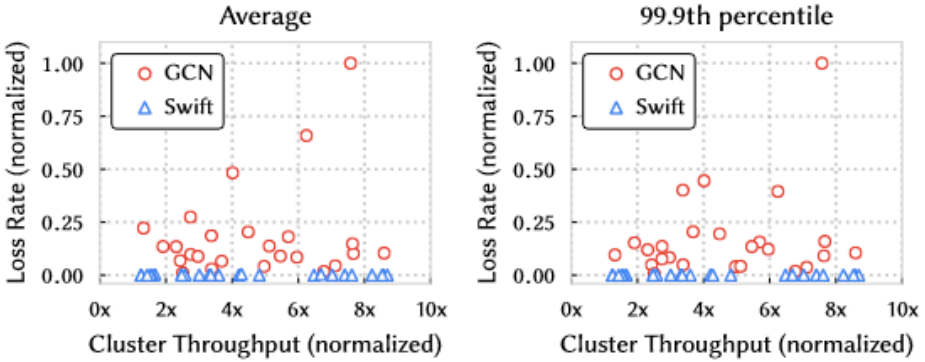

Figure 9: Edge (ToR to host) average and 99.9p loss rate vs. total Swift/GCN throughput in the cluster.

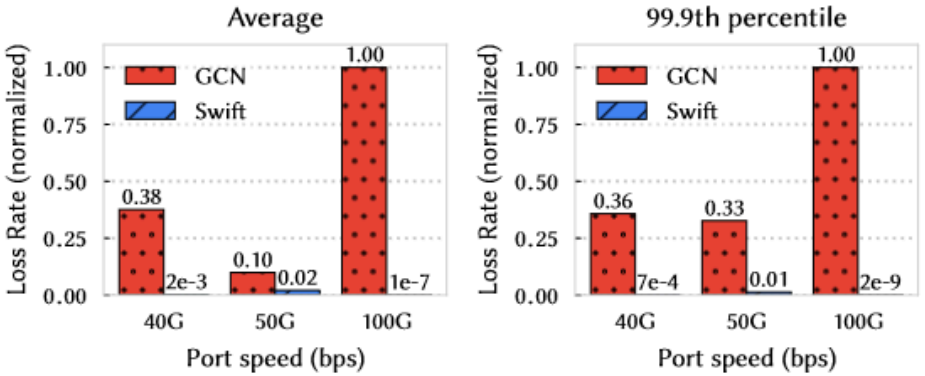

Figure 10: Average and 99.9p loss rate of highly-utilized (>90%) links in each switch group.

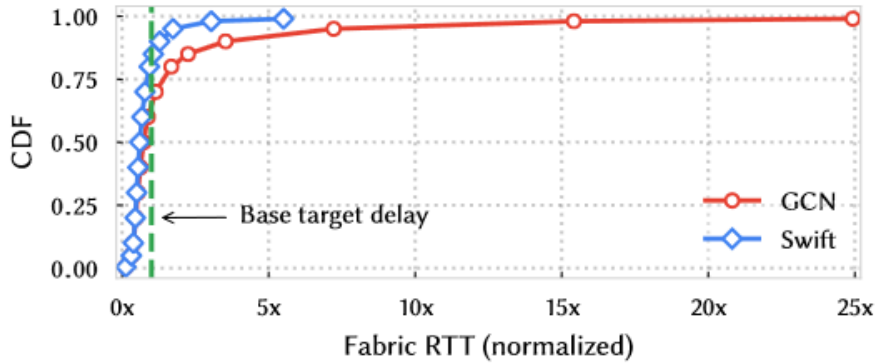

Figure 11: Fabric RTT: Swift controls fabric delay more tightly than GCN.

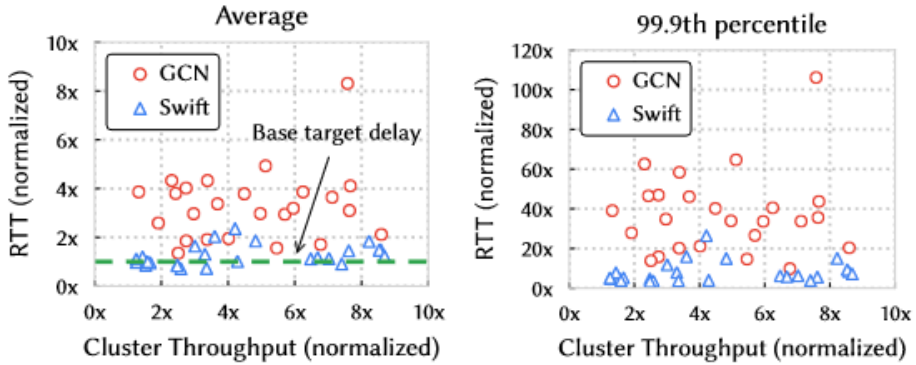

Figure 12: Cluster Swift/GCN Throughput vs. Average RTT. The dashed line is the *base target delay* (normalized to 1).

Experimental Results

Effect of Target Delay

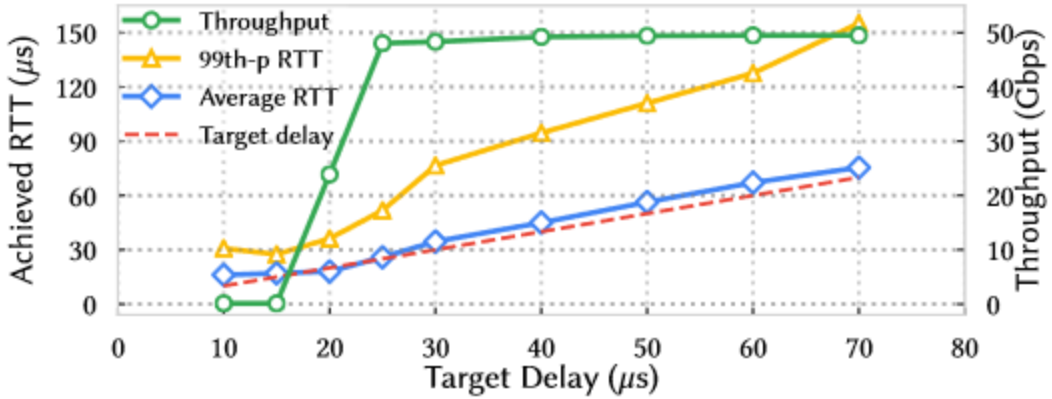

Figure 17: T_1: Achieved RTT and throughput vs. target delay, 100-flow incast.

- 基本延迟必须至少包括传播和NIC/交换机串行化延迟以及测量不准确。**超过这个最小值，更高的目标延迟允许更多的排队。我们希望目标延迟较低以减少延迟，但也要至少足够高以最大限度地提高网络吞吐量。**

Throughput/Latency Curves

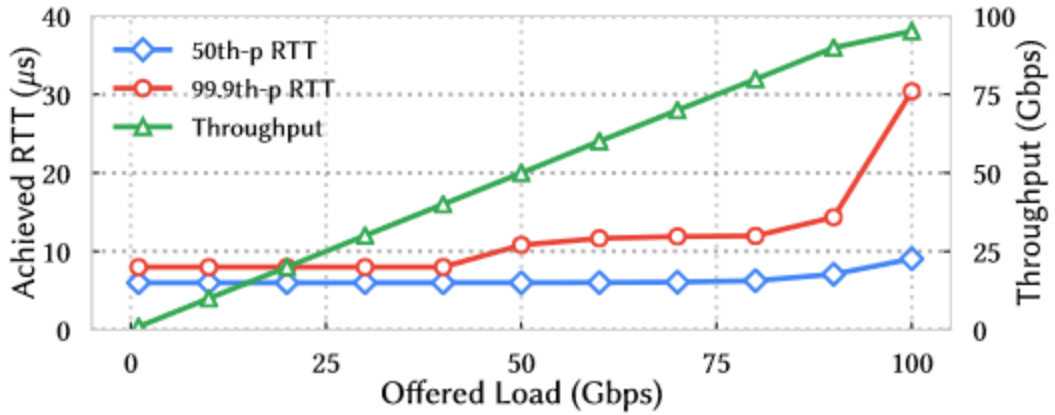

Figure 18: T_2: Achieved RTT and throughput vs. per-machine offered load. Total load varies from 500Gbps to ~50Tbps.

- **吞吐量随着RTT的增加而增加，直到我们超过线路速率的80% 。**
- 在接近 100% 的负载时，Swift 仍能够在接近 50Tbps 的总负载下将 99.9th-p RTT保持在小于 50μs 。

Large-scale Incast

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.

- 前文提到 Swift 为了提高对 large-scale incast 的支持，设置 cwnd<1 ，上面的数据中可以看到：
- **Swift以低延迟和几乎零损失的方式实现了线速率吞吐量——对于 5000:1 的 incast 来说，这是一个出色的性能。相反，在不支持cwnd<1的情况下，协议会降级为高延迟和高丢包率，从而降低吞吐量。**

Endpoint Congestion

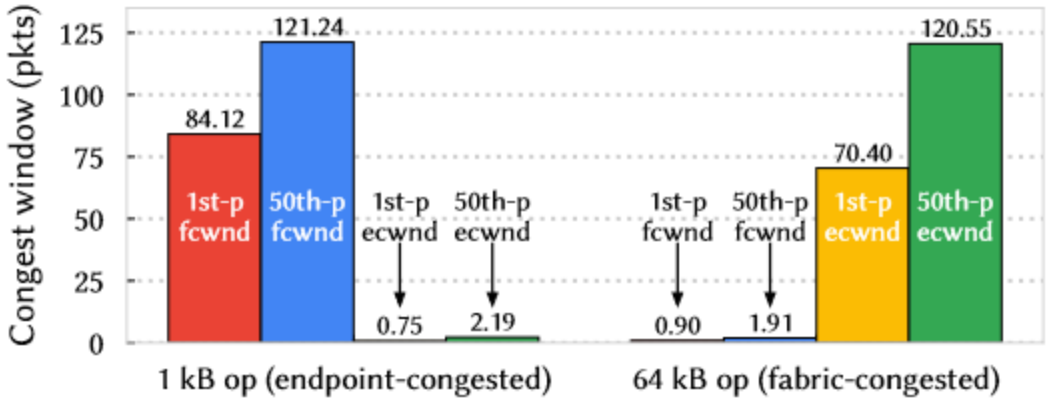

Figure 19: T_1: Fabric ($fcwnd$) and Endpoint ($ecwnd$) congestion windows for a 100-to-1 incast with 1kB and 64kB writes.

- 随着工作负载的变化，我们看到了明显的区别：**IOPS密集型情况受到端点窗口的限制，而字节密集型情况则受到结构窗口的限制。**
- 将端到端RTT分解为结构和端点两部分，使Swift能够对网络和主机的拥塞做出不同的响应。

Flow Fairness

Experimental Results

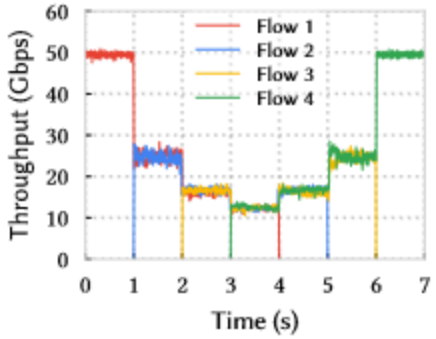

Figure 20: T_1: Throughput of four flows shows fairness achieved by Swift.

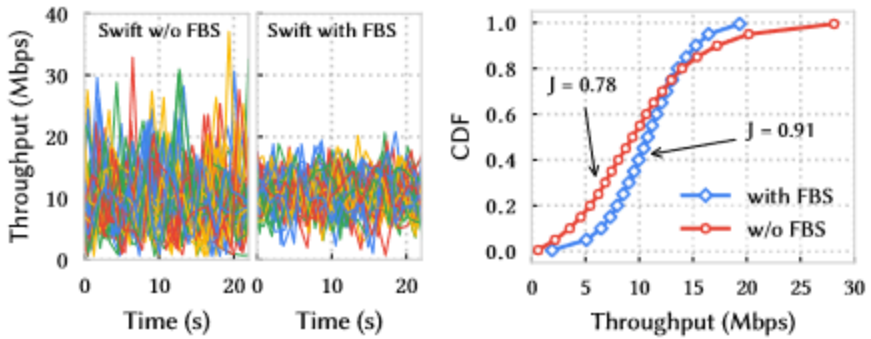

Figure 21: T_1: Throughput with and without flow-based scaling (FBS) for a 5000-to-1 incast. Jain's fairness index (J) shown is measured amongst all 5000 flows using a snapshot of flow rates.

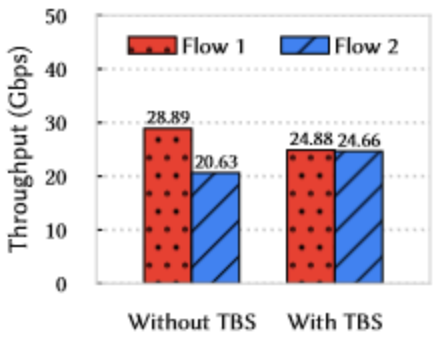

Figure 22: T_1: Throughput of two flows with different path lengths, with and without topology-base scaling (TBS).

- 在图20中，我们从一对机器之间的单个流开始。保持目标机器不变，我们从不同的源机器增量添加一个流，然后开始逐个拆除这些流。我们看到流量分配是严格和公平的。
- 图21中绘制了随时间变化的吞吐量、流速的CDF和Jain的公平指数。尽管在50Gbps链路上，每个流的公平共享速率仅为10Mbps，但Swift以0.91的Jain公平指数实现了良好的公平性。
- Swift根据网络路径长度缩放流的目标延迟。这不仅减少了较短路径的延迟，而且提供了与流的基本RTT无关的公平性。图22中的结果显示了公平吞吐量水平的显著提高。

Over😆 Thanks for listening!