Programmability in NICs for Congestion Control and Transport

Talk @P4 Workshop, May 2021

Nandita Dukkipati, Konstantin Weitz

Google

Key Questions

1

What transport and congestion control capabilities make sense in NICs?

2

Which of the transport capabilities require programmability?

3

Can the transport functionality be expressed with P4?

Google

This Talk

- Context: Congestion Control @Google and Why it Matters.
- Swift Congestion Control and NIC Time as a Service.
- Example: Expressing Congestion Control Functionalities with P4.

Google

Congestion Control @Google and Why it Matters

Bandwidth Management @Google

Swift[1], BBR[2]	Per-flow congestion control.
BwE [3], B4 TE [4]	Centralized control of flow aggregates over WAN.
QoS	Bandwidth sharing at network queues.
Static Limits	BW configuration based on CPU cores, storage etc.

[1] Swift: Delay is Simple and Effective for Congestion Control in the Datacenter, SIGCOMM 2020
[2] BBR: Congestion-based Congestion Control, ACM Queue, 2016
[3] BwE: Flexible, Hierarchical Bandwidth Allocation for WAN Distributed Computing, SIGCOMM 2015.
[4] B4: Experience with a Globally-Deployed Software Defined WAN, SIGCOMM 2013.

Transport in Host Stacks

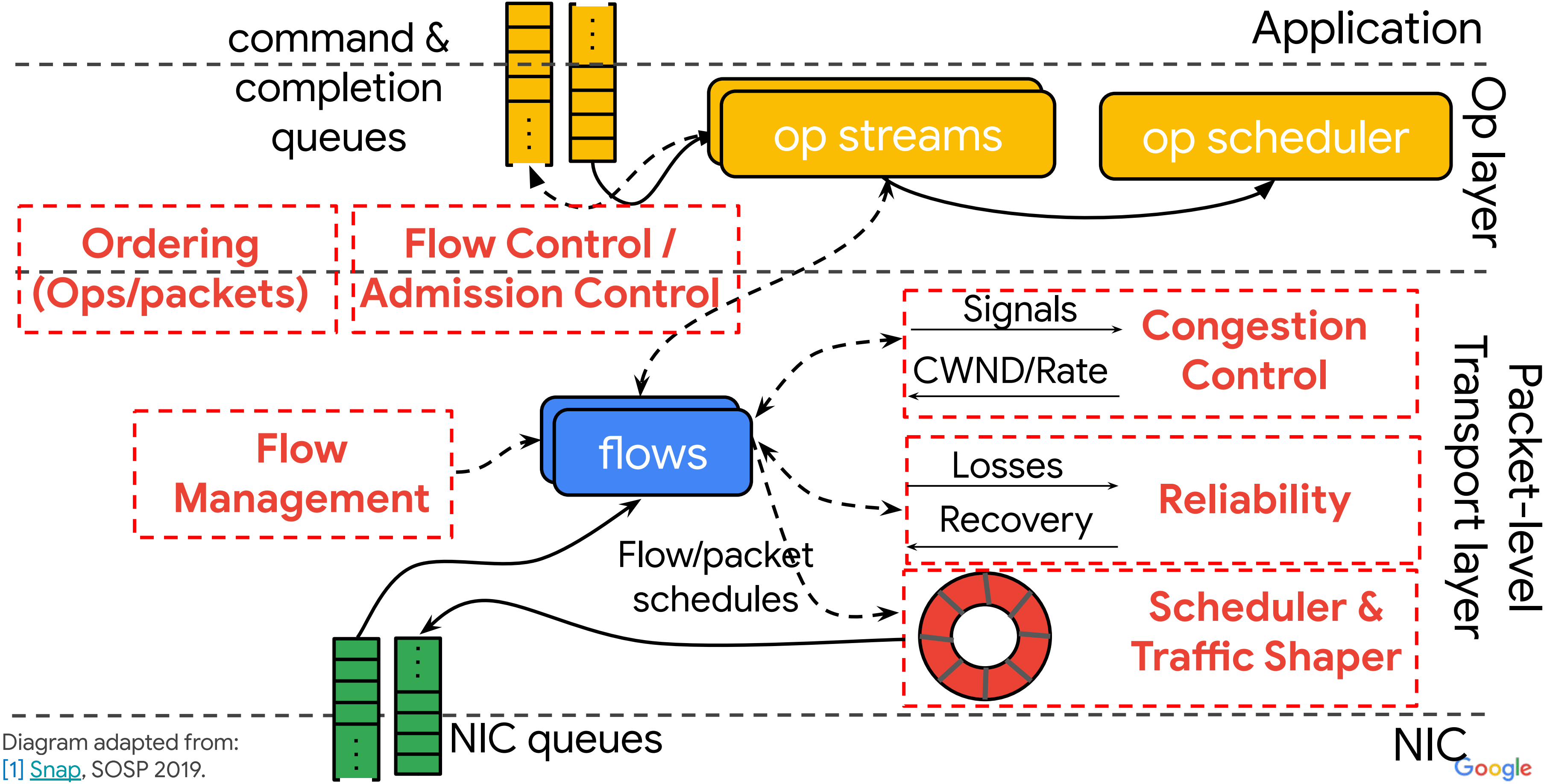


Diagram adapted from:
[1] Snap, SOSP 2019.
[2] Swift, SIGCOMM 2020.

Swift Congestion Control and NIC Time as a Service

Motivated by:
Swift: Delay-based congestion-control algorithm for low-latency networks - External Link

Google

What is Swift?

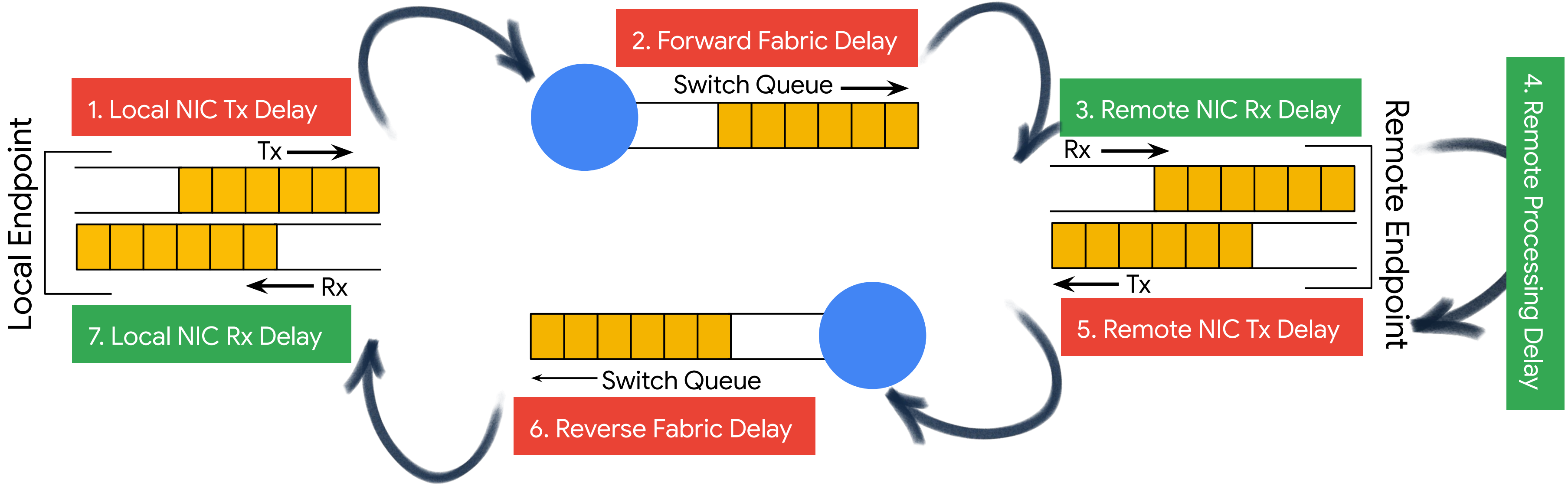
Swift is a delay based congestion-control for Datacenters that achieves low-latency, high-utilization, near-zero loss implemented completely at end hosts supporting diverse workloads like large-scale incast across latency-sensitive, byte and IOPS-intensive applications working seamlessly in heterogeneous datacenters with minimal switch support

Swift achieves ~50μs tail latency for short-flows while maintaining near 100% utilization even at 100Gbps line-rate

Google

Swift Design

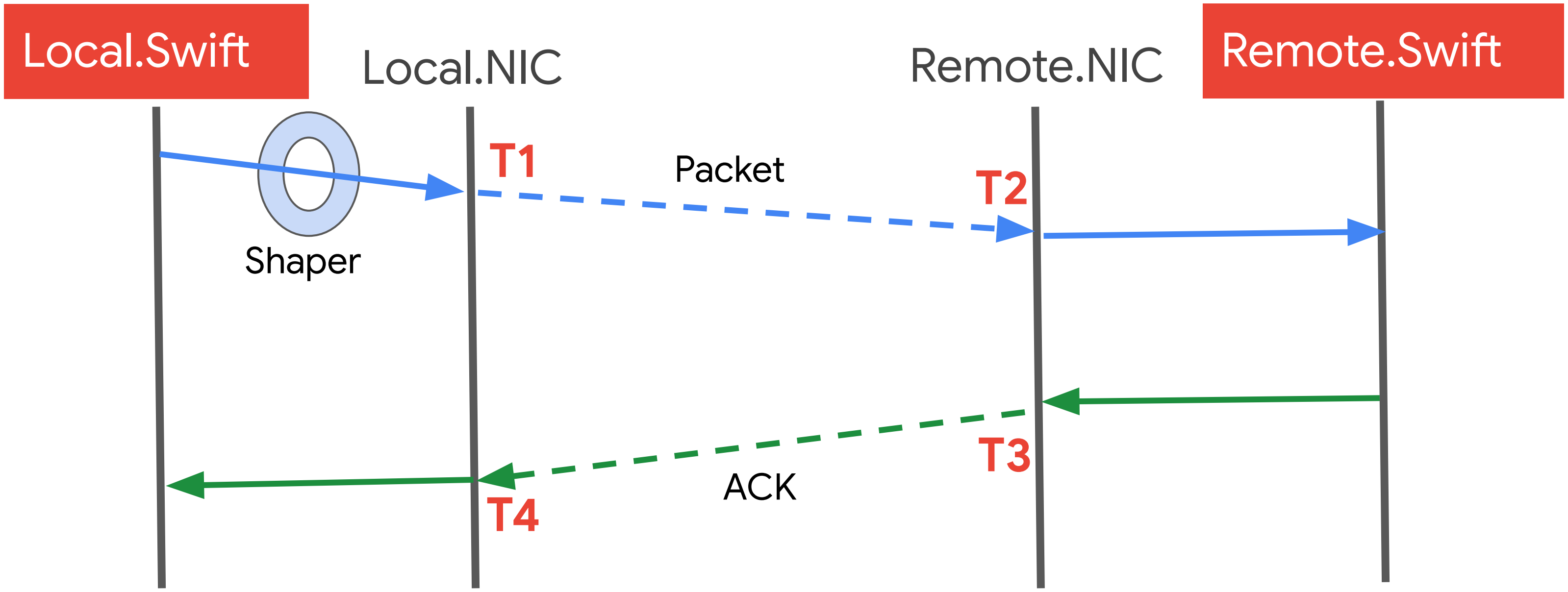
End-to-end delay decomposition of a Packet and its ACK



Swift maintains two congestion-windows (in #packets) - one based on fabric-delay and one based on endpoint-delay with their respective cwnd

Effective cwnd is the **minimum** of the two

Four Key Timestamps



T4-T1	Full round trip delay
(T4-T1)-(T3-T2)	Fabric only round trip delay
T2-T1	Forward fabric delay
T4-T3	Reverse fabric delay

Swift Design contd.

Simple AIMD based on a target-delay

```
if delay < Target
     increase cwnd
   (Additively)
else
     decrease cwnd
   (Multiplicatively)
```

Use of HW and SW timestamps

To provide accurate delay measurements and separate them into fabric and host components

Seamless transition b/w cwnd and rate

Swift allows cwnd to fall below 1 to handle large-scale incast

cwnd < 1 implemented via pacing using Timing Wheel, pacing off when cwnd > 1

Swift Design contd.

Scaling of target-delay

Topology-based scaling (TBS) for RTT-fairness

Flow-based scaling (FBS for fairness)

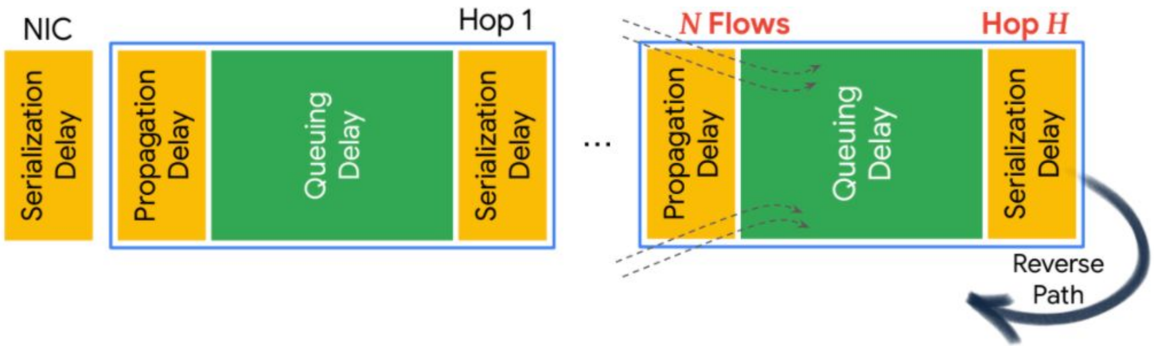
Loss recovery and ACKing policy

Minimal investment in loss-recovery - losses are rare: SACK and RTO.

Coexistence via QoS

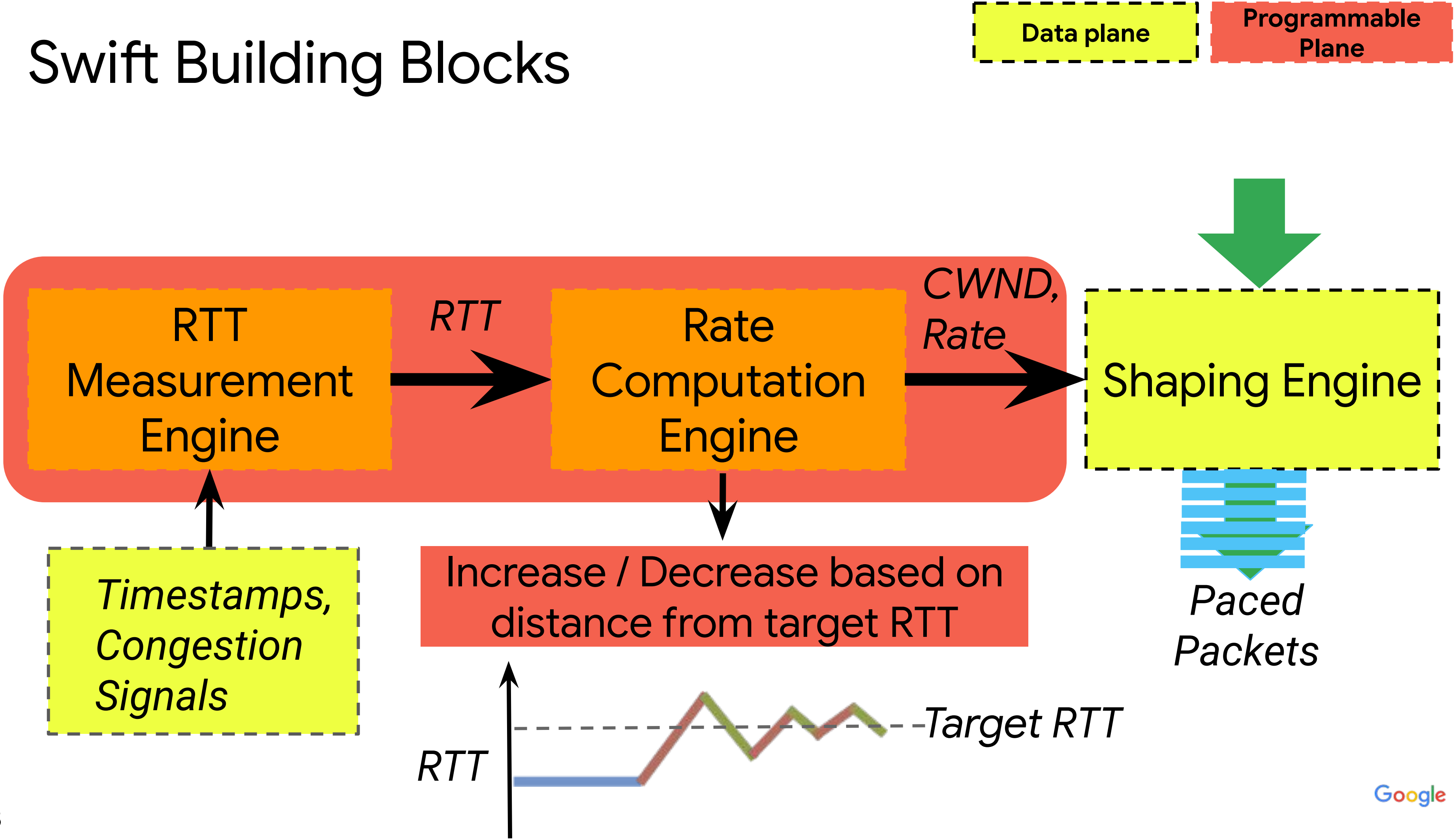
Multiple CC in shared deployments, e.g., WAN traffic, Cloud traffic etc.

Subset of QoS queues reserved for Swift



Google

Swift Building Blocks



Using P4 to realize programmability in Transport

Swift Overview

NIC

Connection State

```
struct ConnectionState {
  bit<16> congestion_window;
  bit<32> last_ack_sequence_number;
};

Register<ConnectionState>(2048) state;
```

maximum number of unacknowledged packets in the network

Packet Headers

```
header SwiftAck {
  bit<32> connection_id;
  bit<32> ack_sequence_number;
  // timestamps
  bit<32> t1;
  bit<32> t2;
  bit<32> t3;
  bit<32> t4;
};
```

RTT Measurement &

Rate Computation Engine

(we need this to be programmable)

Is P4 right for this?

We think **yes**.

Fundamentally, P4 transforms:

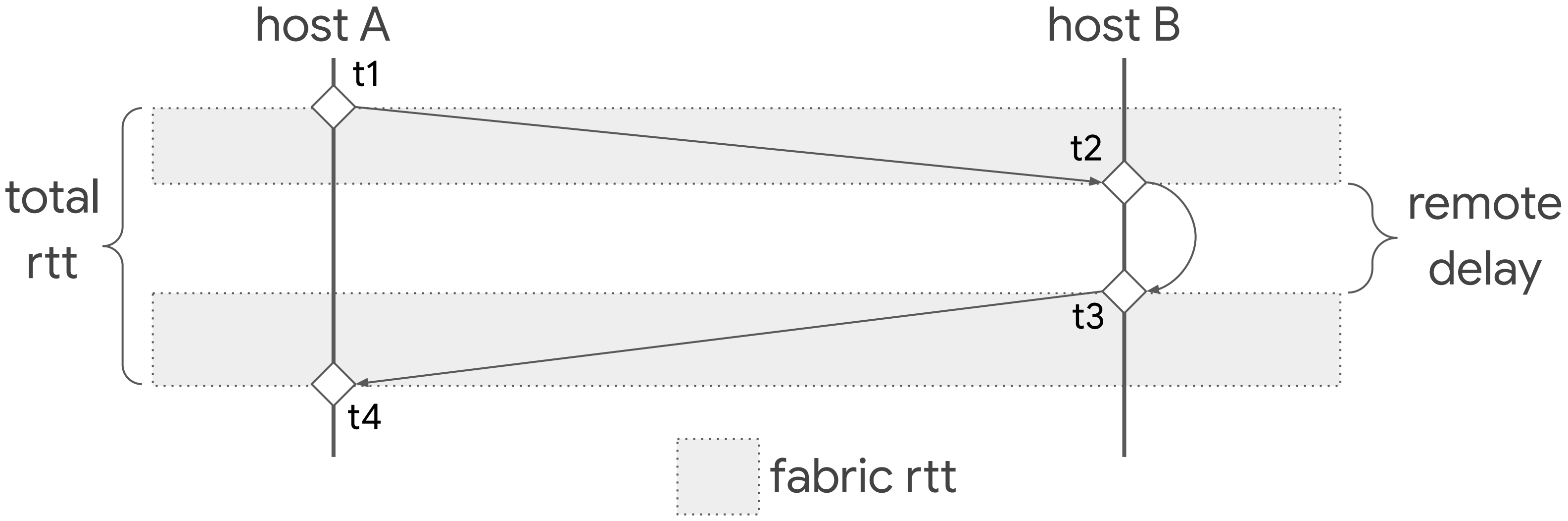
- a fixed size **input**, into // Packet, Connection State
- a fixed size **output**, using // Connection State
- a fixed amount of **computation** // No loops, recursion, etc

But there are also challenges:

- P4/PSA are targeted to switches (e.g. output is a packet).
  Portable NIC Architecture (PNA) should help [https://github.com/p4lang/pna]
- Hardware isn't quite right (need more registers/ALUs, and fewer TCAMs).
  We need your help.

Google

Computing Fabric Round Trip Time




```
bit<32> total_rtt = headers.swift.t4 - headers.swift.t1;
bit<32> remote_delay = headers.swift.t3 - headers.swift.t2;
bit<32> fabric_rtt = total_rtt - remote_delay;
```

Decreasing Congestion Window

Adjust congestion window almost proportionally to rtt, e.g.

fabric rtt = 60μs   current congestion window = 3 packets
target rtt = 40μs   updated congestion window = 2 packets

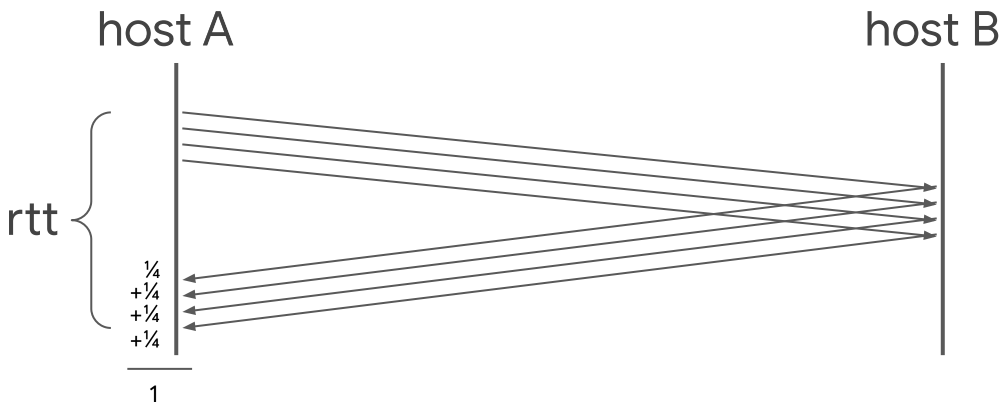
```
if (fabric_rtt > target_delay) {
    bit<32> delay_delta = fabric_rtt - target_delay;
    bit<32> decrease_scale = delay_delta / fabric_rtt;
    bit<32> decrease_factor = 1 - decrease_scale * 0.8;
    connection.congestion_window *= decrease_factor;
}
```

that's why it's just "almost" proportional

Increasing Congestion Window

Increase congestion window by 1 every RTT

e.g. congestion_window = 4, increase by ¼ for every ACK




```
if (fabric_rtt < target_delay) {
    bit<32> num_packets_acked = headers.swift.ack_sequence_number -
                                connection.last_ack_sequence_number;
    connection.last_ack_num = headers.swift.ack_num;
    connection.congestion_window += num_packets_acked / connection.congestion_window;
}
```

Swift-motivated Features and Programmability

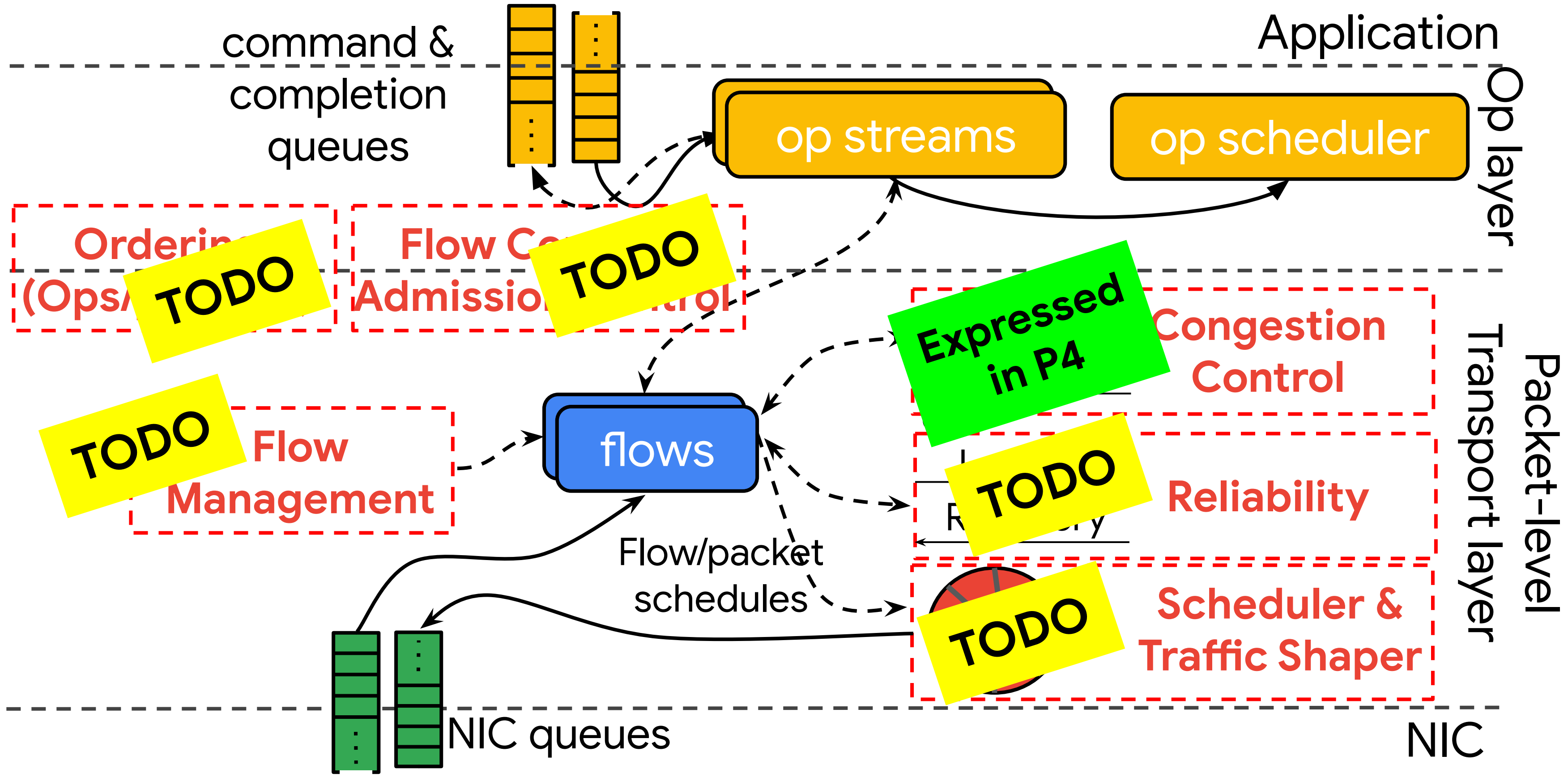
Features:

- Accurate **Tx** `(T1,T3)` and **Rx** `(T2,T4)` timestamps for every packet.
- Availability of `T1,T2,T3,T4` at Senders for LAN and RDMA datapaths.
- Accurate one-way delay (OWD) measurements based on synchronized NIC clocks.

Programmability: delay and rate computations.

- Instantaneous RTT; windowed min-RTT.
- Inference of congestion at end-host vs. fabric, sender vs. receiver.
- Congestion window adaptations based on RTT and OWD.

Takeaways



Future work: Express other building blocks that require programmability in P4 / P4++.

Open problem: Building hardware to run P4-expressed-transport.

Thanks to Many Contributors

Yuliang Li for direct contributions to Swift-on-P4.

Neal Cardwell, Prashant Chandra, Gautam Kumar, Masoud Moshref, Arvind Krishnamurthy, Naveen Kumar, Dan Lenoski, Parveen Patel, Amin Vahdat, Frank Wang, David Wetherall, Haiyong Wang, Hassan Wassel, and the Congestion Control Group @Google.

Google