Advanced Computer Networks

Data Center Transport

Lin Wang

Period 2, Fall 2022

VU VRIJE UNIVERSITEIT AMSTERDAM

Course outline

Essentials

- Introduction (history, principles)
- Networking basics
- Network transport

Data center networking

- Data center networking
- **Data center transport**
- Software defined networking
- Programmable data plane
- Network function virtualization

Network innovations

- Programmable switch architecture
- In-network computing - applications
- In-network computing - hardware
- Network monitoring
- Machine learning for networking

Guest lecture

- Fernando Ramos (University of Lisbon)

Learning objectives

What are the **new challenges** in data center transport?

What **design choices** do we have for data centers transport design?

What is special about data center transport?

Diverse applications and workloads

- Large variety in performance requirements

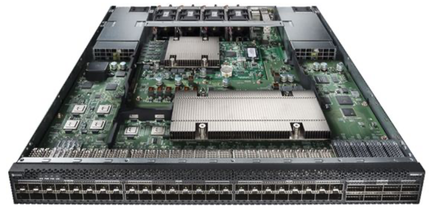
Traffic patterns

- Large long-lived flows vs small short-lived flows
- Scatter-gather, broadcast, multicast

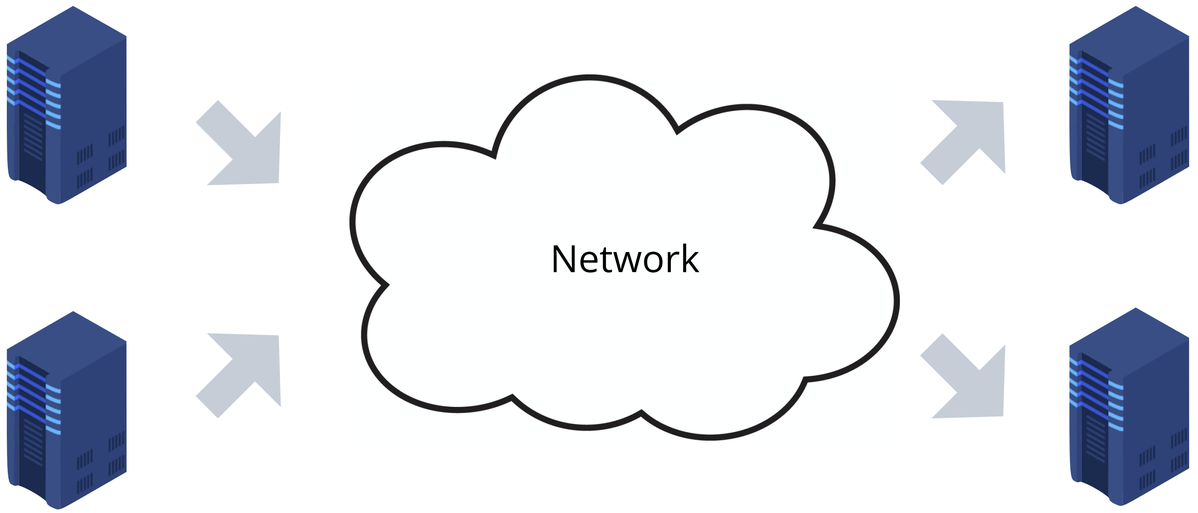
Built out of commodity components: no expensive/customized hardware

Network

- Extremely high speed (100+ Gbps)
- Extremely low latency (10-100s of us)

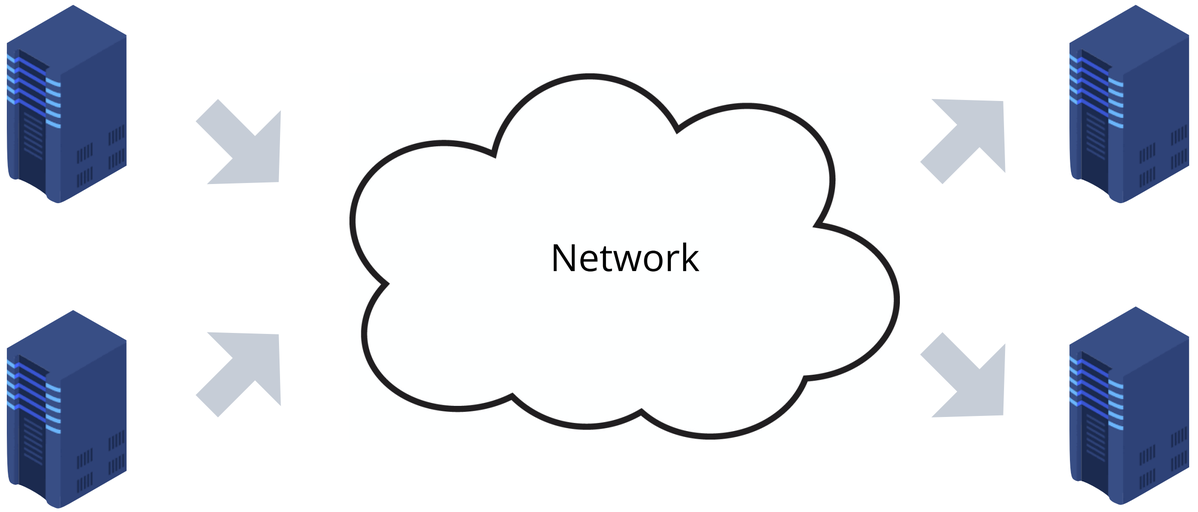


Congestion control recall



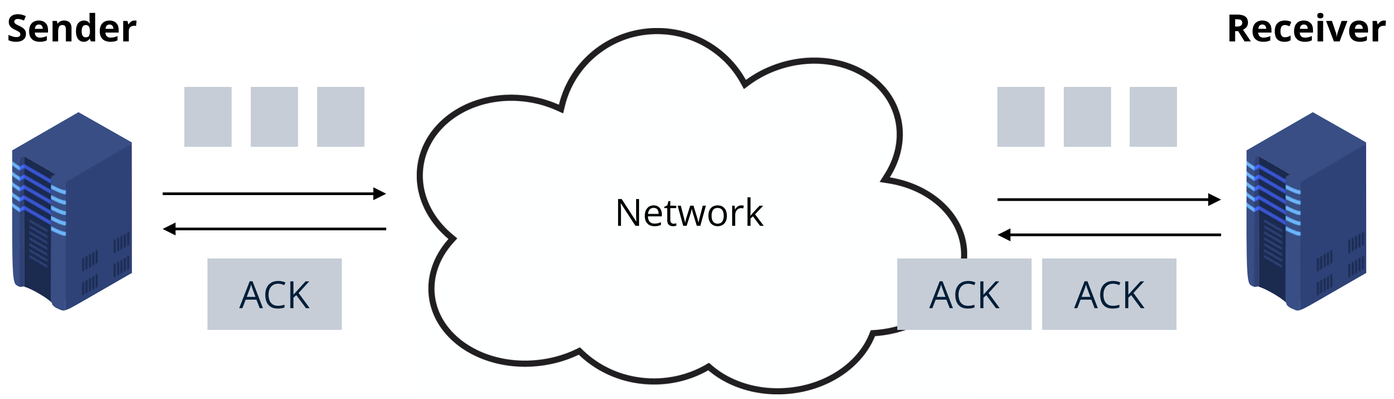
Do you still remember the goal of congestion control?

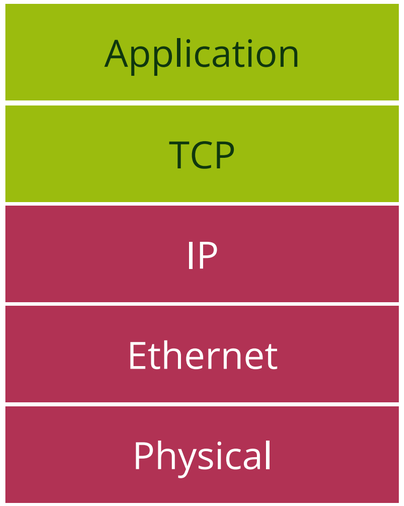
Congestion control recall



Congestion control aims to determine the **rate to send data** on a connection, such that
(1) the sender does not overrun the network capability and (2) the network is efficiently utilized

TCP





Reliable

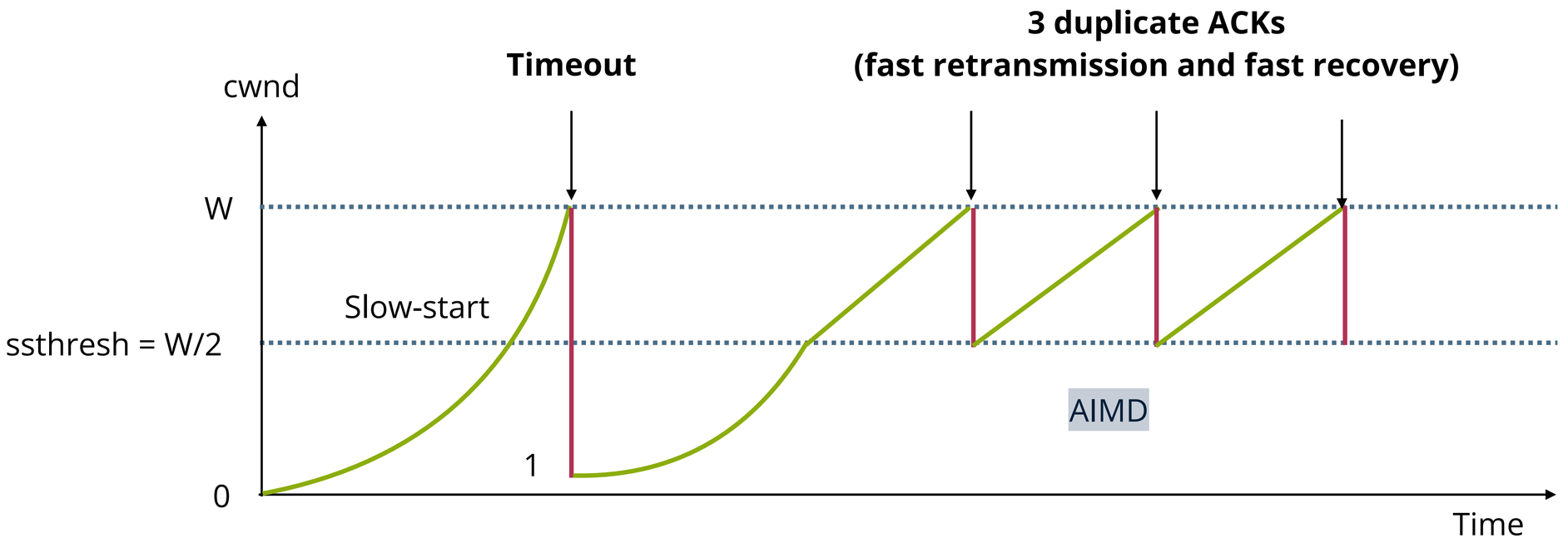
Lossy

The **transport layer** in the network model:

- Reliable, in-order delivery using sequence numbers and acknowledgements
- Make sure not to overrun the receiver (receive window, rwnd) and the network (congestion window, cwnd)
- What can be sent = min(rwnd, cwnd)

TCP AIMD

TCP Reno




```
if cwnd < ssthresh:     //slow-start
   cwnd += 1
else:                   // AIMD
   cwnd += 1 / cwnd
```

What could possibly go wrong in a data center environment?

TCP incast problem

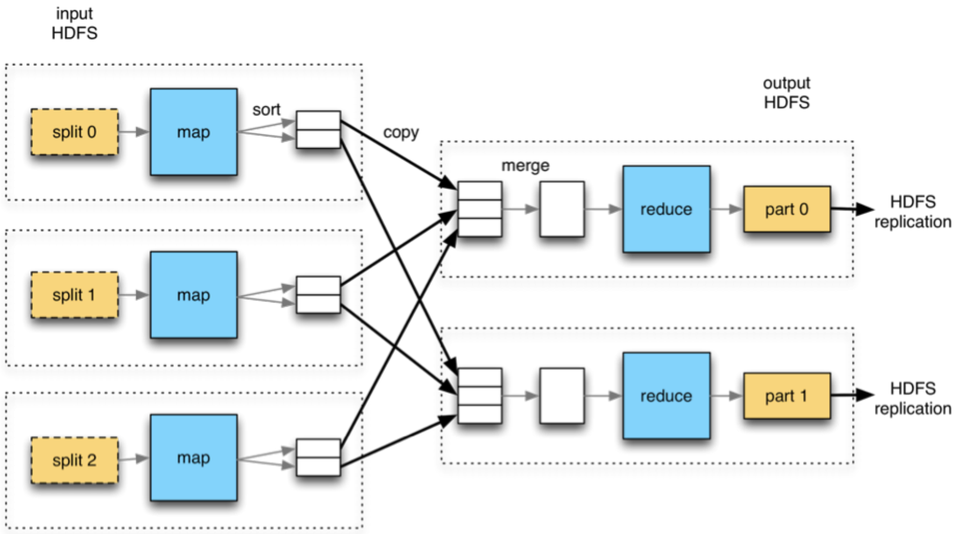
A data center application runs on multiple servers

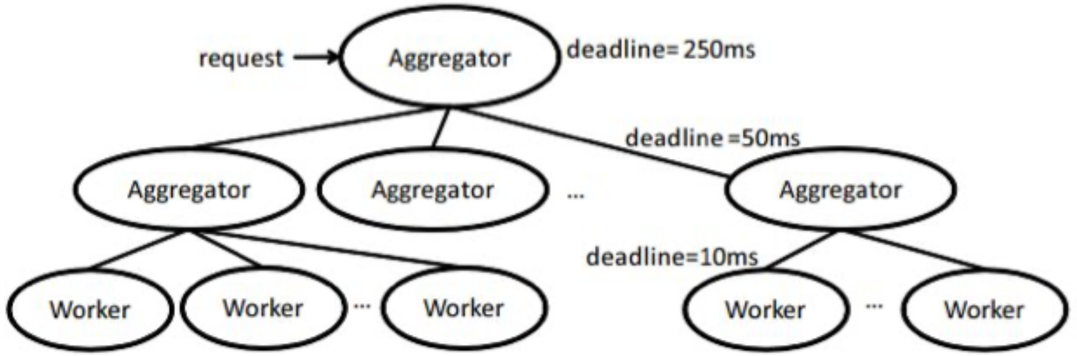
- Storage, cache, data processing (MapReduce)

They use a scatter-gather (or partition-aggregate) work pattern

- **[scatter]** A client sends a request to a bunch of servers for data
- **[gather]** All servers respond to the client

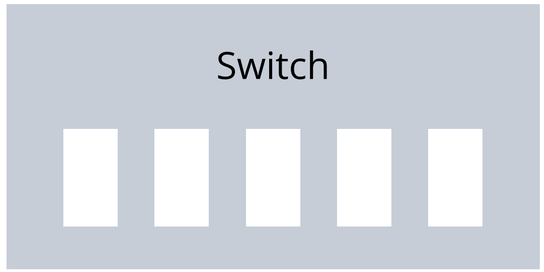
More broadly, a client-facing query might have to collect data from many servers





From a switch point of view

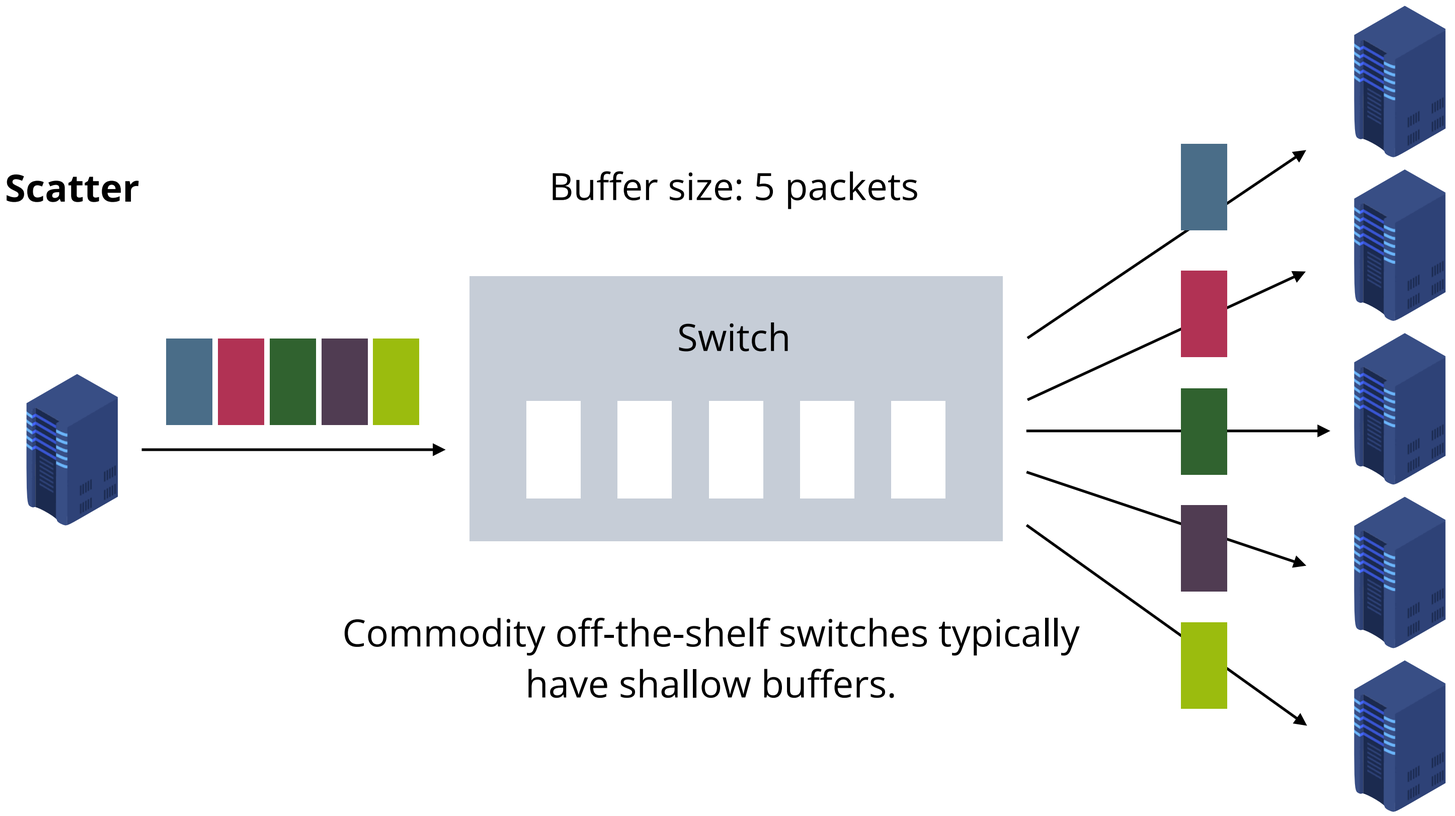
Buffer size: 5 packets



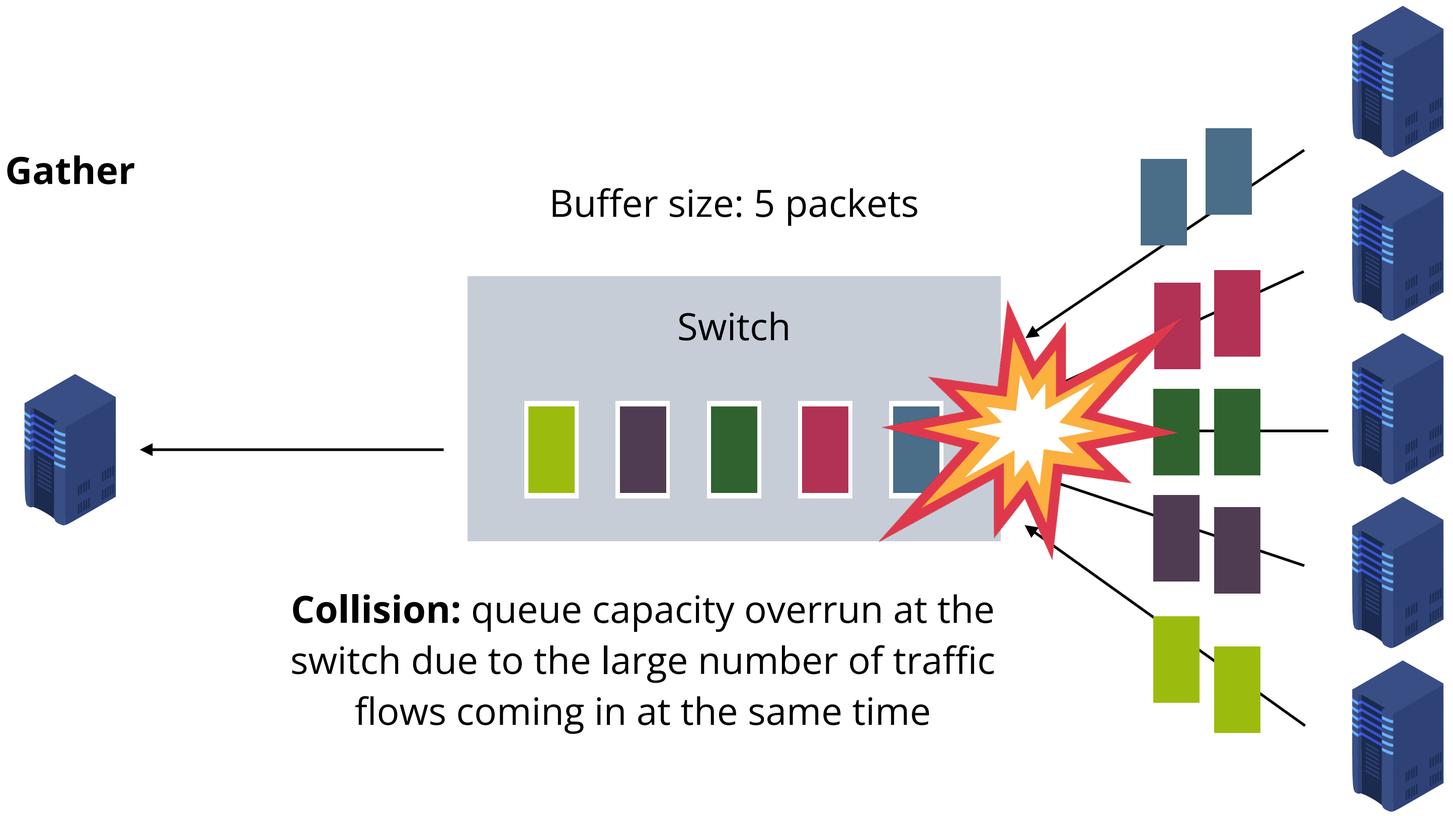
Commodity off-the-shelf switches typically have shallow buffers.

Do you know why?

From a switch point of view



From a switch point of view

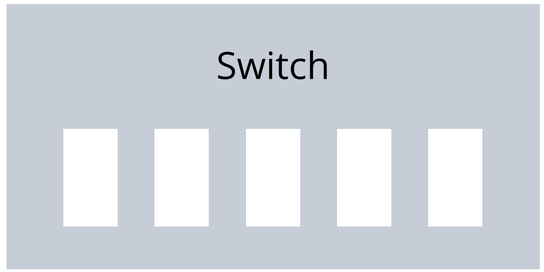


Collision: queue capacity overrun at the switch due to the large number of traffic flows coming in at the same time

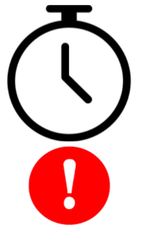
How does TCP handle this?

From a switch point of view

Buffer size: 5 packets

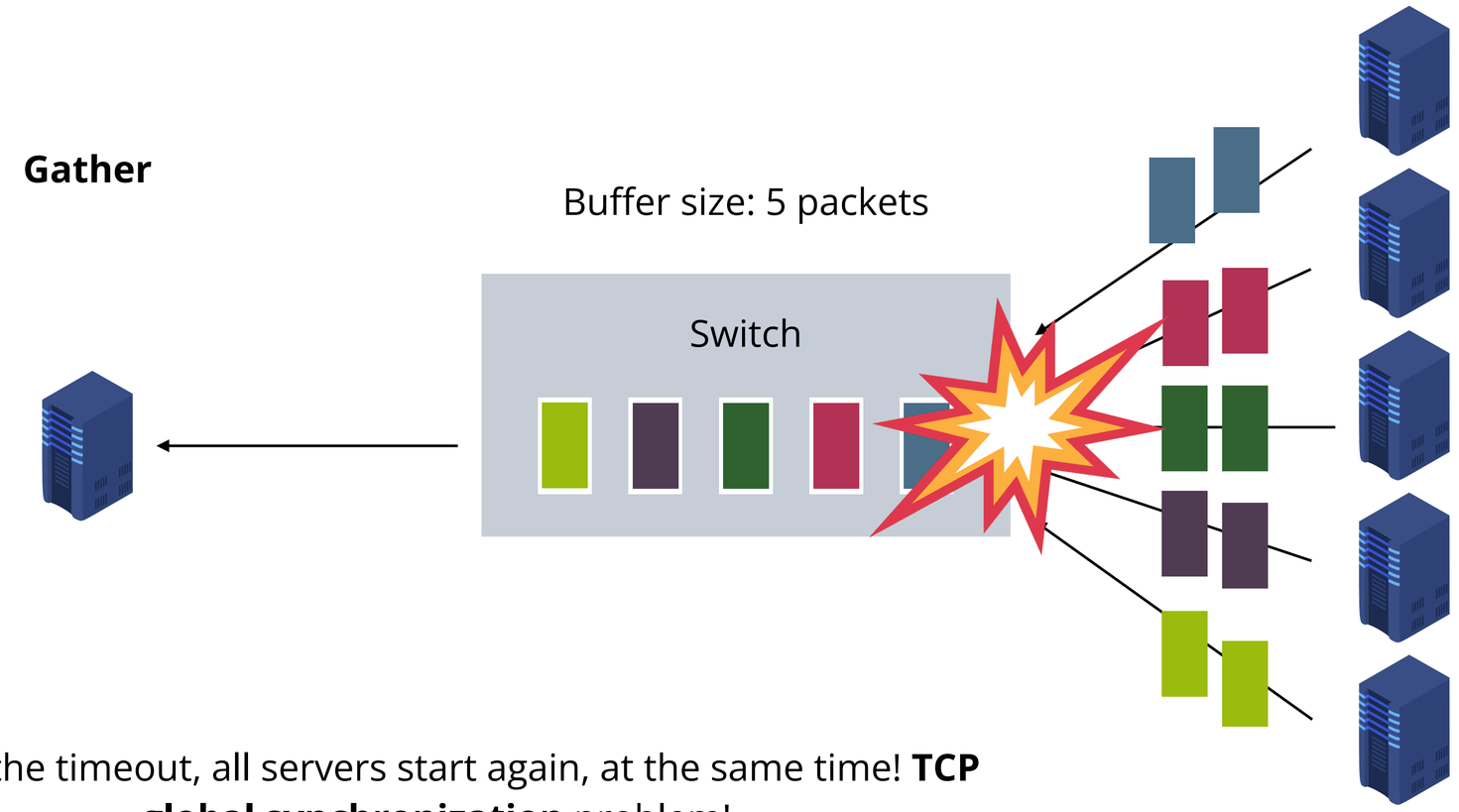


~100ms



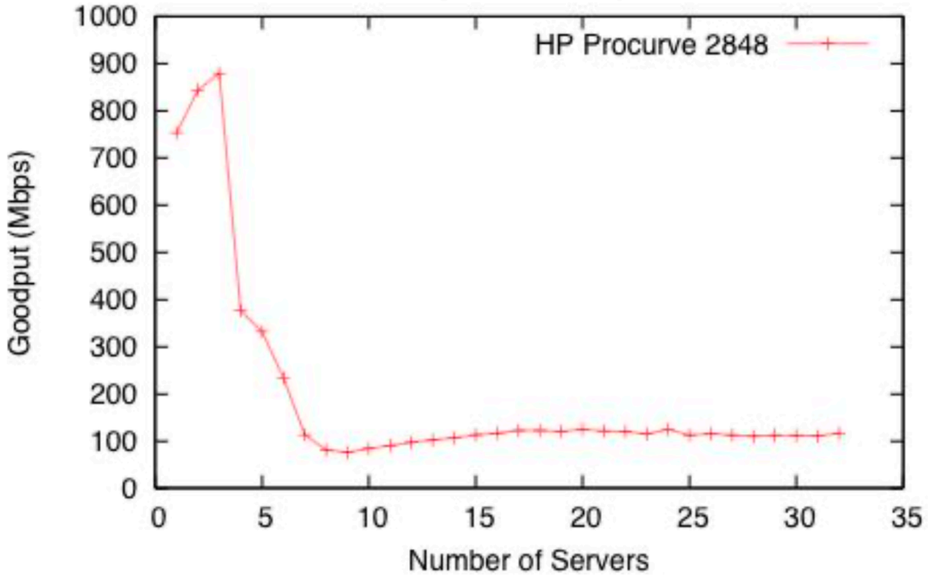
The collision leads to packet loss which will be recognized by the servers after a timeout.

From a switch point of view



After the timeout, all servers start again, at the same time! **TCP global synchronization** problem!

TCP incast problem



Measurement and Analysis of TCP Throughput Collapse in Cluster-based Storage Systems

Amar Phanishayee, Elie Krevat, Vijay Vasudevan,
David G. Andersen, Gregory R. Ganger, Garth A. Gibson, Srinivasan Seshan

Carnegie Mellon University

Abstract

Cluster-based and iSCSI-based storage systems rely on standard TCP/IP-over-Ethernet for client access to data. Unfortunately, when data is striped over multiple networked storage nodes, a client can experience a TCP throughput collapse that results in much lower read bandwidth than should be provided by the available network links. Conceptually, this problem arises because the client simultaneously reads fragments of a data block from multiple sources that together send enough data to overload the switch buffers on the client's link. This paper analyzes this *Incast* problem, explores its sensitivity to various system parameters, and examines the effectiveness of alterna-

client increases past the ability of an Ethernet switch to buffer packets. As we explore further in §2, the problem arises from a subtle interaction between limited Ethernet switch buffer sizes, the communication patterns common in cluster-based storage systems, and TCP's loss recovery mechanisms. Briefly put, data striping couples the behavior of multiple storage servers, so the system is limited by the request completion time of the *slowest* storage node [7]. Small Ethernet buffers are exhausted by a concurrent flood of traffic from many servers, which results in packet loss and one or more TCP timeouts. These timeouts impose a delay of hundreds of milliseconds–orders of magnitude greater than typical data fetch times–significantly degrading overall throughput.

USENIX FAST 2008

https://www.pdl.cmu.edu/PDL-FTP/Storage/FASTIncast.pdf

TCP incast

Packet drops due to the capacity overrun at shared commodify switches

- Can lead to **TCP global synchronization** and even more packet losses
- The link remains idle (hence, reduced capacity and poor performance)
- First discussed in Nagle et al., The Panasas ActiveScale Storage Cluster, SC 2004

Some potential solutions

- Use lower timeouts: (1) can lead to spurious timeouts and retransmissions, (2) high operating system overhead
- Other variants of TCP (SACKS, Cubic): cannot avoid the basic phenomenon of TCP incast
- Larger switch buffer: helps to push the collapse point further, but is expensive and introduces higher packet delay

Can we do better?

The basic challenge is that there are **only** limited number of things we can do once a packet is dropped

- Various acknowledgements schemes (ACK, SACK)
- Various timeouts based optimizations

Whatever clever way you come up with can be over-optimized

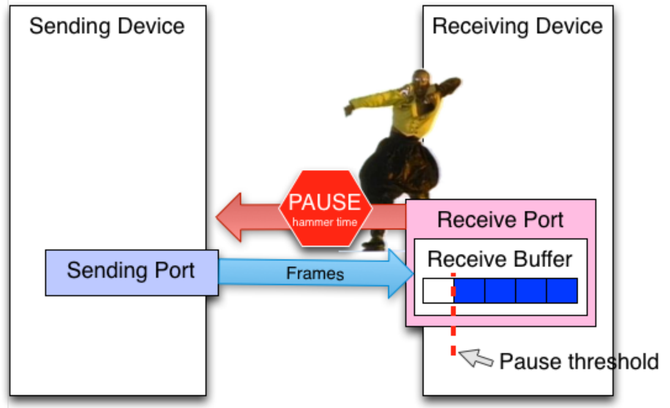
- Imagine deploying that with multiple workloads, flow patterns, and switches

Can we try to avoid packet drops in the first place? If so, how?

Ethernet flow control

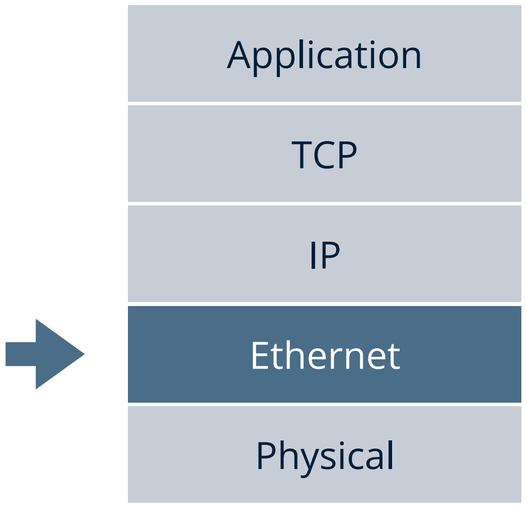
Pause Frame (IEEE 802.3x)

- An overwhelmed Ethernet receiver/NIC can send a "PAUSE" Ethernet frame to the sender
- Upon receiving the PAUSE frame, the sender stops transmission for a certain duration of time



Limitations

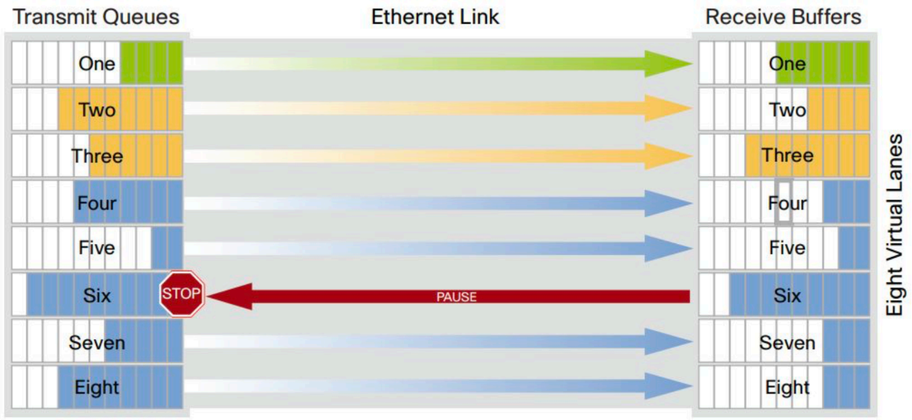
- Designed for end-host NIC (memory, queue) overruns, not switches
- Blocks all transmission at the Ethernet-level (port-level, not flow-level)



Priority-based flow control

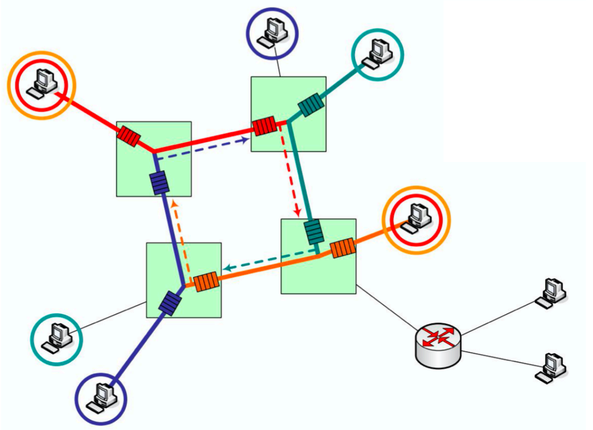
PFC, IEEE 802.1Qbb

- Enhancement over PAUSE frames
- 8 virtual traffic lanes and one can be selectively stopped
- Timeout is configurable



Limitations

- Only 8 lanes: think about the number of flows we may have
- Deadlocks in large networks
- Unfairness (victim flows)



https://www.cisco.com/c/dam/en/us/solutions/collateral/data-center-virtualization/ieee-802-1-data-center-bridging/at_a_glance_c45-460907.pdf

Data Center TCP (DCTCP)

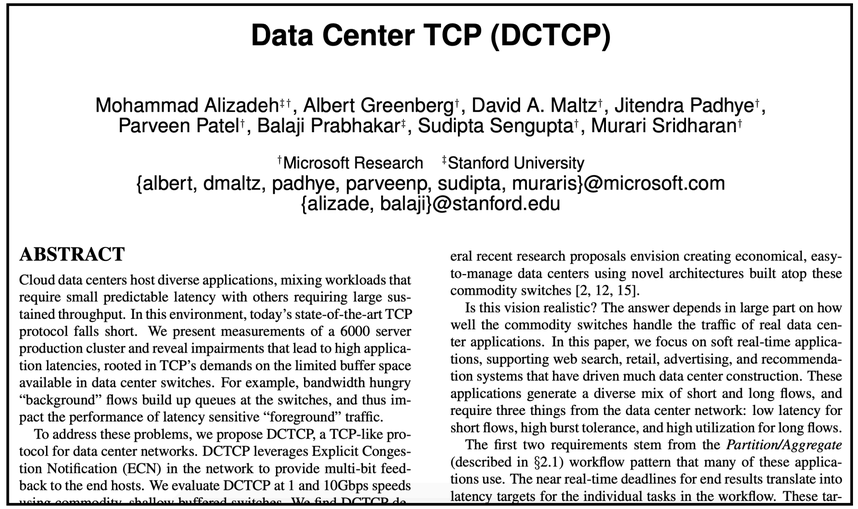
TCP-alike congestion control protocol

Basic idea: pass information about switch queue buildup to senders

- From **where** to pass information?
- **How** to pass information?

At the sender, react to this information by slowing down the transmission

- By **how much**?
- **How frequent**?



Data Center TCP (DCTCP)

Mohammad Alizadeh[‡†], Albert Greenberg[†], David A. Maltz[†], Jitendra Padhye[†],
Parveen Patel[†], Balaji Prabhakar[‡], Sudipta Sengupta[†], Murari Sridharan[†]

[†]Microsoft Research [‡]Stanford University
{albert, dmaltz, padhye, parveenp, sudipta, muraris}@microsoft.com
{alizade, balaji}@stanford.edu

ABSTRACT

Cloud data centers host diverse applications, mixing workloads that require small predictable latency with others requiring large sustained throughput. In this environment, today's state-of-the-art TCP protocol falls short. We present measurements of a 6000 server production cluster and reveal impairments that lead to high application latencies, rooted in TCP's demands on the limited buffer space available in data center switches. For example, bandwidth hungry "background" flows build up queues at the switches, and thus impact the performance of latency sensitive "foreground" traffic.

To address these problems, we propose DCTCP, a TCP-like protocol for data center networks. DCTCP leverages Explicit Congestion Notification (ECN) in the network to provide multi-bit feedback to the end hosts. We evaluate DCTCP at 1 and 10Gbps speeds

eral recent research proposals envision creating economical, easy-to-manage data centers using novel architectures built atop these commodity switches [2, 12, 15].

Is this vision realistic? The answer depends in large part on how well the commodity switches handle the traffic of real data center applications. In this paper, we focus on soft real-time applications, supporting web search, retail, advertising, and recommendation systems that have driven much data center construction. These applications generate a diverse mix of short and long flows, and require three things from the data center network: low latency for short flows, high burst tolerance, and high utilization for long flows.

The first two requirements stem from the *Partition/Aggregate* (described in §2.1) workflow pattern that many of these applications use. The near real-time deadlines for end results translate into latency targets for the individual tasks in the workflow. These tar-

ACM SIGCOMM 2010

Explicit Congestion Notification (ECN)

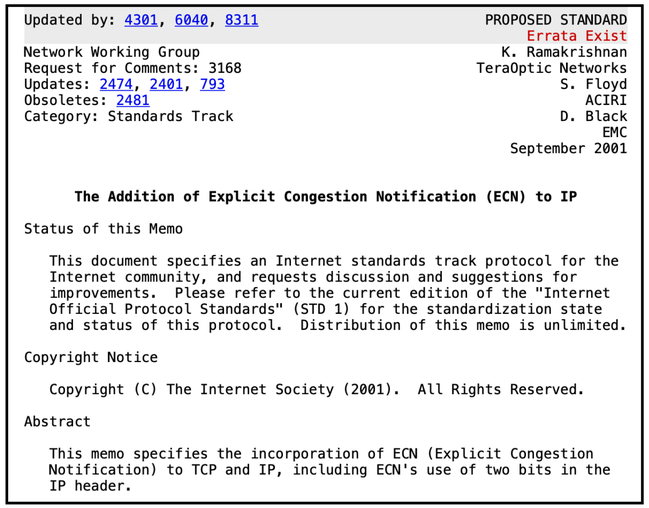
RFC 3168

ECN is a standardized way of passing "the presence of congestion"

- Part of the IP packet header (2 bits): uses 1 bit for capability/ ACK and 1 bit for congestion indication (yes/no)
- Supported by most commodity switches

Idea: For a queue size of N, when the queue occupancy goes beyond K, mark the passing packet's ECN bit as "yes"

- There are more sophisticated logics (Random Early Detection, RED) that can probabilistically mark packets



Updated by: 4301, 6040, 8311 — PROPOSED STANDARD
Errata Exist

Network Working Group — K. Ramakrishnan
Request for Comments: 3168 — TeraOptic Networks
Updates: 2474, 2401, 793 — S. Floyd
Obsoletes: 2481 — ACIRI
Category: Standards Track — D. Black
EMC
September 2001

The Addition of Explicit Congestion Notification (ECN) to IP

Status of this Memo

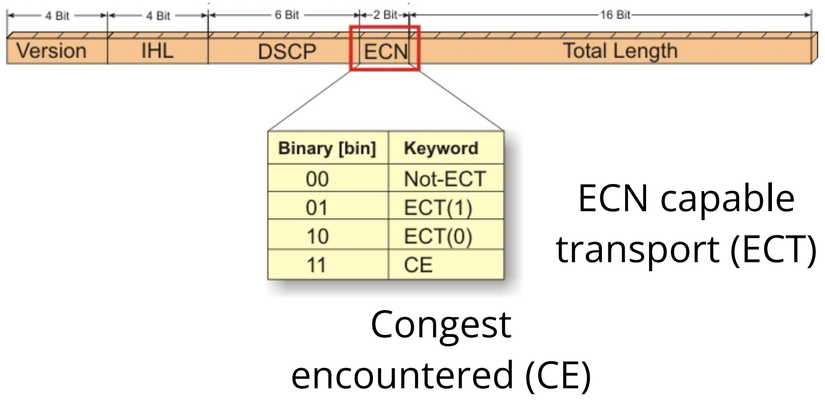
This document specifies an Internet standards track protocol for the Internet community, and requests discussion and suggestions for improvements. Please refer to the current edition of the "Internet Official Protocol Standards" (STD 1) for the standardization state and status of this protocol. Distribution of this memo is unlimited.

Copyright Notice

Copyright (C) The Internet Society (2001). All Rights Reserved.

Abstract

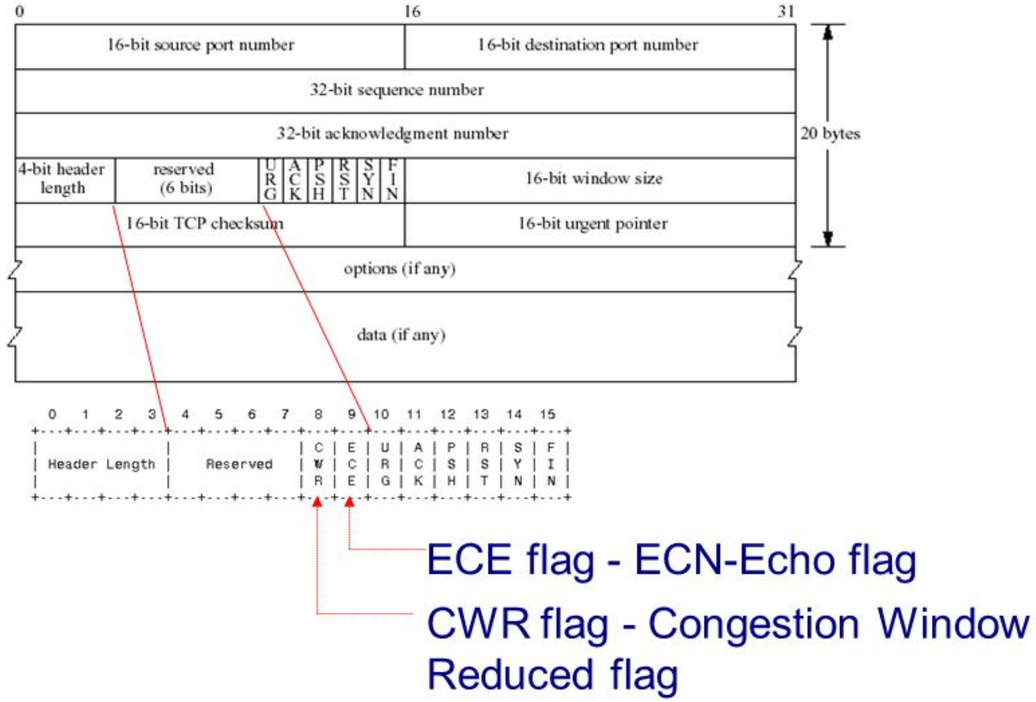
This memo specifies the incorporation of ECN (Explicit Congestion Notification) to TCP and IP, including ECN's use of two bits in the IP header.



ECN capable transport (ECT)

Congest encountered (CE)

The ECN bits location in TCP header



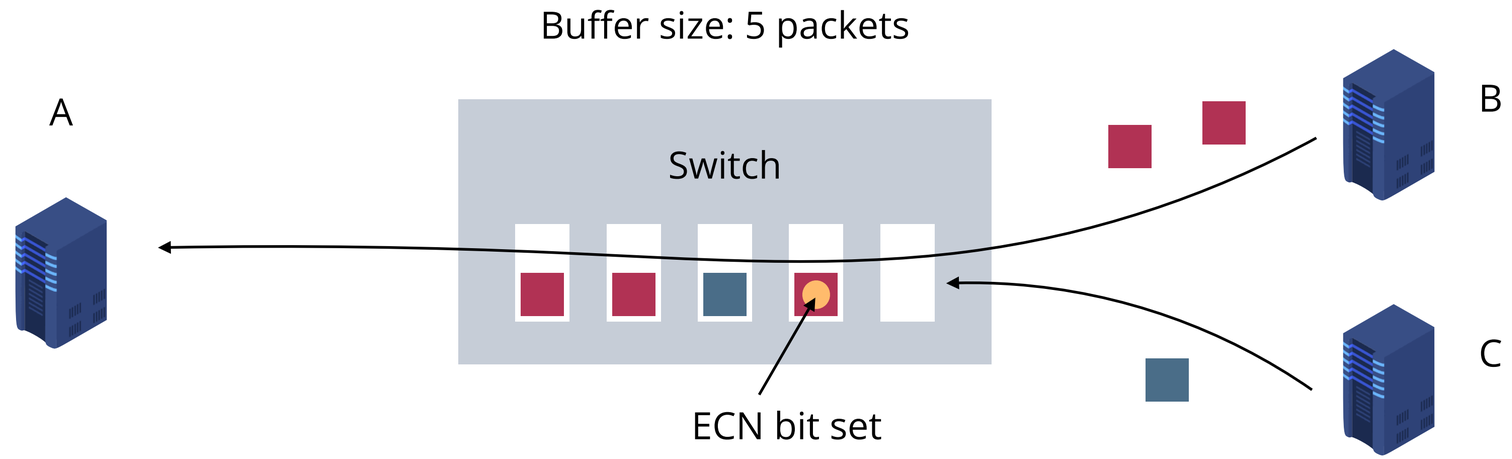
The TCP congestion window logic:

- Additive increase: W → W + 1 per RTT
- Multiplicative decrease: W → W / 2
  - Packet loss
  - A packet received with ECN marked

https://en.wikipedia.org/wiki/Explicit_Congestion_Notification

ECN bit in action

Assume that B is sending TCP data packets to A.
At some point of time, C also starts to send packets, and the queue is getting full.
The switch starts to mark packets with ECN bits.



How does B get to know there was a congestion at the switch?

DCTCP main idea

Simple marking at the switch

- After threshold K start marking packets with ECN (instantaneous vs. average marking)
- Uses instantaneous marking for fast notification

Typical ECN receiver

- Mark ACKs with the ECE flag, until the sender ACKs back using CWR flag bit

DCTCP receiver

- Only mark ACKs corresponding to the ECN packet

Sender’s congestion control

- Estimate the packets that are marked with ECN in a running window

DCTCP congestion window calculations

In every RTT, calculate the percentage of ECN-marked ACKs

$$F = \frac{\#\text{ECN-marked ACKs}}{\#\text{Total ACKs}}$$

Use a sliding window to estimate the average percentage of ECN-marked ACKs

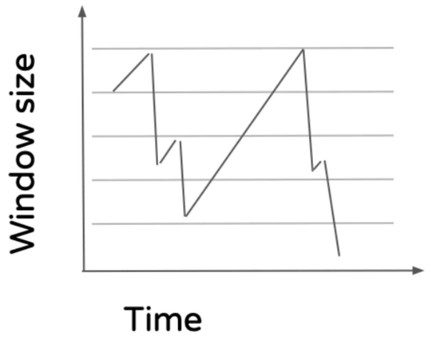
$$\alpha \leftarrow (1 - g) \times \alpha + g \times F$$

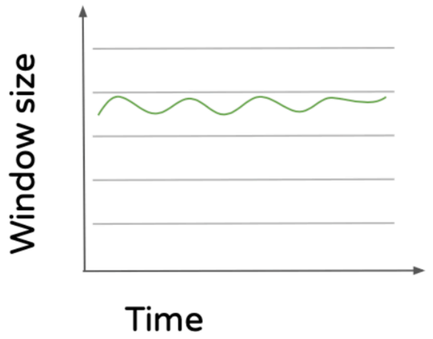
Decide the congestion window based on the estimation

$$cwnd \leftarrow cwnd \times (1 - \alpha/2)$$

DCTCP vs TCP example

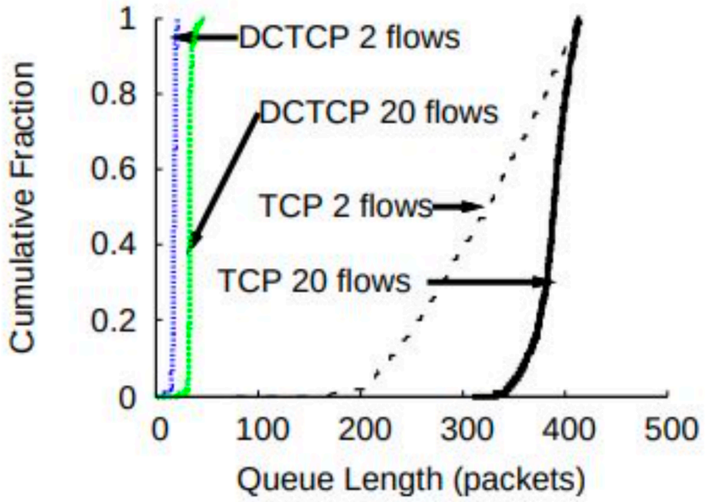
ECN-marks on ACKs	TCP	MPTCP
0**11**000**1**00**1**	Cut window by 50% (every time)	Cut window by 40%
0000000000**1**	Cut window by 50%	Cut window by 10%



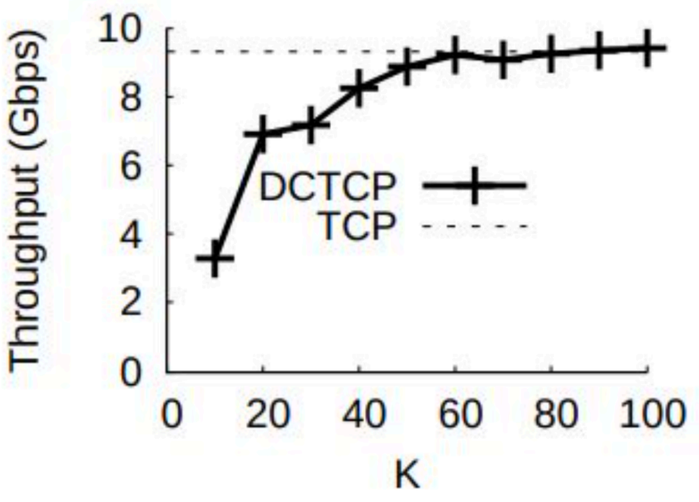


DCTCP performance

1Gbps, same bandwidth but lower switch queue occupancy

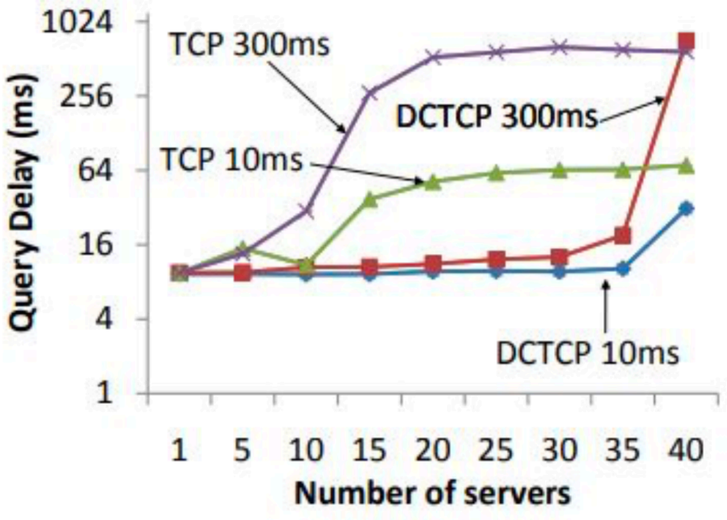


At 10Gbps, after certain K (queue occupancy parameter) threshold, the same bandwidth

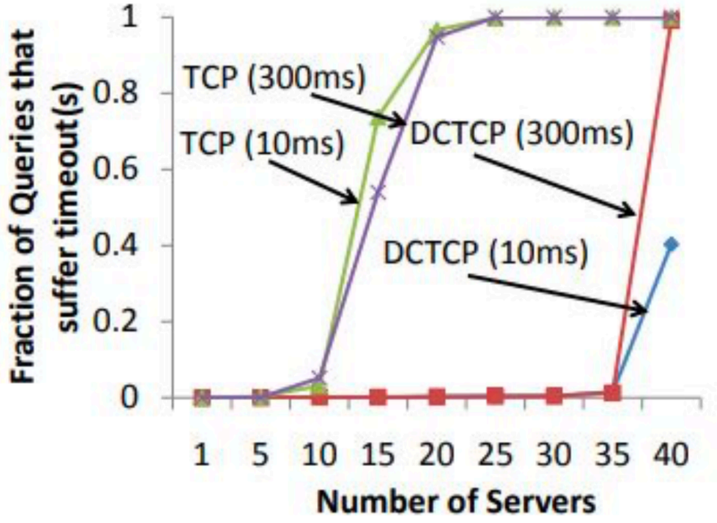


What about incast?

Query: 1 machine sending 1MB/n data to n machines and waiting for the echo



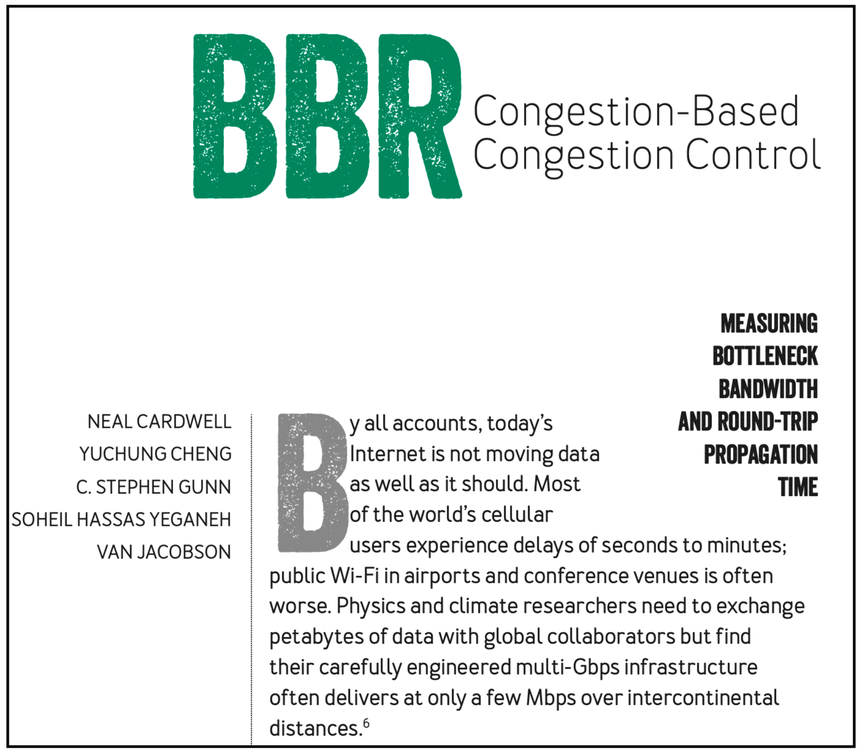
Better performance than TCP up to a point where (#servers=35) where not even a single packet can pass from the switch



DCTCP has very low packet losses in comparison to TCP

Can we use a different congestion signal than the queue occupancy on the switch?

Recall BBR



BBR Congestion-Based Congestion Control

NEAL CARDWELL
YUCHUNG CHENG
C. STEPHEN GUNN
SOHEIL HASSAS YEGANEH
VAN JACOBSON

MEASURING BOTTLENECK BANDWIDTH AND ROUND-TRIP PROPAGATION TIME

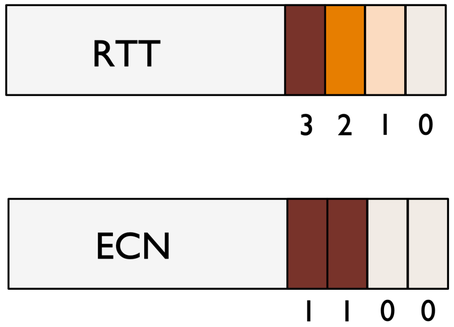
By all accounts, today's Internet is not moving data as well as it should. Most of the world's cellular users experience delays of seconds to minutes; public Wi-Fi in airports and conference venues is often worse. Physics and climate researchers need to exchange petabytes of data with global collaborators but find their carefully engineered multi-Gbps infrastructure often delivers at only a few Mbps over intercontinental distances.[6]

Can we directly apply BBR on data center networks?

TIMELY

Use Round Trip Time (RTT) as the indication of congestion signal

- RTT is a multi-bit signal indicating end-to-end congestion throughout the network — no explicit switch support required to do any marking
- RTT covers ECN signal completely, but not vice versa!



TIMELY: RTT-based Congestion Control for the Datacenter

Radhika Mittal*(UC Berkeley), Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi*(Microsoft), Amin Vahdat, Yaogong Wang, David Wetherall, David Zats

Google, Inc.

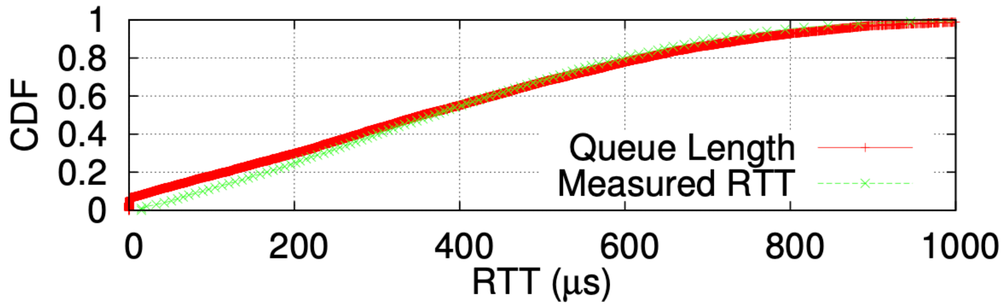
ABSTRACT

Datacenter transports aim to deliver low latency messaging together with high throughput. We show that simple packet delay, measured as round-trip times at hosts, is an effective congestion signal without the need for switch feedback. First, we show that advances in NIC hardware have made RTT measurement possible with microsecond accuracy, and that these RTTs are sufficient to estimate switch queueing. Then we describe how TIMELY can adjust transmission rates using RTT gradients to keep packet latency low while delivering high bandwidth. We implement our design

1. INTRODUCTION

Datacenter networks run tightly-coupled computing tasks that must be responsive to users, e.g., thousands of back-end computers may exchange information to serve a user request, and all of the transfers must complete quickly enough to let the complete response to be satisfied within 100 ms [24]. To meet these requirements, datacenter transports must simultaneously deliver high bandwidth (≫Gbps) and utilization at low latency (≪msec), even though these aspects of performance are at odds. Consistently low latency matters because even a small fraction of late operations

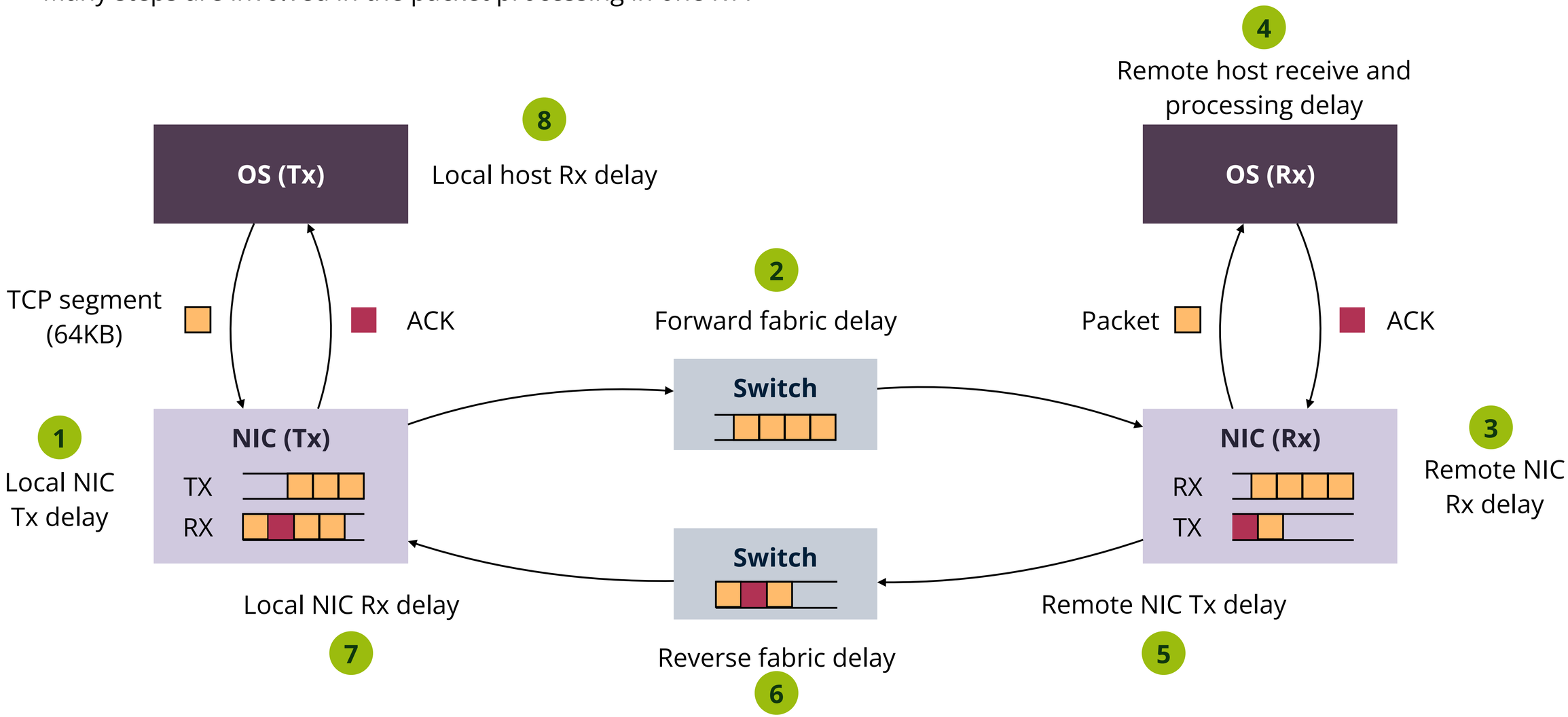
ACM SIGCOMM 2015



However, getting precise RTT is challenging. Why?

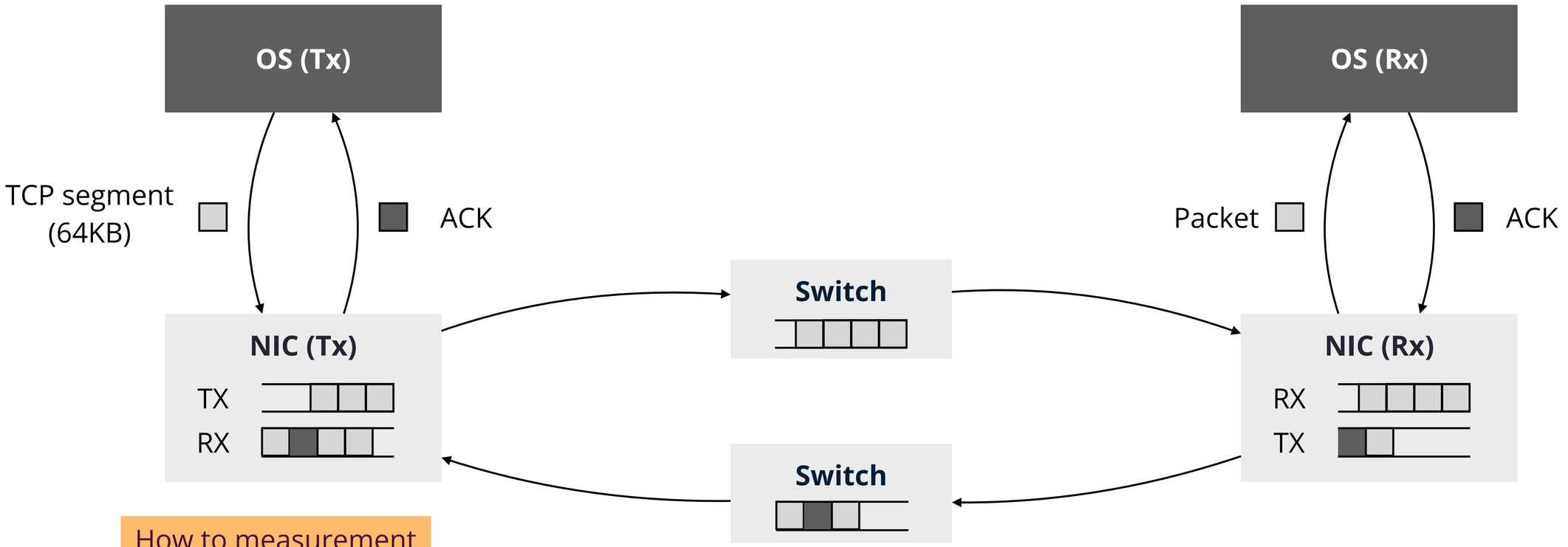
RTT calculation challenges

Many steps are involved in the packet processing in one RTT



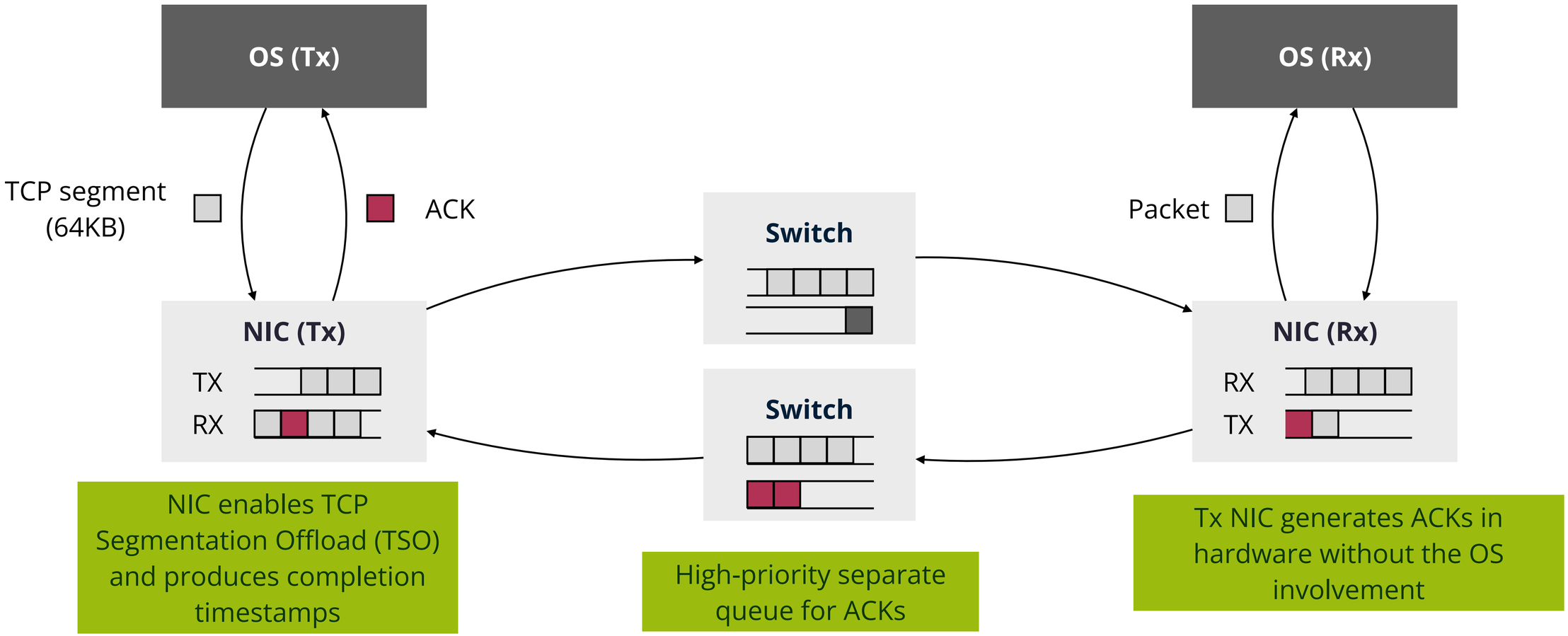
RTT calculation challenges

Many steps are involved in the packet processing in one RTT



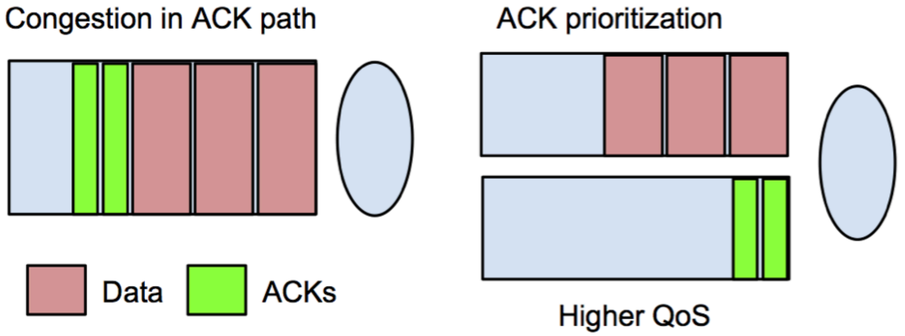
RTT calculation challenges

Many steps are involved in the packet processing in one RTT

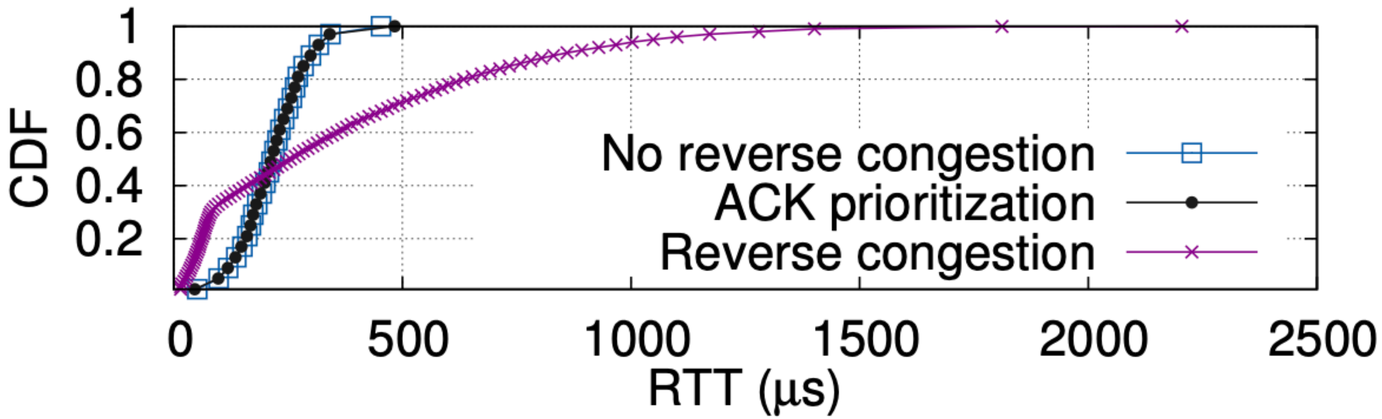


Separate ACK queuing to solve reverse congestion

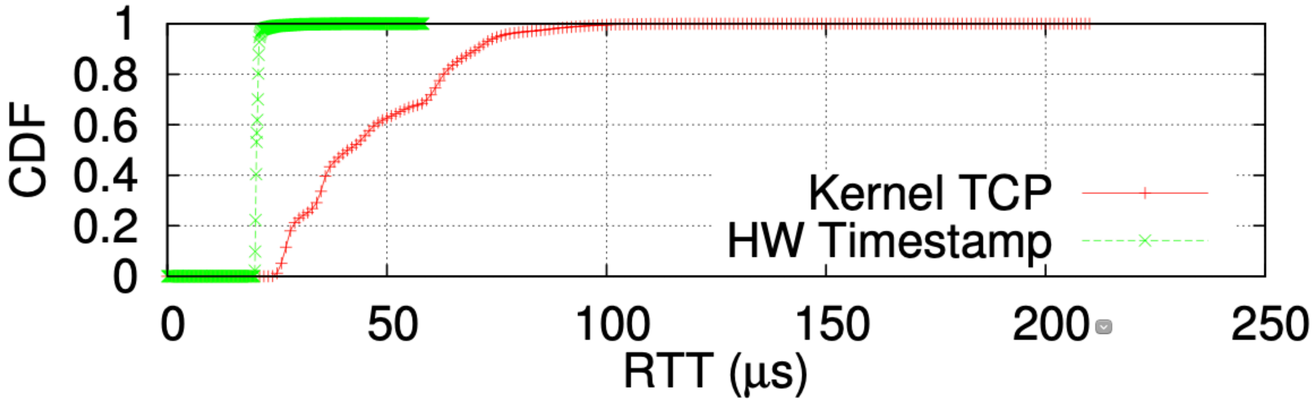
One queue for data and ACKs



Separate queues for data and ACKs, ACKs are prioritized

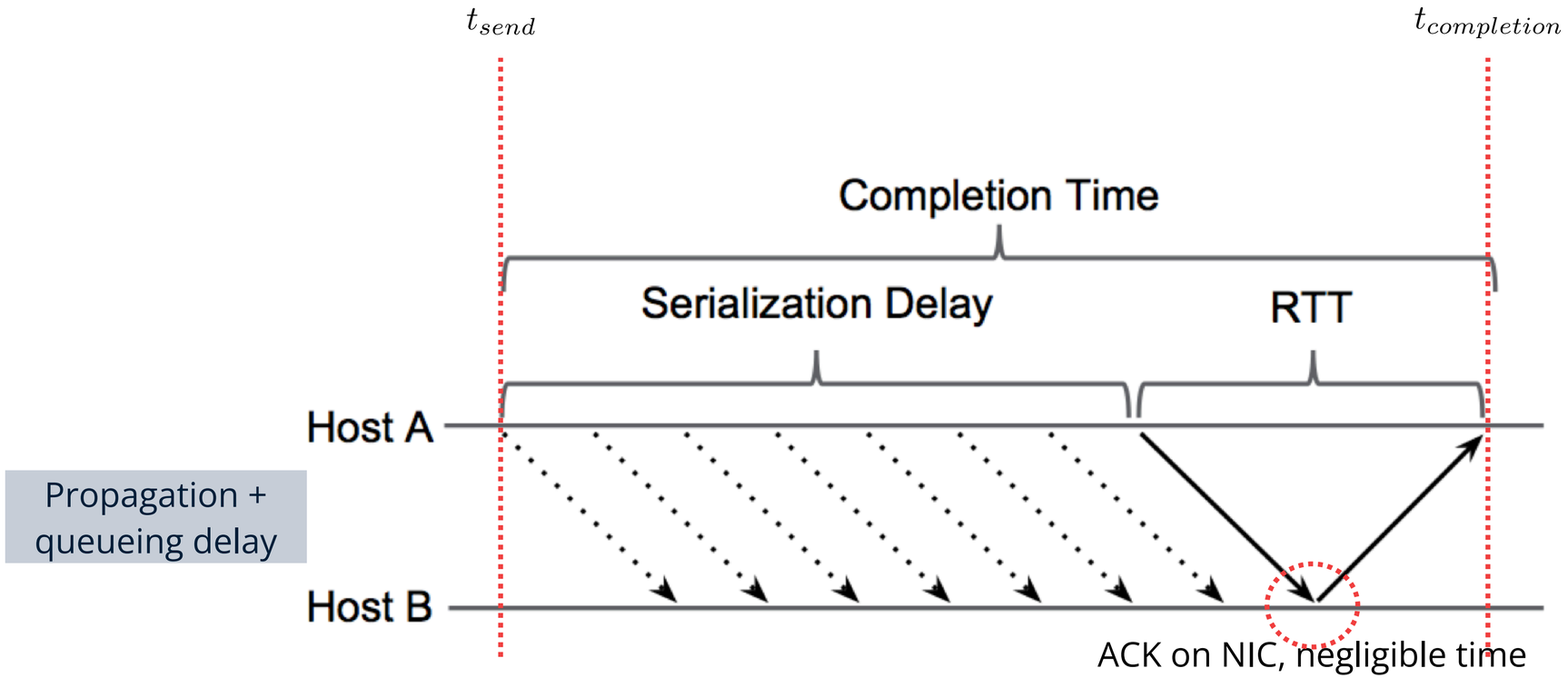


Can we measure RTTs precisely?



Yes, the random variance is much smaller than the kernel TCP measurements

RTT calculation



$$RTT = t_{completion} - t_{send} - seg.size/NIC.linerate$$

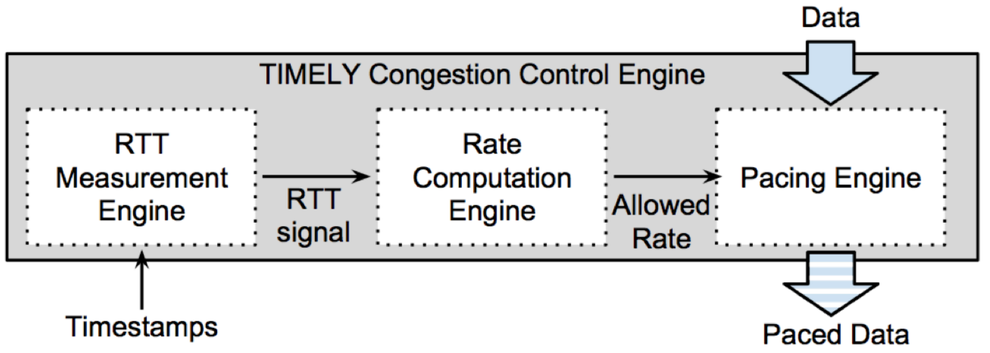
TIMELY

Independent of the transport used

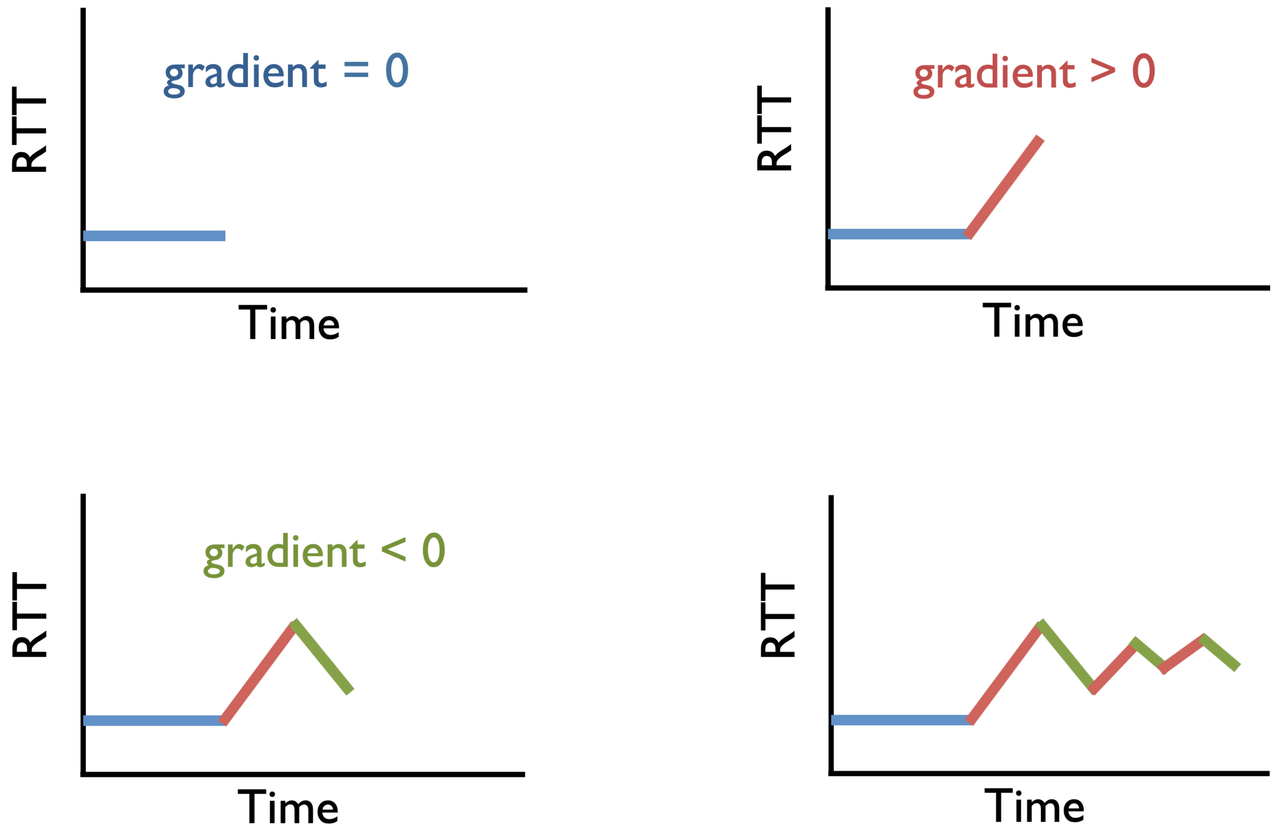
- Assumes an ACK-based protocol (TCP)
- Receivers must generate ACKs for incoming data

Key concept

- Absolute RTTs are not used, only the **gradient** of the RTTs
- Positive gradient → rising RTT → queue buildup
- Negative gradient → decreasing RTT → queue depletion



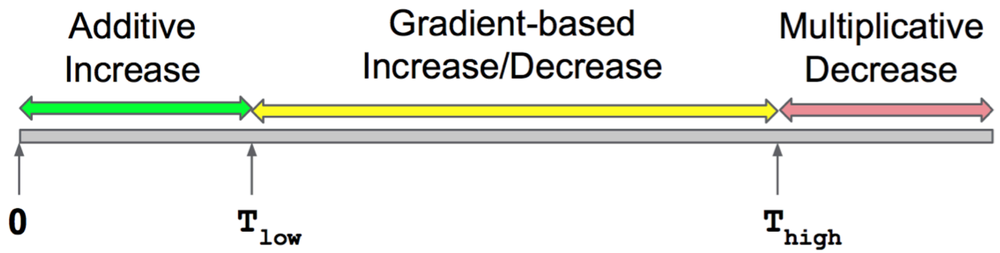
RTT gradient



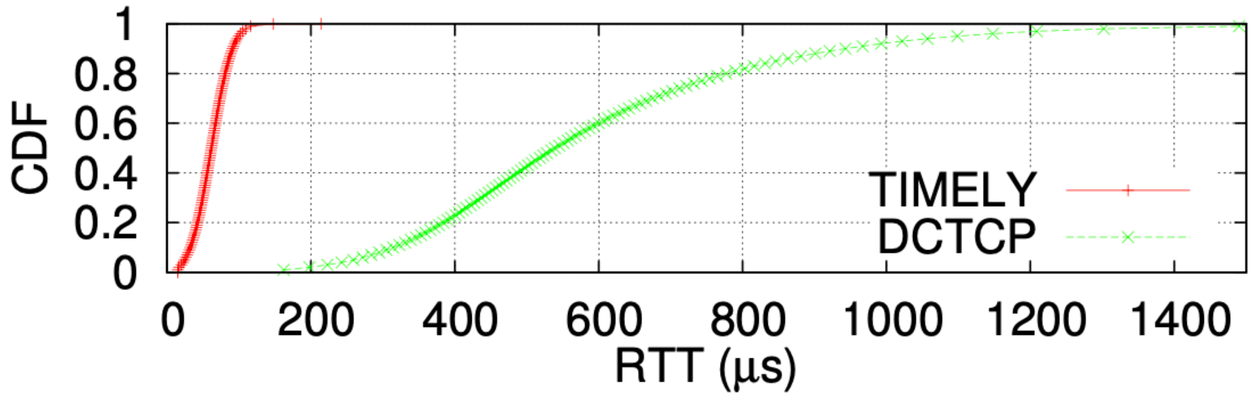
TIMELY pacing engine

Algorithm 1: TIMELY congestion control.

Data: new_rtt
Result: Enforced rate
new_rtt_diff = new_rtt - prev_rtt ;
prev_rtt = new_rtt ;
rtt_diff = $(1 - \alpha) \cdot$ rtt_diff + $\alpha \cdot$ new_rtt_diff ;
▷ α: EWMA weight parameter
normalized_gradient = rtt_diff / minRTT ;
if *new_rtt* < T_{low} **then**
  rate ← rate + δ ;
  ▷ δ: additive increment step
  return;
if *new_rtt* > T_{high} **then**
  rate ← rate $\cdot \left(1 - \beta \cdot \left(1 - \frac{T_{high}}{new\_rtt}\right)\right)$;
  ▷ β: multiplicative decrement factor
  return;
if *normalized_gradient* ≤ 0 **then**
  rate ← rate + N $\cdot \delta$;
  ▷ N = 5 if gradient<0 for five completion events (HAI mode); otherwise N = 1
else
  rate ← rate $\cdot$ (1 - $\beta \cdot$ normalized_gradient)



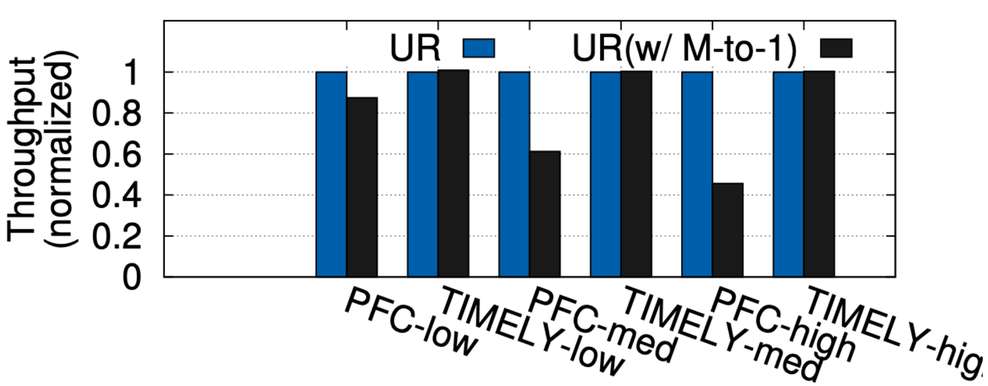
TIMELY performance: vs. DCTCP

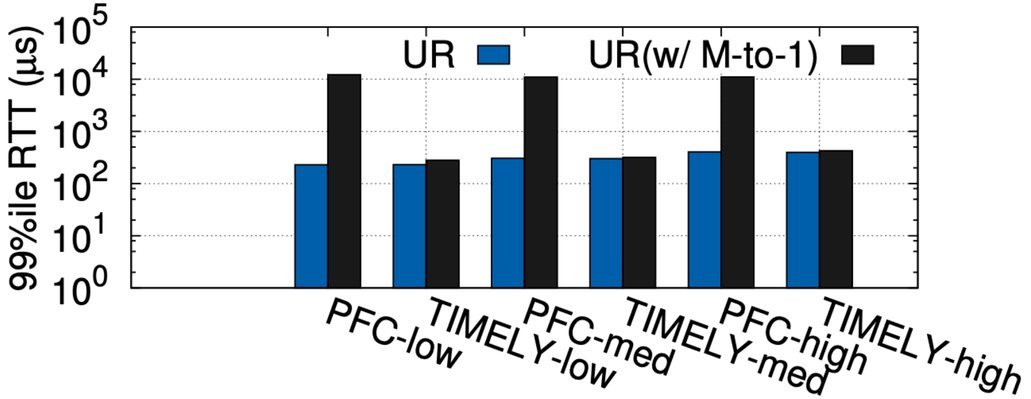


TIMELY achieves much lower yet stable RTTs

TIMELY performance: incast

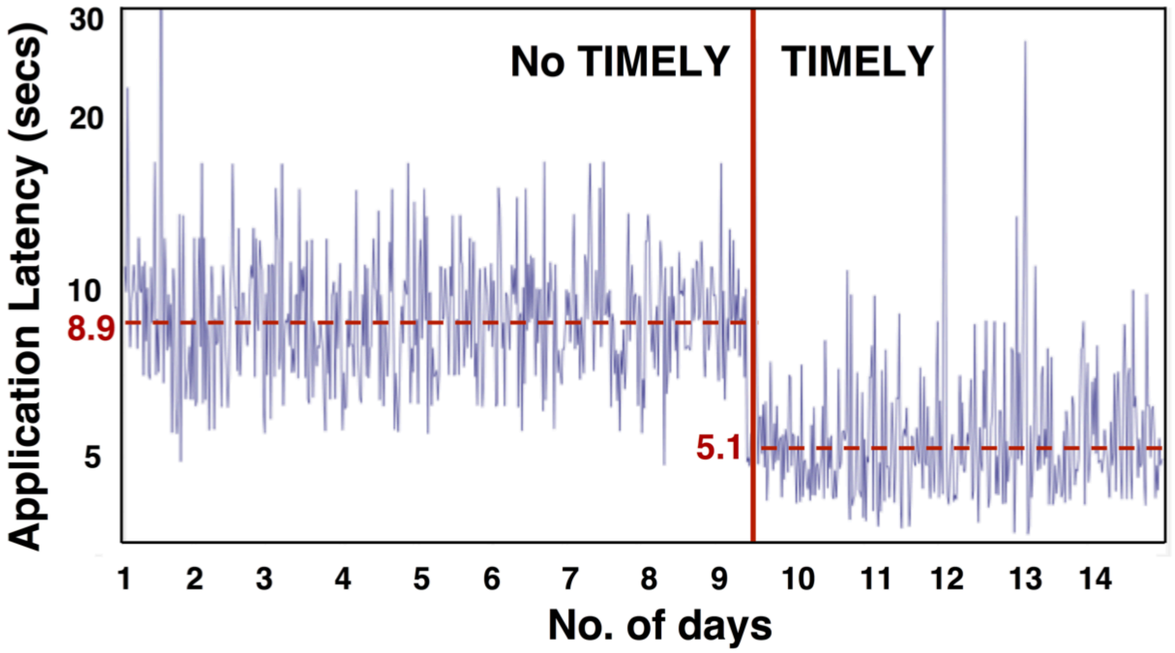
UR: Uniform Random





TIMELY throughput and latency stay the same
with and without incast traffic

TIMEly performance: application level



A (unknown) RPC latency of a data center storage benchmark

TIMELY issues

Gradient-based pacing

- **Complex** to tune the parameters

RTT measurement

- Only the fabric-related delay calculated with timestamps on NICs
- Does not differentiate between **fabric congestion** and **end-host congestion**

Extreme incast

- What happens if the **number of flows** is **larger** than the **path BDP**?
- Even one packet per flow would overrun the network

Swift

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

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

ACM SIGCOMM 2020

Swift designs

Simple target delay window control

Separating fabric and host congestion

Fractional congestion window to handle large-scale incast

Simple target delay window control

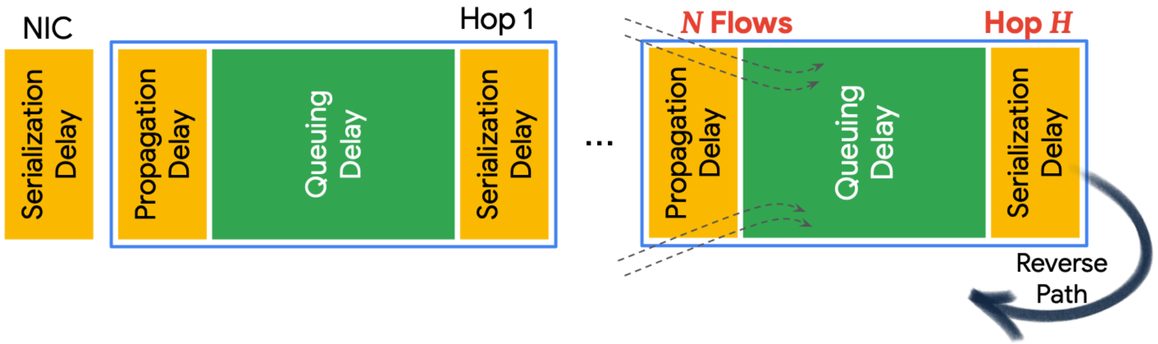
```
3  bool can_decrease ←                          ▷ Enforces MD once every RTT
     (now − t_last_decrease ≥ rtt)
4  On Receiving ACK
5    retransmit_cnt ← 0
6    target_delay ← TargetDelay()                ▷ See S3.5
7    if delay < target_delay then                ▷ Additive Increase (AI)
8      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
```

Cumulative increase over an RTT is equal to ai

Constrained to be max. once per RTT

Proportional to the extent of congestion

Deciding target delay



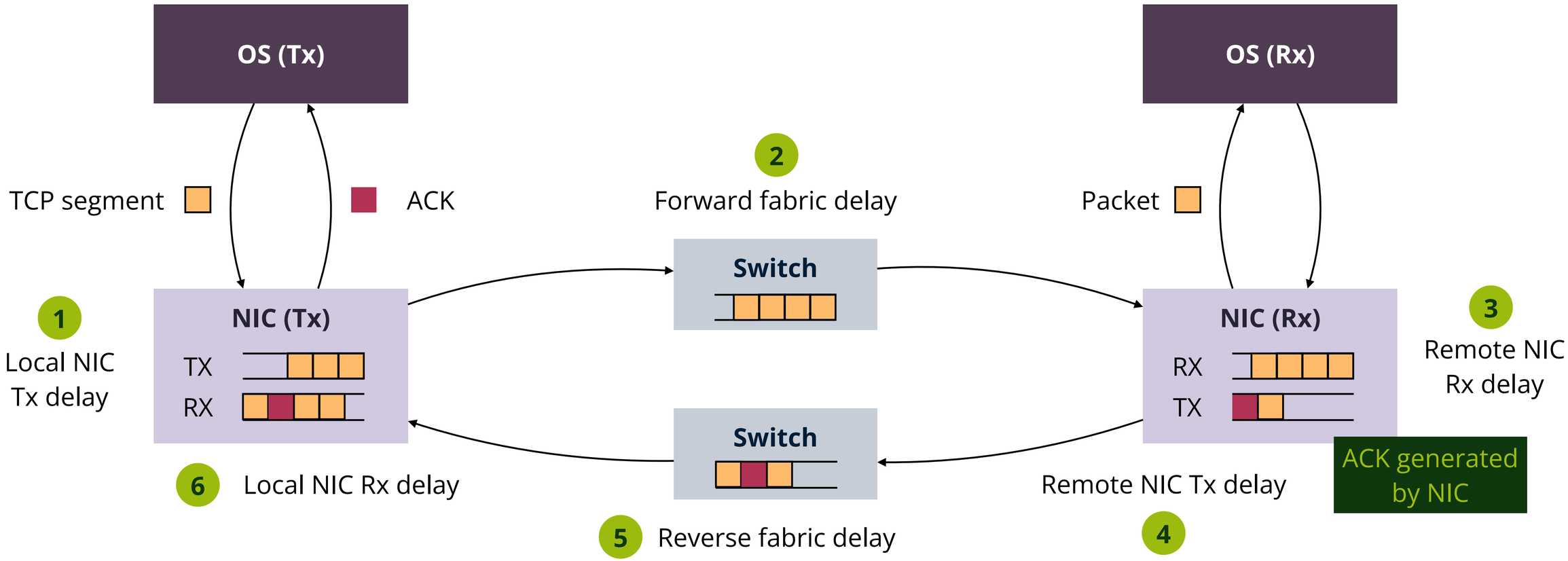
Topology-based scaling:

- Fixed base delay plus a fixed per-hop delay
- Forward path hop count measured with IP TTL and reflected back with ACKs

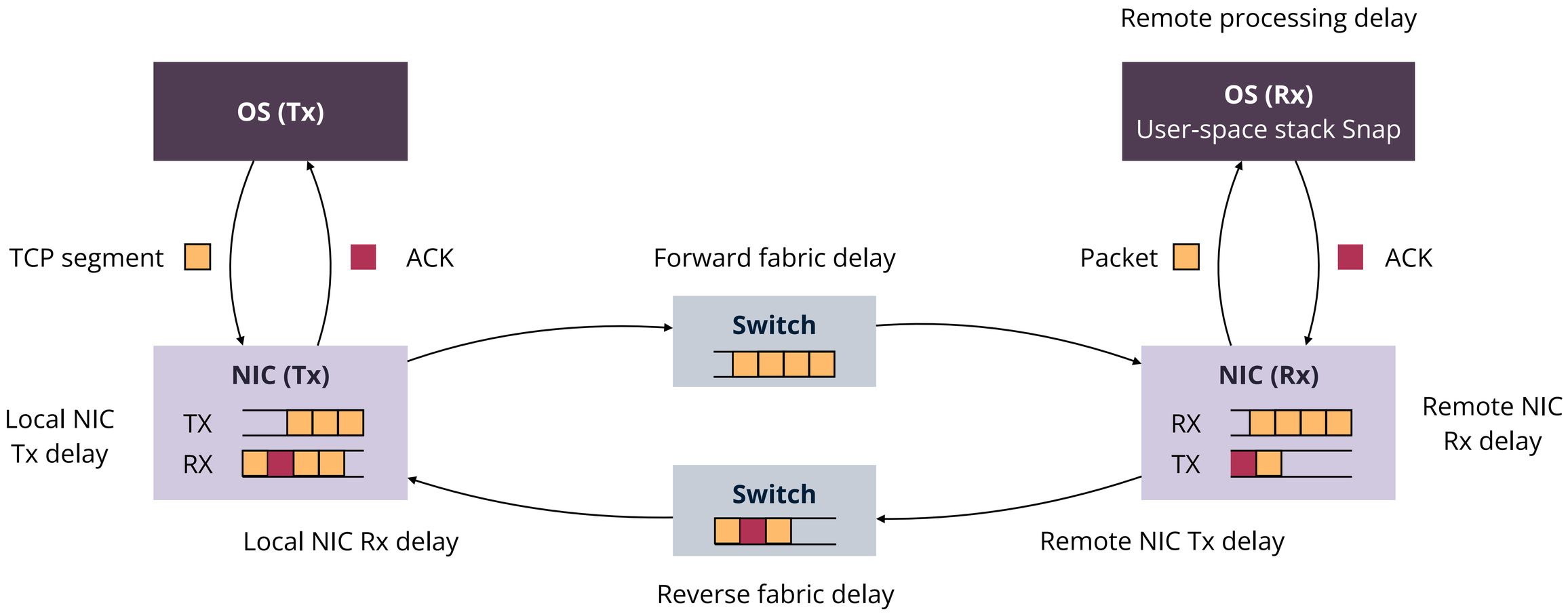
Flow-based scaling:

- Target delay increases with the number of competing flows
- Average queue length grows as $O(N)$
- Adjust the target in proportion to $1/\sqrt{cwnd}$

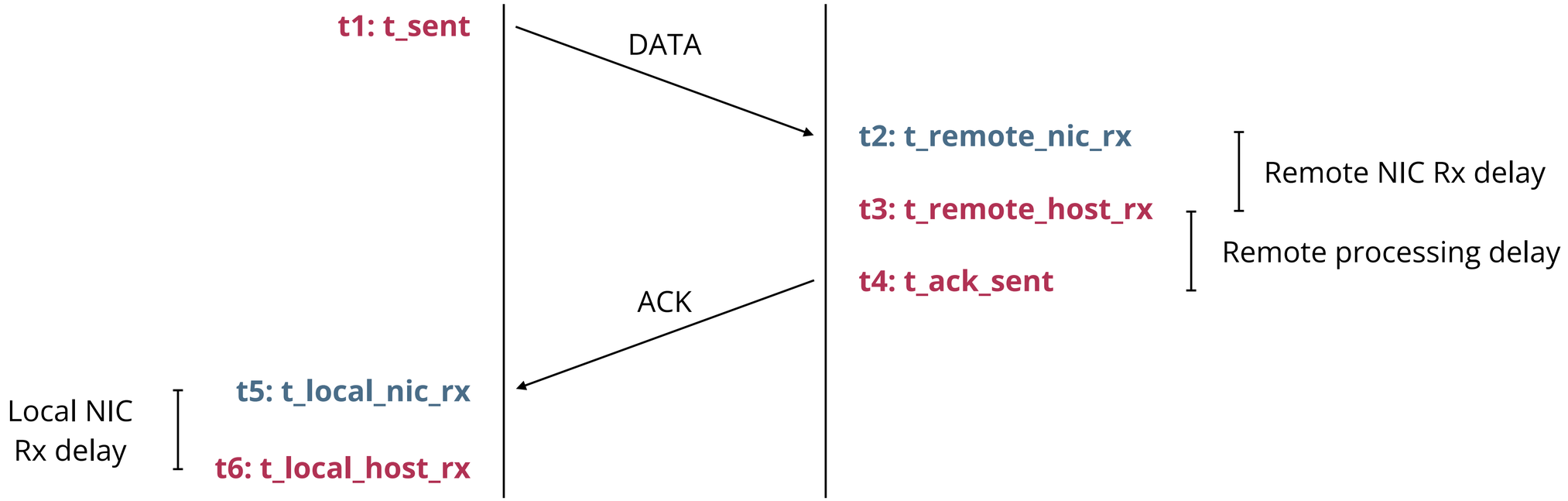
Delay measurement in TIMELY



Fabric vs. host congestion

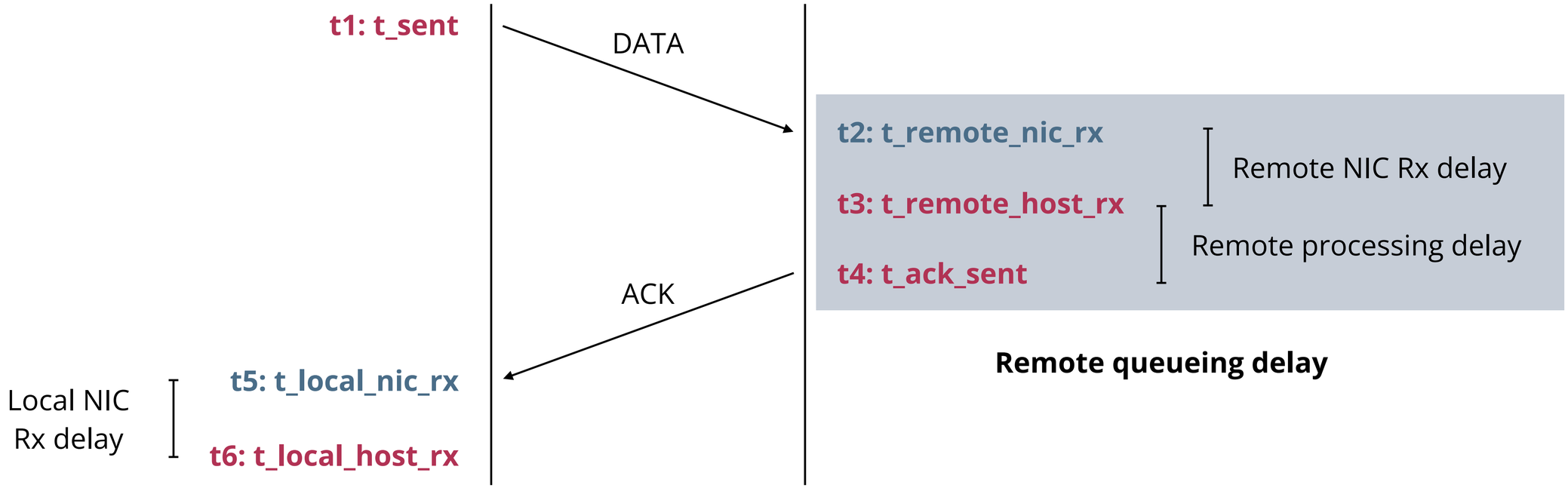


Timestamps measurement



Which part represents the remote queueing delay?

Timestamps measurement



Endpoint congestion control

Endpoint delay: (t4 - t2) + (t6 - t5)

Fabric delay: RTT - endpoint_delay

```
3  bool can_decrease ←                          ▷ Enforces MD once every RTT
     (now − t_last_decrease ≥ rtt)
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
```

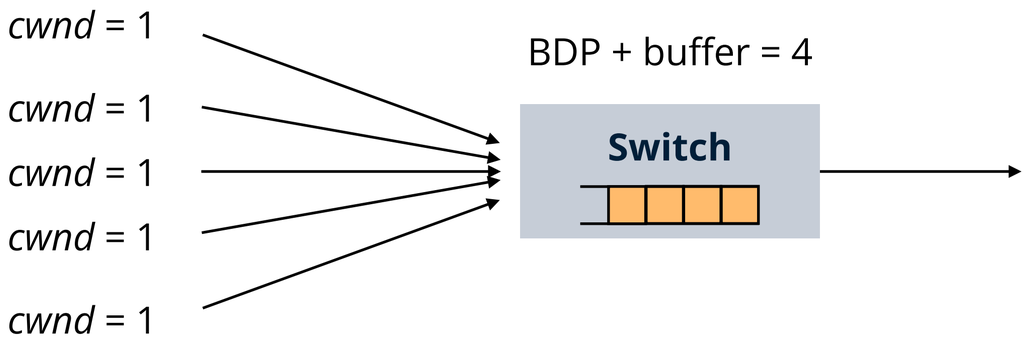
Follow the same target delay window control mechanism to decide *ecwnd*

Target delay for endpoints is decided based on Exponential Weighted Moving Average (EWMA) to remove noise

Actual cwnd = min (*fcwnd*, *ecwnd*)

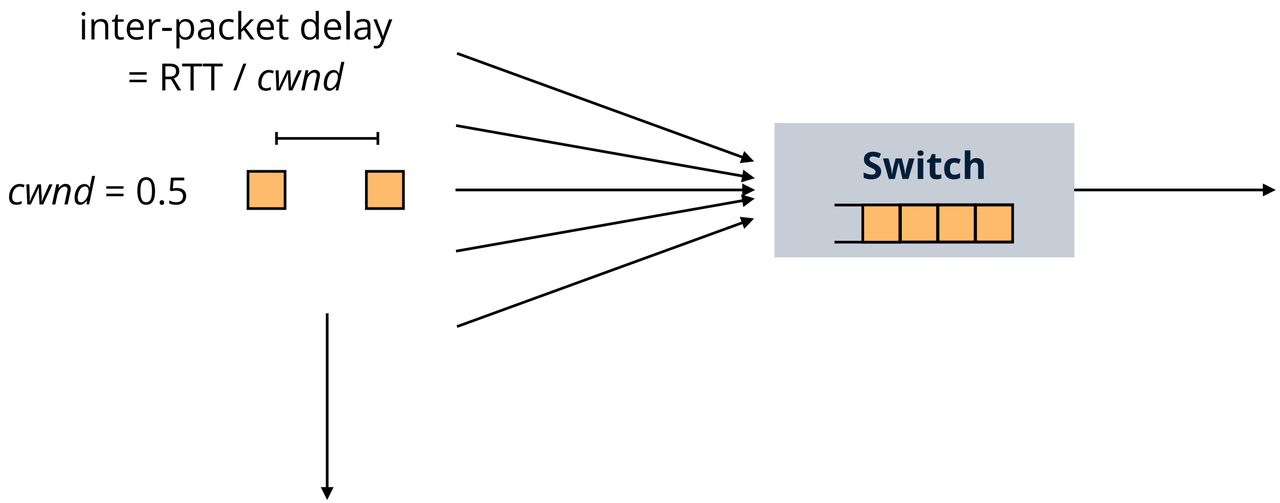
fcwnd: fabric congestion window based on the fabric delay
ecwnd: endpoint congestion window based on the endpoint delay

Handling large-scale incast



Even a **congestion window of one** is not able to prevent the incast congestion

Fractional window size smaller than one

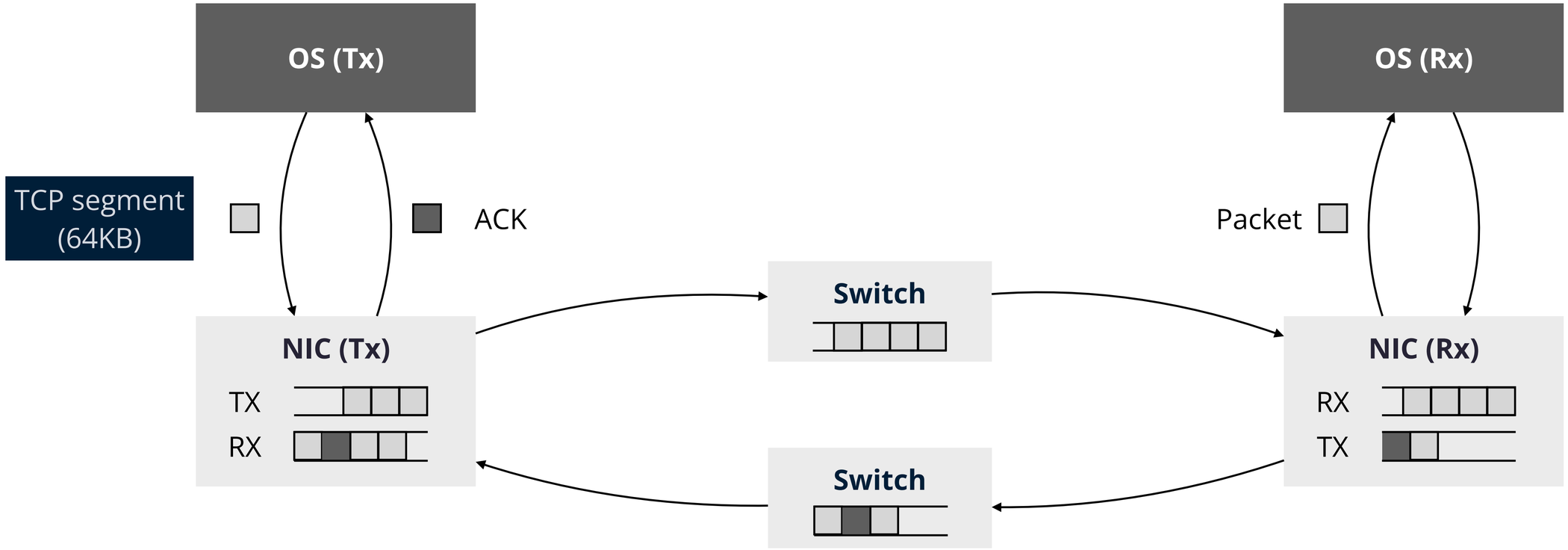


Pacing implemented with Timing Wheel [1] → High CPU overhead for cases with *cwnd* > 1 → Use **ACK-clocking** in such cases

Why not a problem in TIMELY?

[1] Ahmed Saeed, Nandita Dukkipati, Vytautas Valancius, Vinh The Lam, Carlo Contavalli, Amin Vahdat. Carousel: Scalable Traffic Shaping at End Hosts. ACM SIGCOMM 2017.

Packing overhead in TIMELY

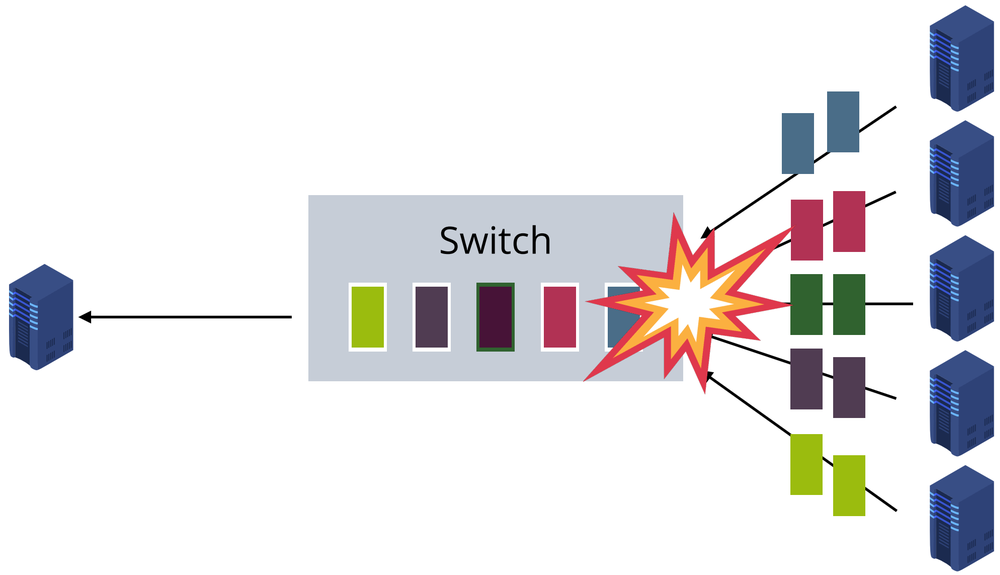


Packing is done in chunks of 64KB, instead of MTU-size level in Swift

Summary

Congestion control challenges in data centers

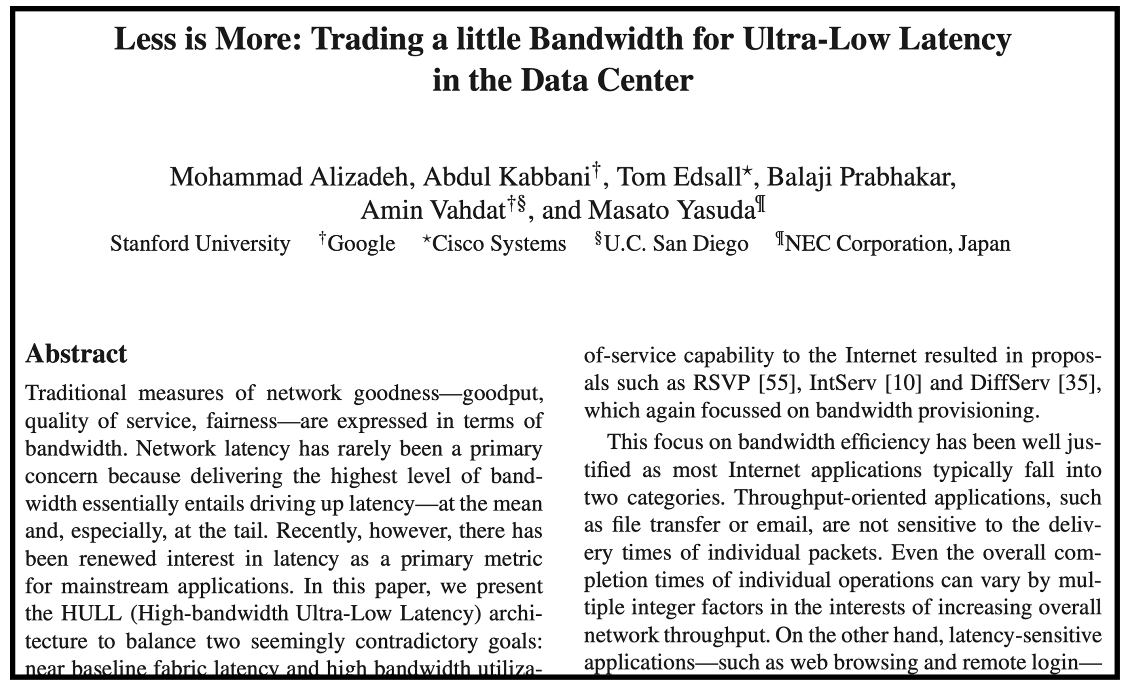
- Network has low latency and high throughput
- Applications are diverse with different requirements
- Incast congestion



Transport in data centers

- PFC has limited capability
- DCTCP: ECN-based congestion control
- TIMELY: RTT-based congestion control
- Swift: simpler window control logic, handling endpoint congestion, and dealing with large-scale incase

Other reading material



Less is More: Trading a little Bandwidth for Ultra-Low Latency in the Data Center

Mohammad Alizadeh, Abdul Kabbani†, Tom Edsall*, Balaji Prabhakar, Amin Vahdat†§, and Masato Yasuda¶

Stanford University †Google *Cisco Systems §U.C. San Diego ¶NEC Corporation, Japan

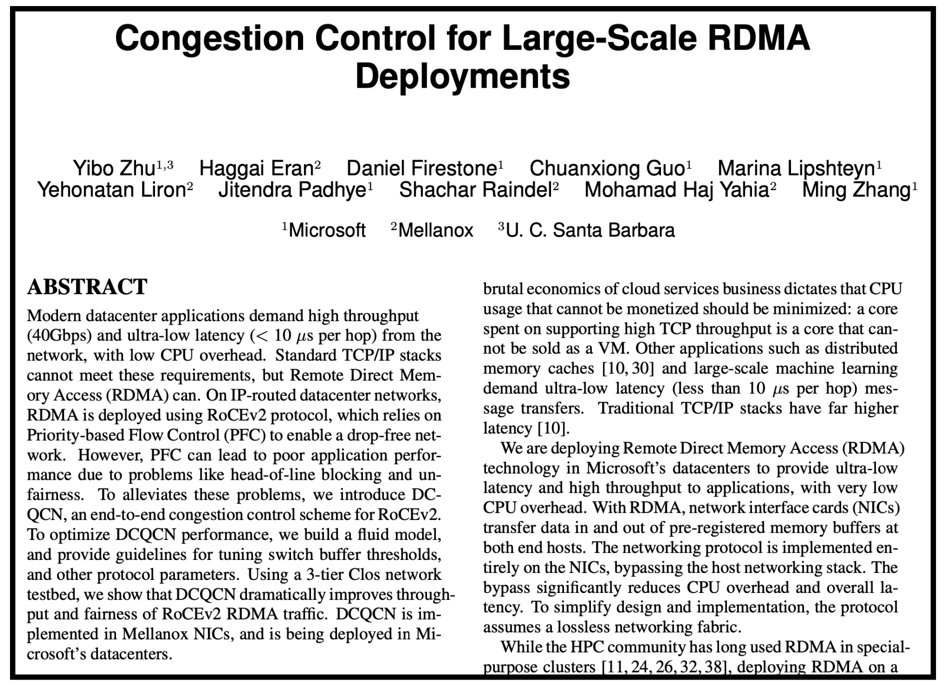
Abstract

Traditional measures of network goodness—goodput, quality of service, fairness—are expressed in terms of bandwidth. Network latency has rarely been a primary concern because delivering the highest level of bandwidth essentially entails driving up latency—at the mean and, especially, at the tail. Recently, however, there has been renewed interest in latency as a primary metric for mainstream applications. In this paper, we present the HULL (High-bandwidth Ultra-Low Latency) architecture to balance two seemingly contradictory goals: near baseline fabric latency and high bandwidth utiliza-

of-service capability to the Internet resulted in proposals such as RSVP [55], IntServ [10] and DiffServ [35], which again focussed on bandwidth provisioning.

This focus on bandwidth efficiency has been well justified as most Internet applications typically fall into two categories. Throughput-oriented applications, such as file transfer or email, are not sensitive to the delivery times of individual packets. Even the overall completion times of individual operations can vary by multiple integer factors in the interests of increasing overall network throughput. On the other hand, latency-sensitive applications—such as web browsing and remote login—

HULL, USENIX NSDI 2012
(ECN-based)



Congestion Control for Large-Scale RDMA Deployments

Yibo Zhu[1,3] Haggai Eran[2] Daniel Firestone[1] Chuanxiong Guo[1] Marina Lipshteyn[1] Yehonatan Liron[2] Jitendra Padhye[1] Shachar Raindel[2] Mohamad Haj Yahia[2] Ming Zhang[1]

[1]Microsoft [2]Mellanox [3]U. C. Santa Barbara

ABSTRACT

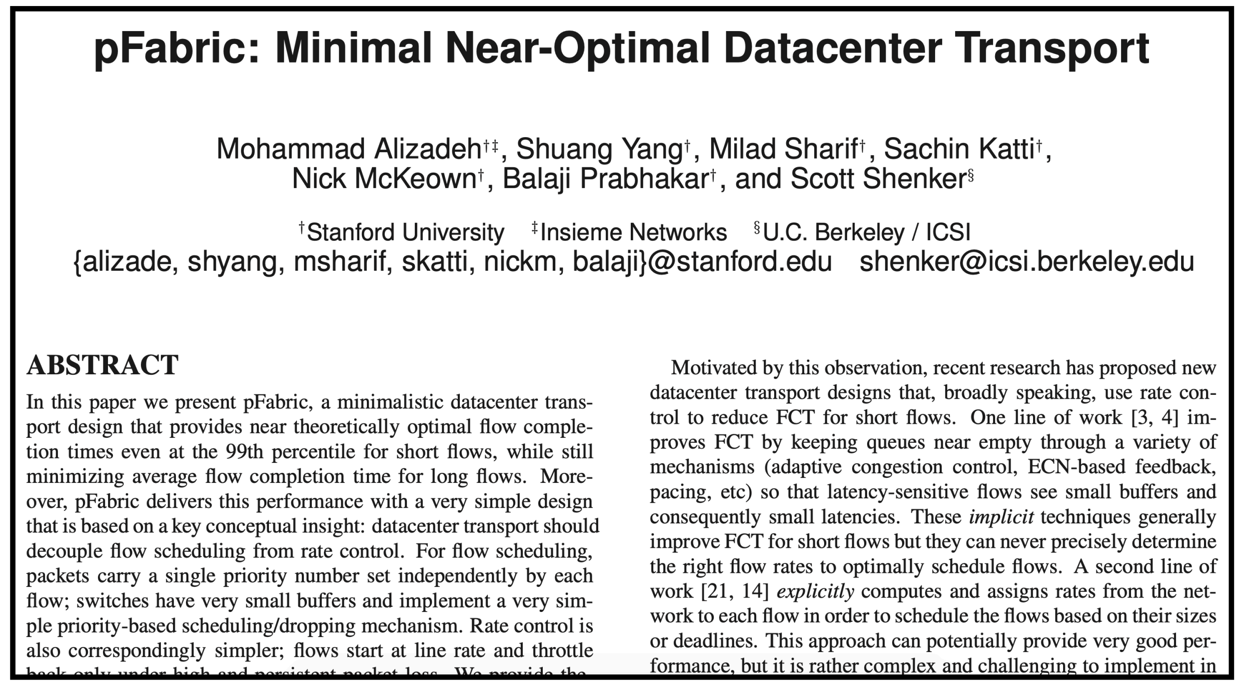
Modern datacenter applications demand high throughput (40Gbps) and ultra-low latency ($< 10\ \mu s$ per hop) from the network, with low CPU overhead. Standard TCP/IP stacks cannot meet these requirements, but Remote Direct Memory Access (RDMA) can. On IP-routed datacenter networks, RDMA is deployed using RoCEv2 protocol, which relies on Priority-based Flow Control (PFC) to enable a drop-free network. However, PFC can lead to poor application performance due to problems like head-of-line blocking and unfairness. To alleviates these problems, we introduce DCQCN, an end-to-end congestion control scheme for RoCEv2. To optimize DCQCN performance, we build a fluid model, and provide guidelines for tuning switch buffer thresholds, and other protocol parameters. Using a 3-tier Clos network testbed, we show that DCQCN dramatically improves throughput and fairness of RoCEv2 RDMA traffic. DCQCN is implemented in Mellanox NICs, and is being deployed in Microsoft's datacenters.

brutal economics of cloud services business dictates that CPU usage that cannot be monetized should be minimized: a core spent on supporting high TCP throughput is a core that cannot be sold as a VM. Other applications such as distributed memory caches [10, 30] and large-scale machine learning demand ultra-low latency (less than 10 μs per hop) message transfers. Traditional TCP/IP stacks have far higher latency [10].

We are deploying Remote Direct Memory Access (RDMA) technology in Microsoft's datacenters to provide ultra-low latency and high throughput to applications, with very low CPU overhead. With RDMA, network interface cards (NICs) transfer data in and out of pre-registered memory buffers at both end hosts. The networking protocol is implemented entirely on the NICs, bypassing the host networking stack. The bypass significantly reduces CPU overhead and overall latency. To simplify design and implementation, the protocol assumes a lossless networking fabric.

While the HPC community has long used RDMA in special-purpose clusters [11, 24, 26, 32, 38], deploying RDMA on a

DCQCN, ACM SIGCOMM 2015
(Explicit feedback)



pFabric: Minimal Near-Optimal Datacenter Transport

Mohammad Alizadeh†‡, Shuang Yang†, Milad Sharif†, Sachin Katti†, Nick McKeown†, Balaji Prabhakar†, and Scott Shenker§

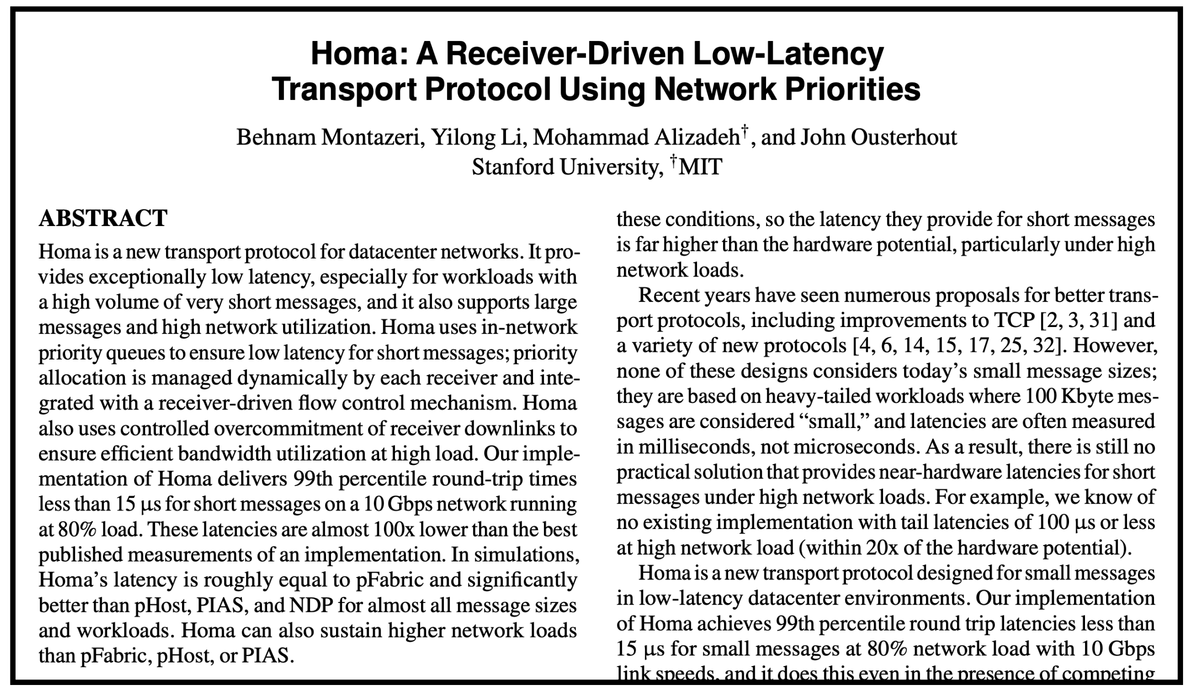
†Stanford University ‡Insieme Networks §U.C. Berkeley / ICSI
{alizade, shyang, msharif, skatti, nickm, balaji}@stanford.edu shenker@icsi.berkeley.edu

ABSTRACT

In this paper we present pFabric, a minimalistic datacenter transport design that provides near theoretically optimal flow completion times even at the 99th percentile for short flows, while still minimizing average flow completion time for long flows. Moreover, pFabric delivers this performance with a very simple design that is based on a key conceptual insight: datacenter transport should decouple flow scheduling from rate control. For flow scheduling, packets carry a single priority number set independently by each flow; switches have very small buffers and implement a very simple priority-based scheduling/dropping mechanism. Rate control is also correspondingly simpler; flows start at line rate and throttle

Motivated by this observation, recent research has proposed new datacenter transport designs that, broadly speaking, use rate control to reduce FCT for short flows. One line of work [3, 4] improves FCT by keeping queues near empty through a variety of mechanisms (adaptive congestion control, ECN-based feedback, pacing, etc) so that latency-sensitive flows see small buffers and consequently small latencies. These *implicit* techniques generally improve FCT for short flows but they can never precisely determine the right flow rates to optimally schedule flows. A second line of work [21, 14] *explicitly* computes and assigns rates from the network to each flow in order to schedule the flows based on their sizes or deadlines. This approach can potentially provide very good performance, but it is rather complex and challenging to implement in

pFabric, ACM SIGCOMM 2013
(Packet scheduling)



Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities

Behnam Montazeri, Yilong Li, Mohammad Alizadeh†, and John Ousterhout
Stanford University, †MIT

ABSTRACT

Homa is a new transport protocol for datacenter networks. It provides exceptionally low latency, especially for workloads with a high volume of very short messages, and it also supports large messages and high network utilization. Homa uses in-network priority queues to ensure low latency for short messages; priority allocation is managed dynamically by each receiver and integrated with a receiver-driven flow control mechanism. Homa also uses controlled overcommitment of receiver downlinks to ensure efficient bandwidth utilization at high load. Our implementation of Homa delivers 99th percentile round-trip times less than 15 µs for short messages on a 10 Gbps network running at 80% load. These latencies are almost 100x lower than the best published measurements of an implementation. In simulations, Homa's latency is roughly equal to pFabric and significantly better than pHost, PIAS, and NDP for almost all message sizes and workloads. Homa can also sustain higher network loads than pFabric, pHost, or PIAS.

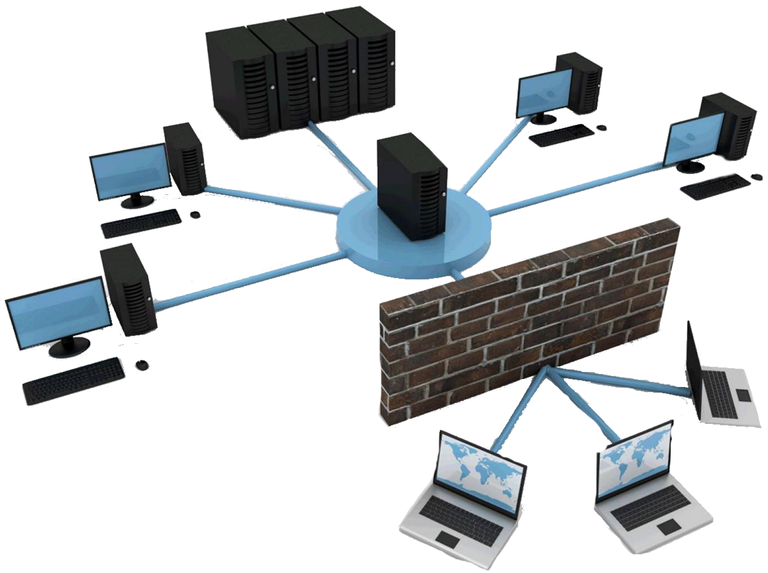
these conditions, so the latency they provide for short messages is far higher than the hardware potential, particularly under high network loads.

Recent years have seen numerous proposals for better transport protocols, including improvements to TCP [2, 3, 31] and a variety of new protocols [4, 6, 14, 15, 17, 25, 32]. However, none of these designs considers today's small message sizes; they are based on heavy-tailed workloads where 100 Kbyte messages are considered "small," and latencies are often measured in milliseconds, not microseconds. As a result, there is still no practical solution that provides near-hardware latencies for short messages under high network loads. For example, we know of no existing implementation with tail latencies of 100 µs or less at high network load (within 20x of the hardware potential).

Homa is a new transport protocol designed for small messages in low-latency datacenter environments. Our implementation of Homa achieves 99th percentile round trip latencies less than 15 µs for small messages at 80% network load with 10 Gbps

Homa, ACM SIGCOMM 2018
(Credit-based)

Next week: software defined networking



How do we manage a complex network?

- Remember all the protocols
- Remember the configurations with every protocol
- Diagnose problems with networking tools like ping, traceroute, tcpdump?