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

Gautam Kumar, et, al.
Google LLC

ACM SIGCOMM `20

Presenter: Junghwan Song

## Outline

- Introduction
- Design
    - Design requirements
    - Key ideas
    - Two cwnd
    - cwnd < 1
- Evaluation
- Conclusion

# Needs for low-latency in datacenters - 1

- Recently, ‘resource disaggregation’ in datacenters becomes a key driver for low-latency
  - Data center applications each use different resources
  - 1.5% of applications consume 98.5% of CPU resources, another 1.8% of applications use 98.2% of memory resources *

- With disaggregation, low-latency messaging is needed to tap the potential of next-generation storage
  - 100μs access latency at 100k+ IOPS to use Flash effectively
  - Upcoming NVMe [55, 56] needs 10μs latency at 1M+ IOPS

* Sheng Di et al., “Characterizing Cloud Applications on a Google Data Center,” ICPP2013, 2013, pp.468-473.

# Needs for low-latency in datacenters - 2

- Tight tail latency is also important
  - Datacenter applications often use partition-aggregate communication patterns across many hosts
  - BigQuery *, a query engine for Google Cloud, relies on a shuffle operation with high IOPS per server

➔ Congestion control for low-latency is a key enabler (or limiter) of performance in datacenters

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

## Design

## Design requirements

- Provide low, tightly-bound network latency, near zero loss, and high throughput while scaling to a large datacenter across a range of workloads
- Provide end-to-end congestion-control that manages congestion not only in the network fabric but also in the NIC, and on hosts
- Be highly CPU-efficient so as to not compromise an otherwise CPU-efficient OS bypass communication

## Key ideas - 1

- Use of HW and SW timestamps
  - To provide accurate delay measurements and separate them into fabric and host components
- Simple AIMD based on a target-delay
  - If observed delay < target delay, increase cwnd (AI)
  - Otherwise, decrease cwnd (MD)

# Key ideas - 2

- Seamless transition between cwnd and rate
  - Swift allows **cwnd to fall below 1** to handle large-scale incast
  - cwnd < 1 implemented via pacing using Timing Wheel
- Scaling of target-delay
  - Target scaled as per known network-distances
- Loss recovery and ACK
  - Minimal investment in loss-recovery – losses are rare
  - SACK for fast recovery, ack-loss for timeout

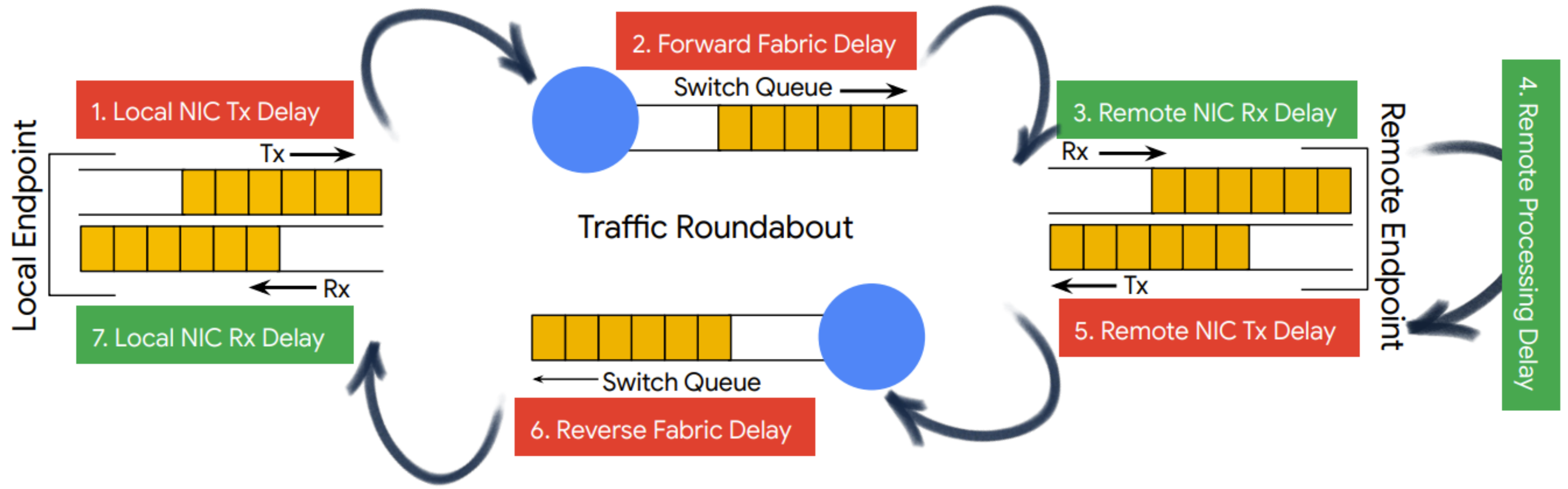
## Core keywords I think..

- **Delay based** congestion control

- **Two components in delay:** fabric and end-point

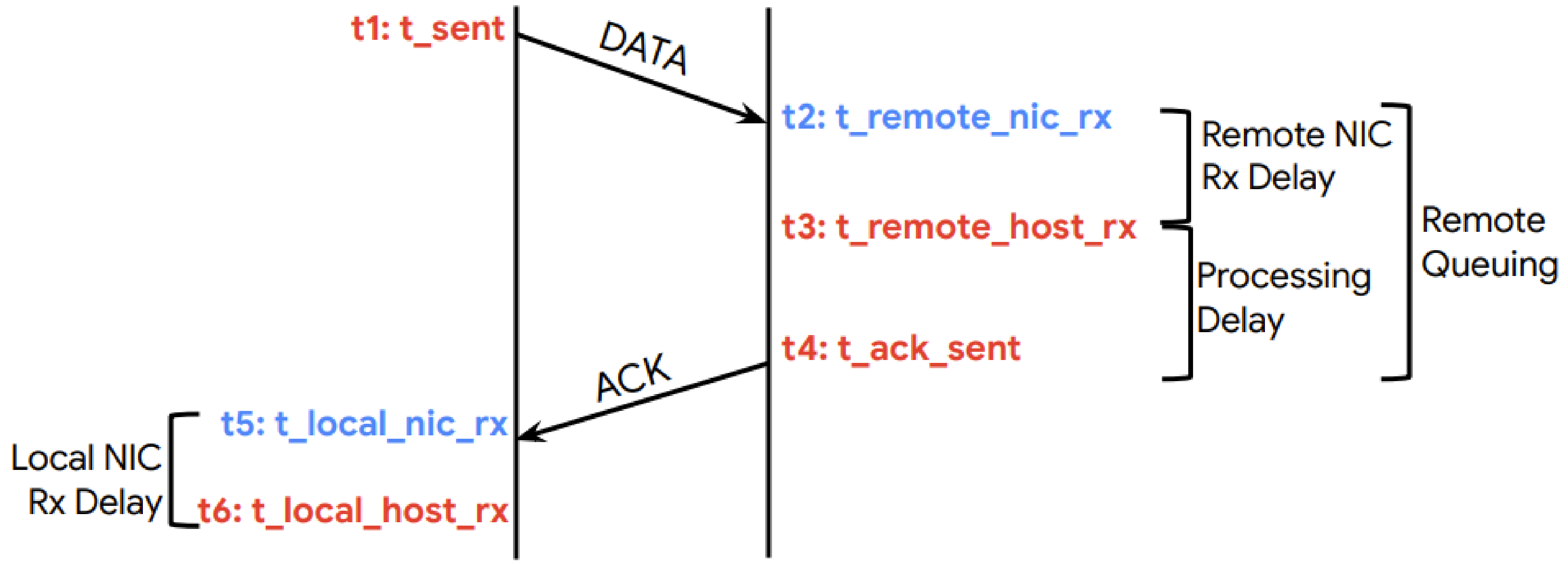
- **cwnd < 1** for large incast

# Two components of delay



- Swift maintains **two congestion windows**
  - fcwnd: The reds based on fabric-delay
  - ecwnd: The green based on endpoint-delay
- Effective cwnd: min(fcwnd, ecwnd)

# How to measure delays

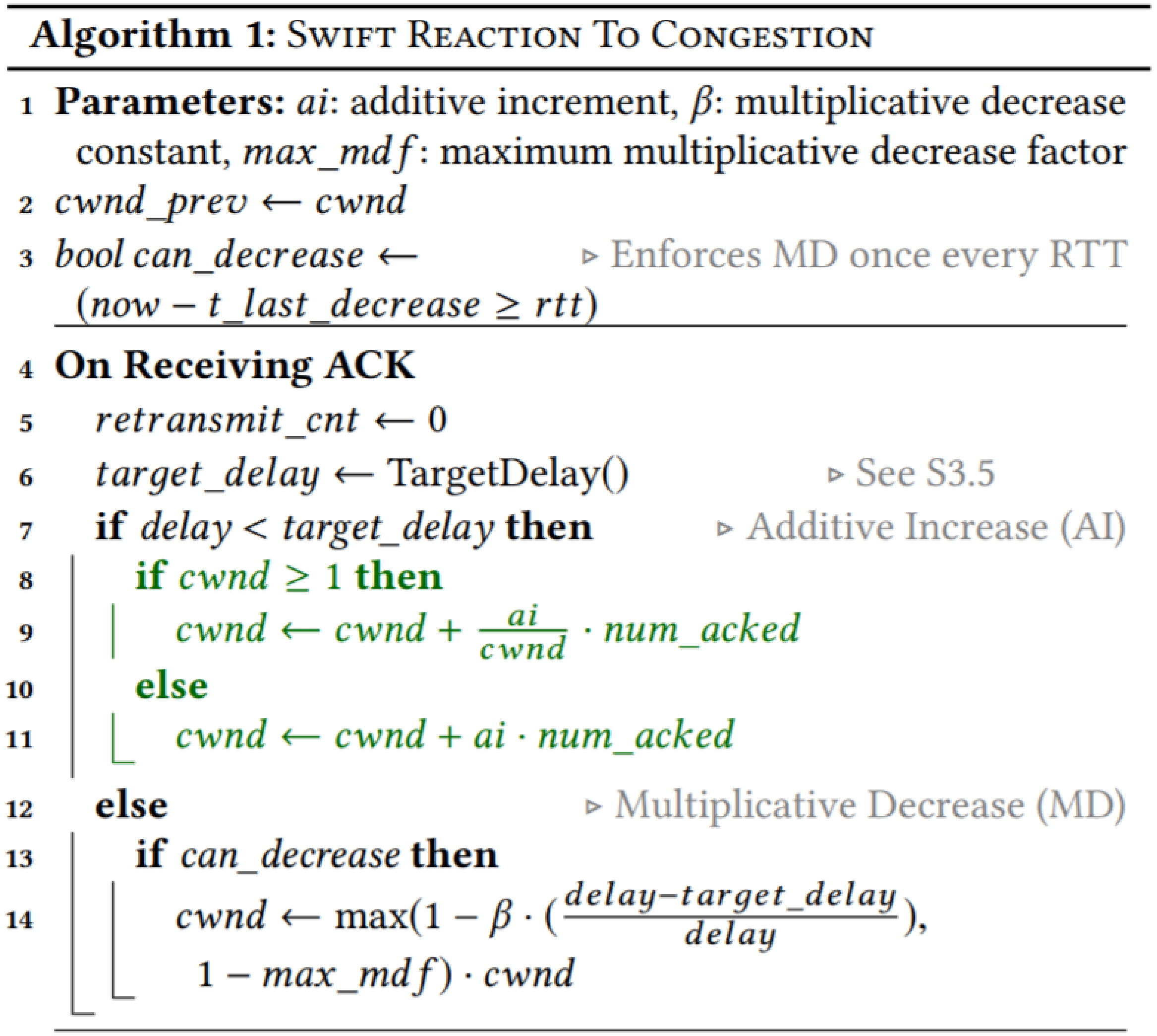


- End-to-end RTT is t6-t1
  - Sender NIC calculates locally using hardware timestamps
- Remote endpoint delay is t4-t2
  - Receiver should synchronize NIC and host clock
  - 4 bytes in ACK header are used for sending endpoint delay to sender

## Target delay window control

- Simple AIMD algorithm based on delay

- If delay > target delay
  - Increase cwnd by ai/cwnd
- Otherwise
  - Decrease cwnd depending on how far the delay is from the target



**Algorithm 1:** SWIFT REACTION TO CONGESTION

```
Parameters: $ai$: additive increment, $\beta$: multiplicative decrease
  constant, $max\_mdf$: maximum multiplicative decrease factor
$cwnd\_prev \leftarrow cwnd$
$bool\ can\_decrease \leftarrow$                ▹ Enforces MD once every RTT
  $(now - t\_last\_decrease \geq rtt)$
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
      $cwnd \leftarrow \max(1 - \beta \cdot (\frac{delay - target\_delay}{delay}),$
        $1 - max\_mdf) \cdot cwnd$
```

# How to calculate target delay

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

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

Fixed part;

Delays incurred for a single hop network with a few flows;

Including propagation delay, serialization delay, queueing delay, errors

Topology scaling;

Datacenter topology is known;

#hops = Starting TTL – received TTL, h = per hop scaling factor

Flow scaling;

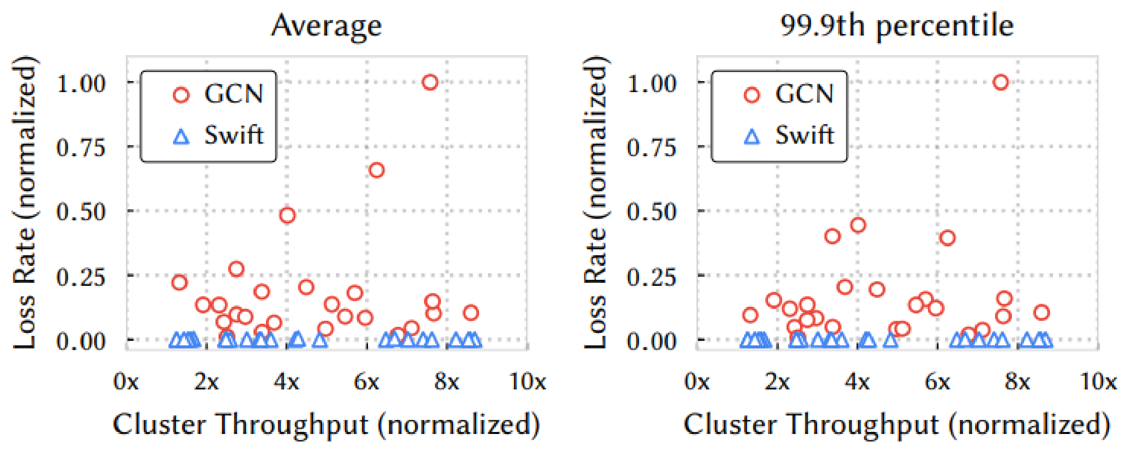
Considering bottleneck with N flows;

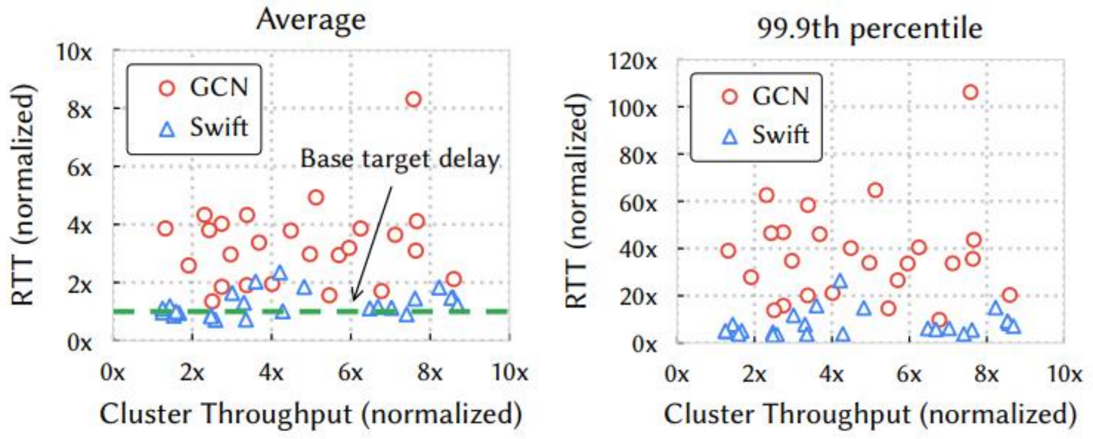
## cwnd < 1 for large-scale incast

- Some datacenter applications rely on extremely large incasts
  - Thousands of flows destined to a single host simultaneously
- Number of flows can exceed the path BDP
  - Even a congestion window of one
- Swift allows cwnd to fall below one packet
  - Minimum of 0.001 packets
- Pacing is implements using a Timing Wheel

## Evaluation & conclusion

# Evaluation



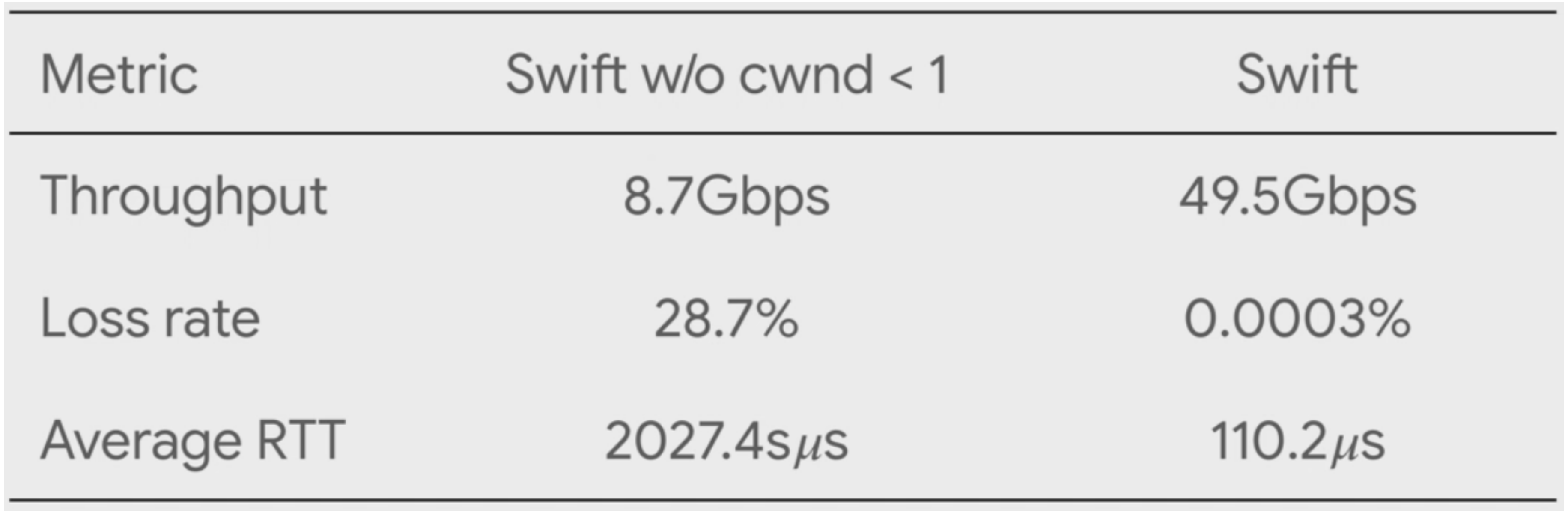


- Swift keeps loss-rates very small even at the 99.9th-p and at near line-rate utilization

- Similarly, Swift also shows shorter RTT than GCN (delay-based congestion scheme)

- Loss-rate and RTT improvements do not come at the cost of throughput

## Evaluation



| Metric | Swift w/o cwnd < 1 | Swift |
|---|---|---|
| Throughput | 8.7Gbps | 49.5Gbps |
| Loss rate | 28.7% | 0.0003% |
| Average RTT | 2027.4s$\mu$s | 110.2$\mu$s |

- 5000-to-1 incast support at line-rate 50Gbps
- Allowing cwnd to fall below 1 and using pacing via Timing Wheel is crucial

# Conclusion

## Delay works well

Use of delay as a multi-bit congestion signal has proven effective for excellent performance

Use of absolute target delay is performant and robust

Simplicity that has helped greatly with operational issues

## Fabric and host congestion are both important

Both forms matter across a range of workloads

Delay is decomposable to separate concerns

Important for end-to-end performance for apps

## Wide range of workloads

Including large scale incast

Pace packets when there are more flows than the BDP

Use a congestion window at higher flow rates for CPU efficiency