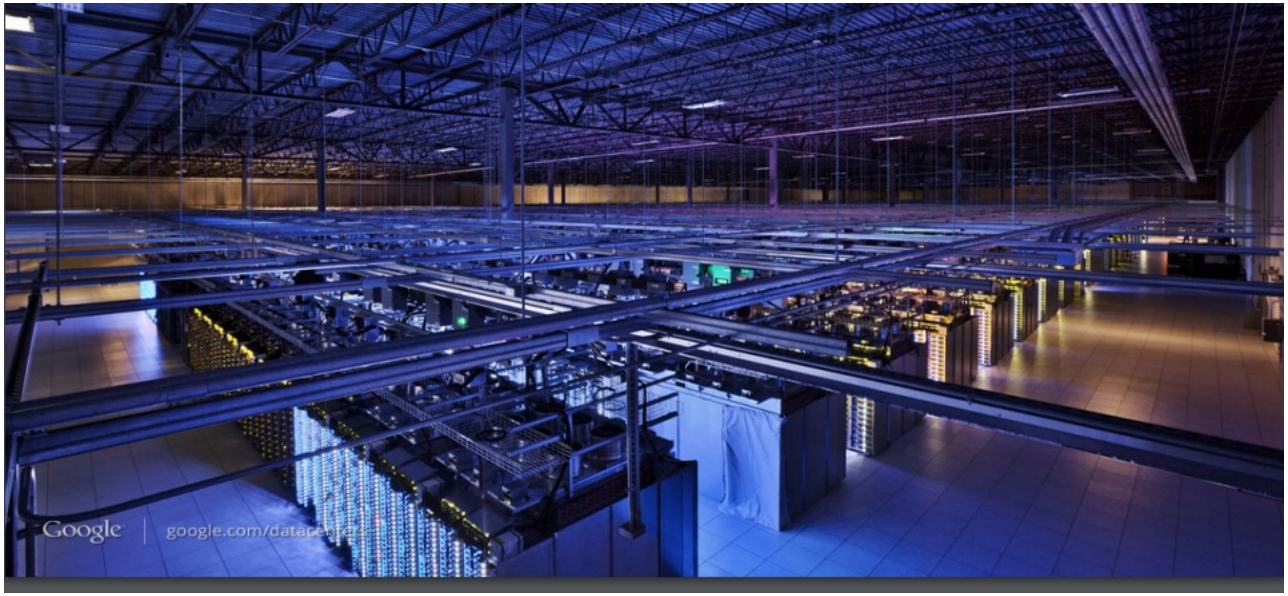
Data Center Congestion Control

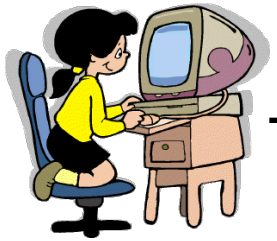


Note: The slides are adapted from the authors' materials.

Mohammad Alizadeh et. al, DCTCP [SIGCOMM'10],

Radhika Mittal et. al, TIMELY [SIGCOMM'15]

Transport inside the DC



100Kbps–100Mbps links

~100ms latency

INTERNET

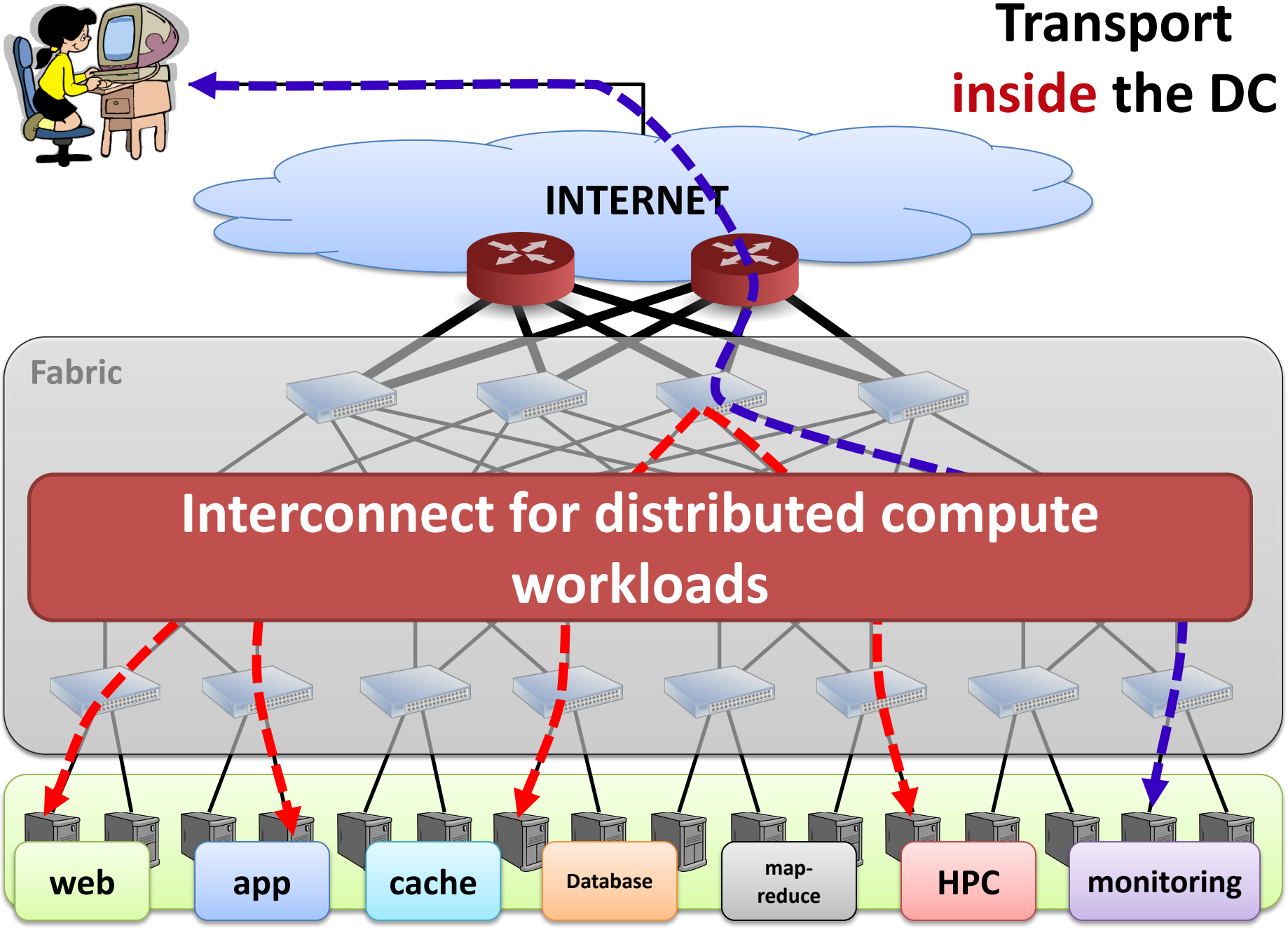
Fabric

10–40Gbps links

~10–100µs latency

Servers

Transport inside the DC



What's Different About DC Transport?

- Network characteristics
  - Very high link speeds (Gb/s); very low latency (microseconds)
- Application characteristics
  - Large-scale distributed computation
- Challenging traffic patterns
  - Diverse mix of mice & elephants
  - Incast
- Cheap switches
  - Single-chip shared-memory devices; shallow buffers

Data Center Workloads

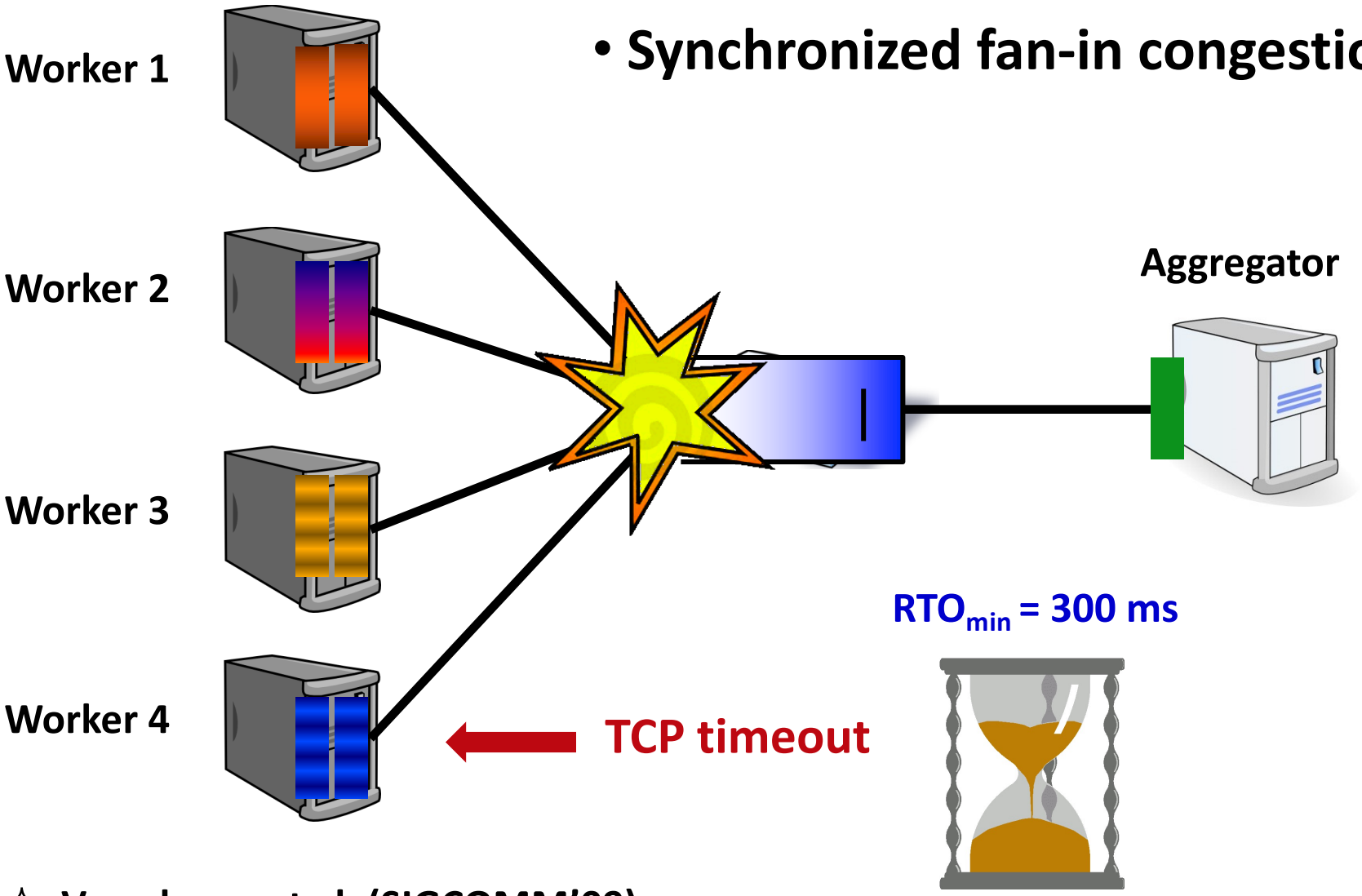
Mice & Elephants

- Short messages
  (e.g., query, coordination) → **Low Latency**

- Large flows
  (e.g., data update, backup) → **High Throughput**

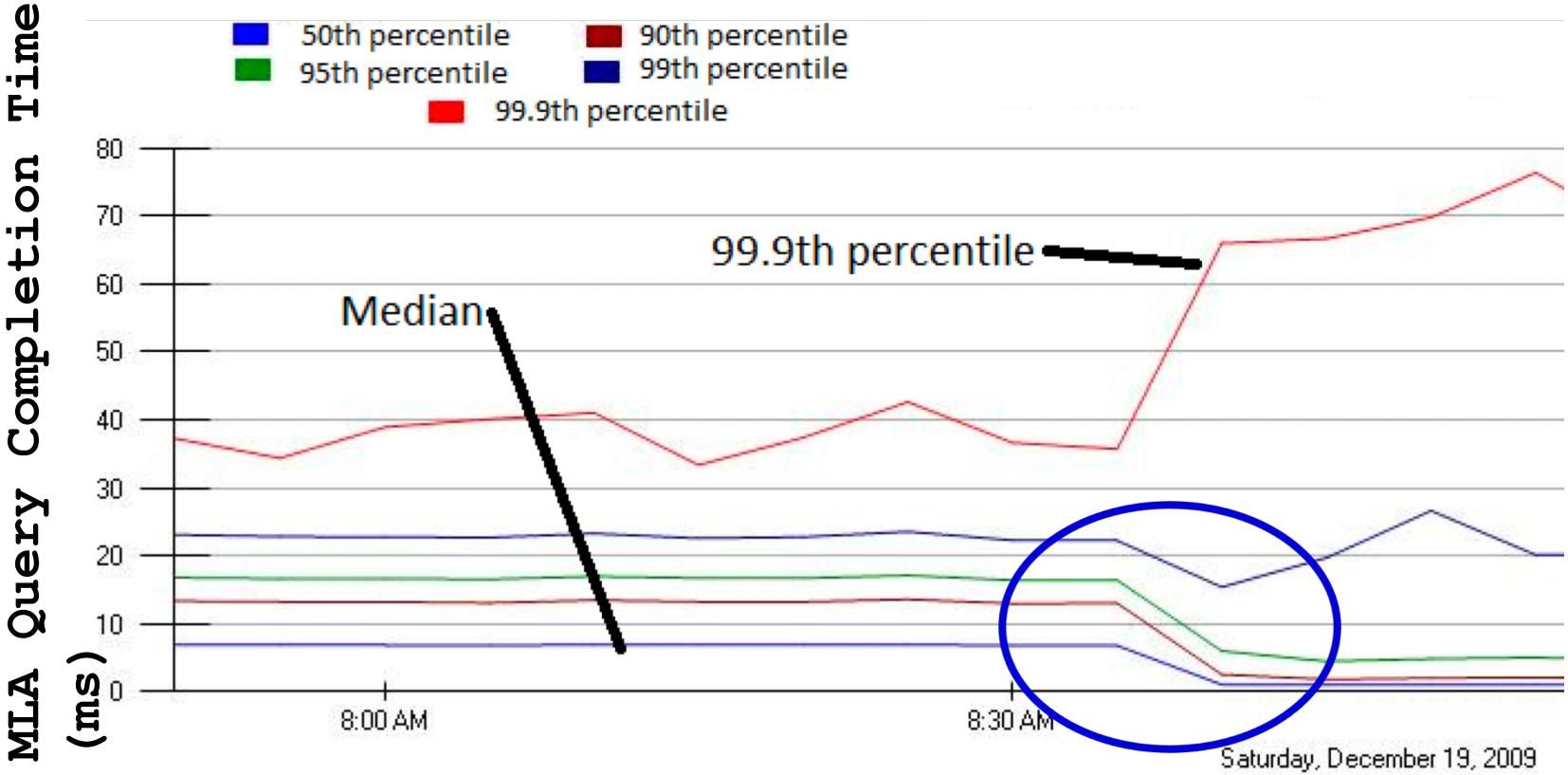
Incast

- Synchronized fan-in congestion



✧ Vasudevan et al. (SIGCOMM'09)

Incast in Bing



Jittering trades of median for high percentiles

DC Transport Requirements

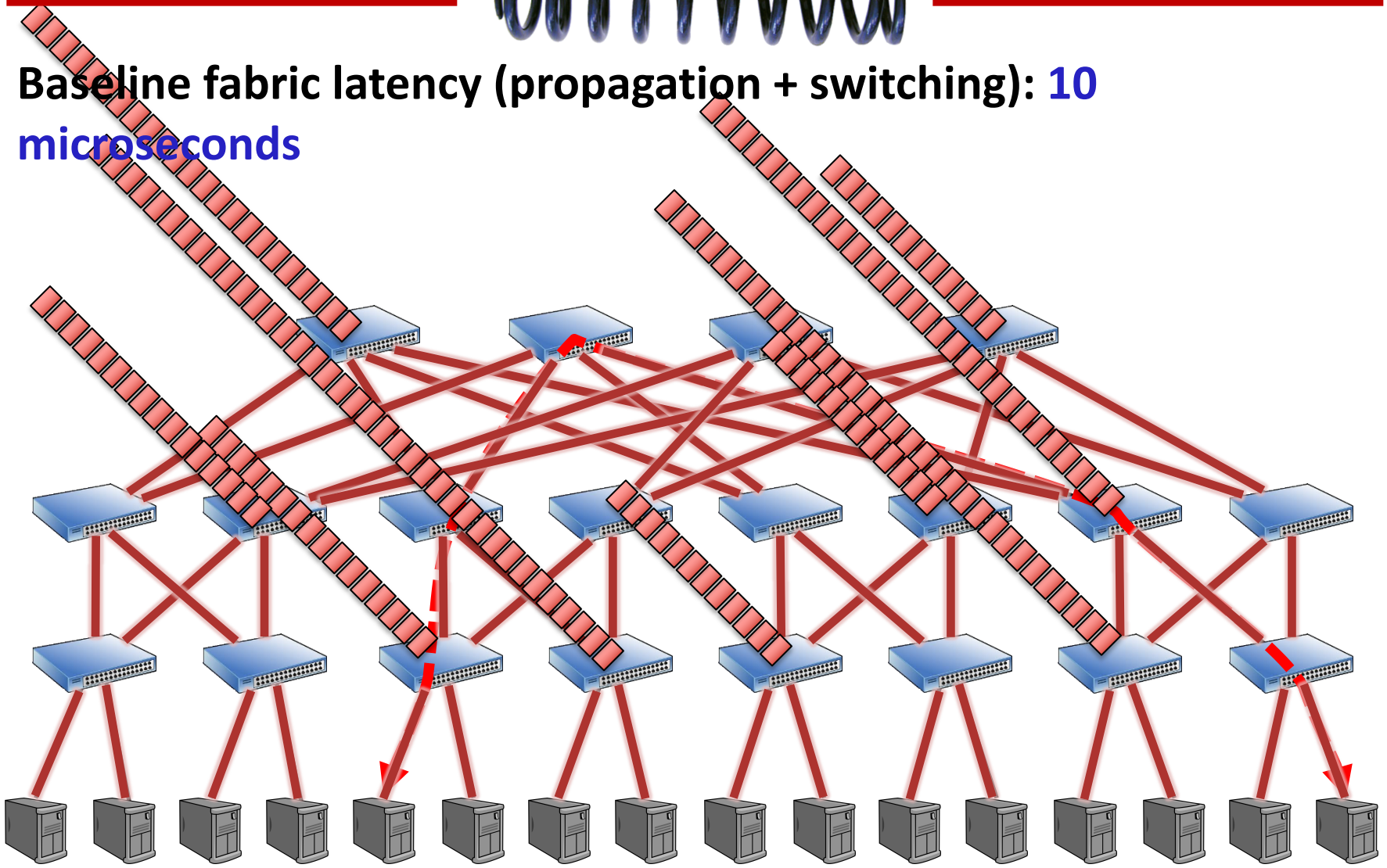
1. **Low Latency**
   - Short messages, queries
2. **High Throughput**
   - Continuous data updates, backups
3. **High Burst Tolerance**
   - Incast

The challenge is to achieve these *together*

High Throughput

Low Latency

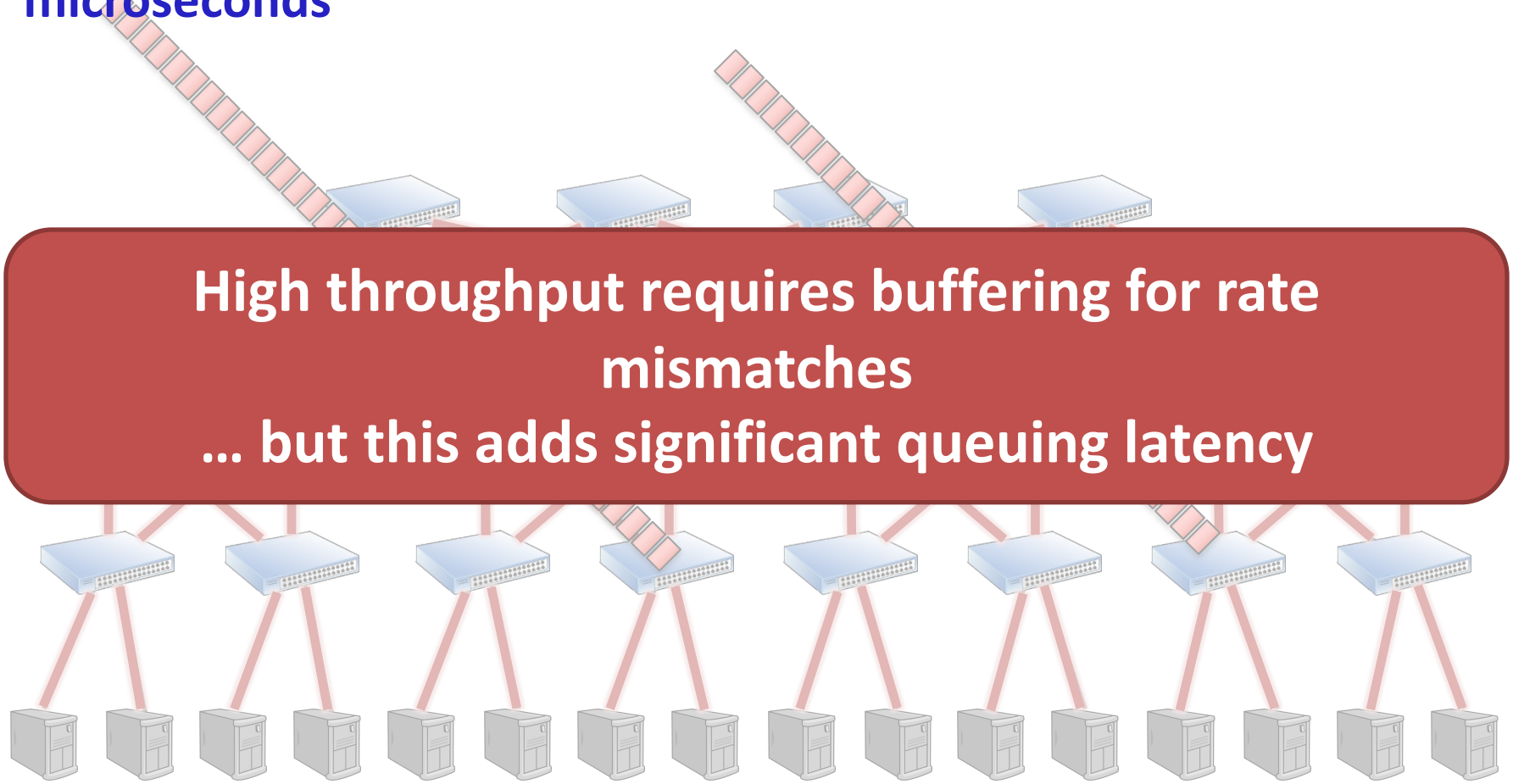
Baseline fabric latency (propagation + switching): 10 microseconds



High Throughput

Low Latency

Baseline fabric latency (propagation + switching): 10 microseconds



High throughput requires buffering for rate mismatches
… but this adds significant queuing latency

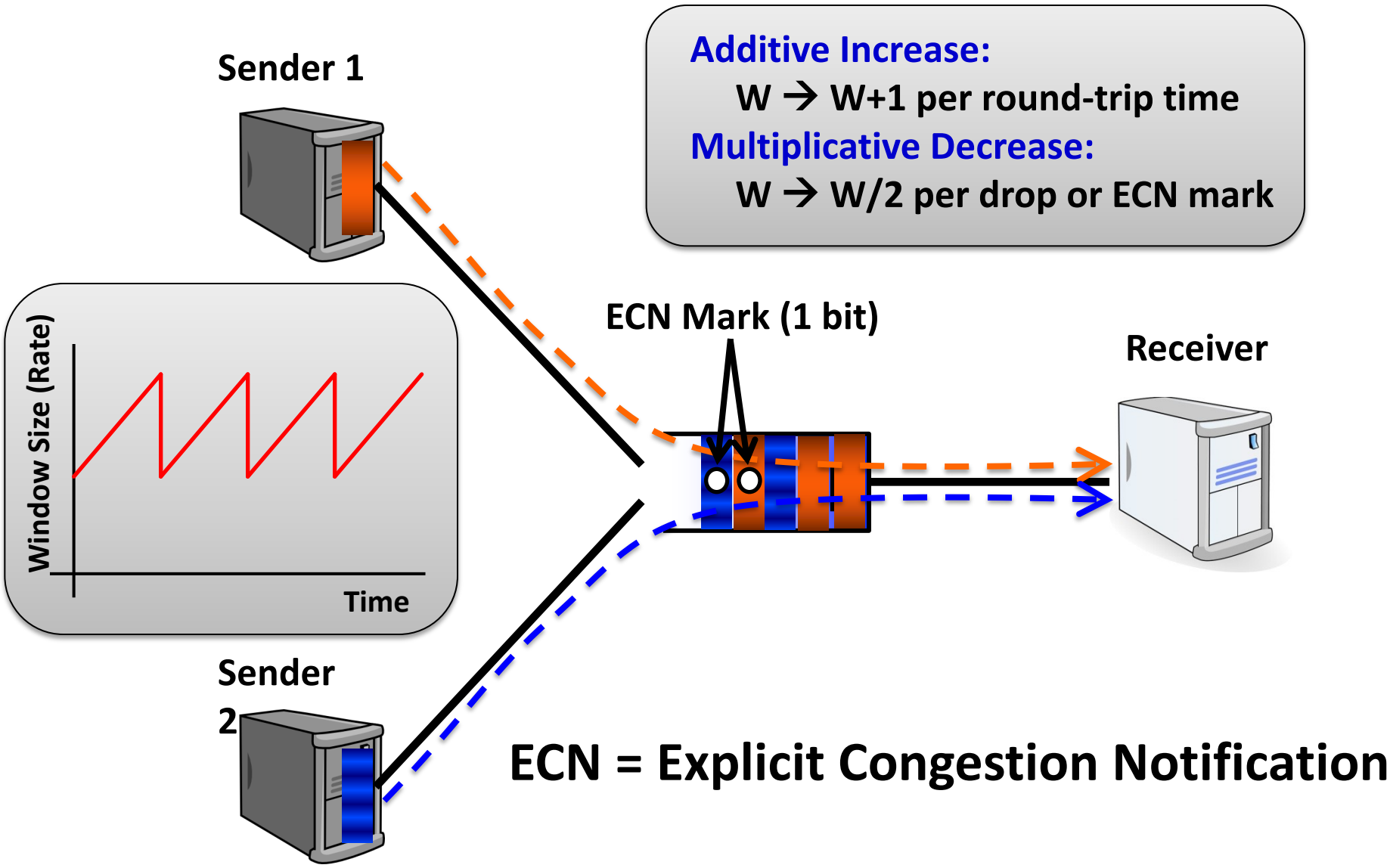
Data Center TCP

TCP in the Data Center

- TCP [Jacobsen et al.'88] is widely used in the data center
  - More than **99%** of the traffic
- Operators work around TCP problems
  - Ad-hoc, inefficient, often expensive solutions
  - TCP is deeply ingrained in applications

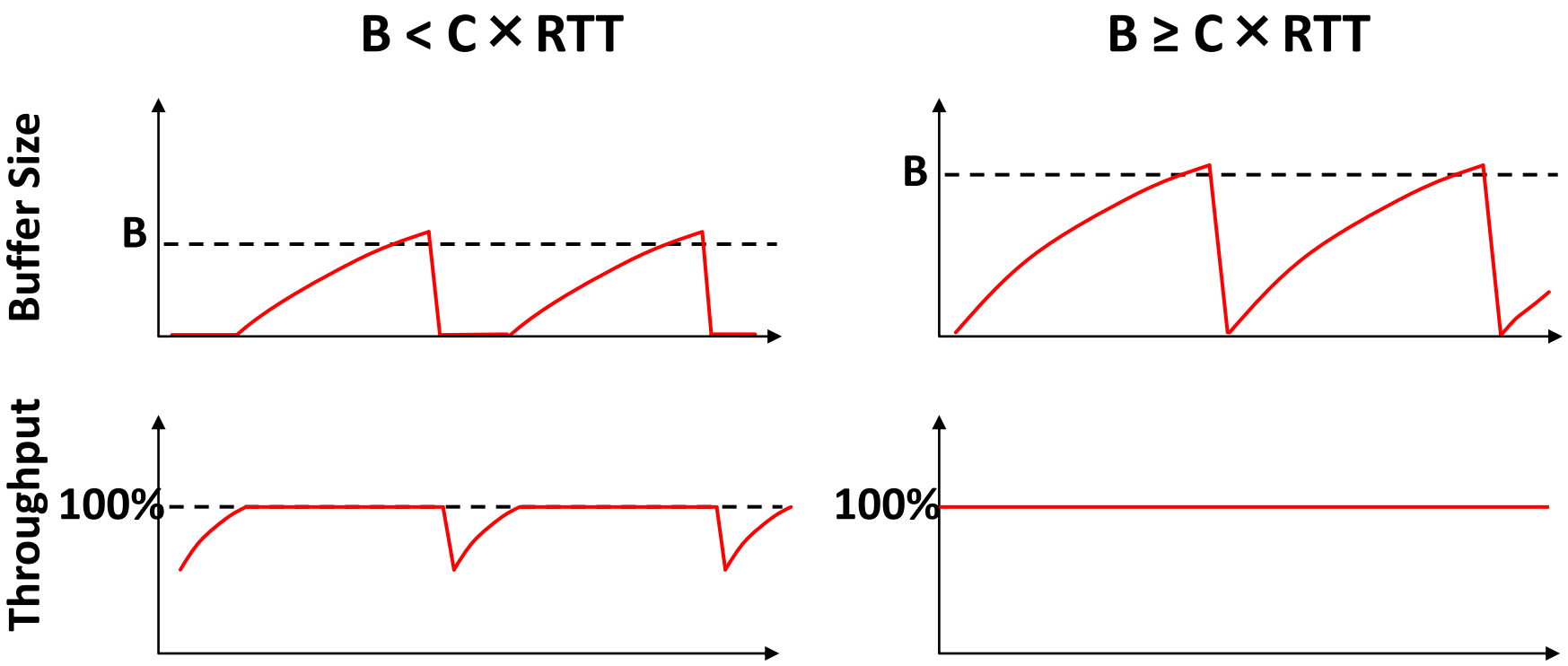
Practical deployment is hard
→ keep it simple!

Review: The TCP Algorithm



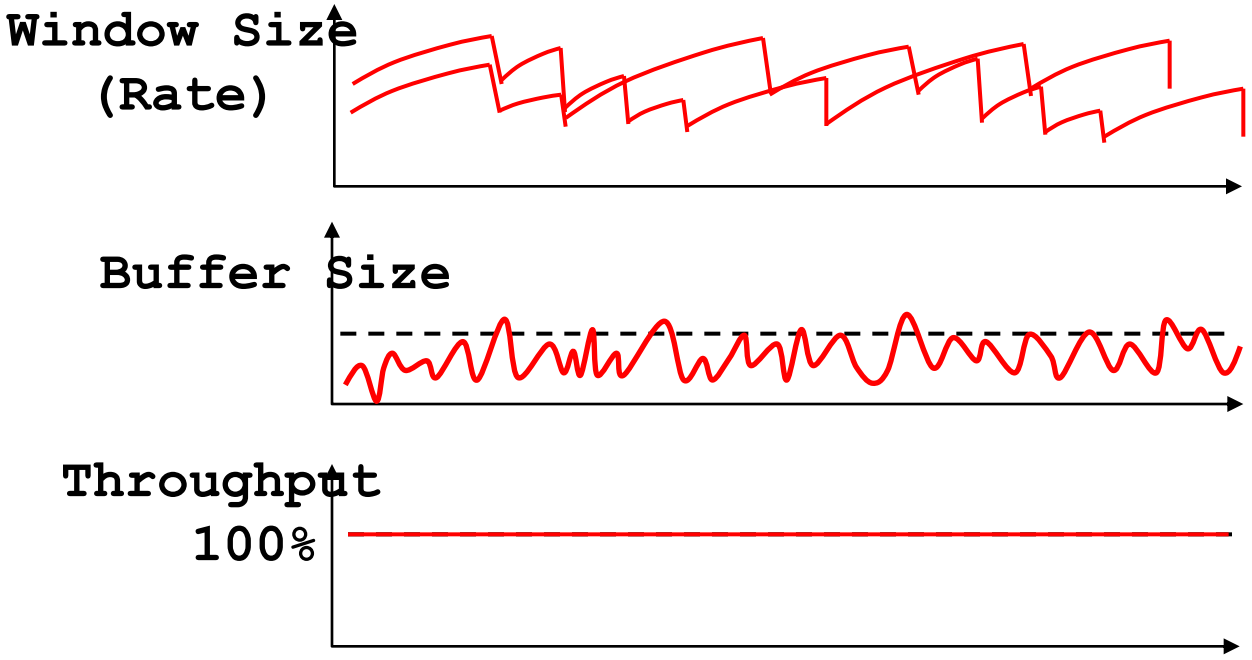
TCP Buffer Requirement

- Bandwidth-delay product rule of thumb:
  - A single flow needs **C × RTT** buffers for **100% Throughput.**



Reducing Buffer Requirements

- Appenzeller et al. (SIGCOMM '04):
  - Large # of flows: $C \times RTT/\sqrt{N}$ is enough.



Reducing Buffer Requirements

- Appenzeller et al. (SIGCOMM '04):
  - Large # of flows: $C \times RTT/\sqrt{N}$ is enough

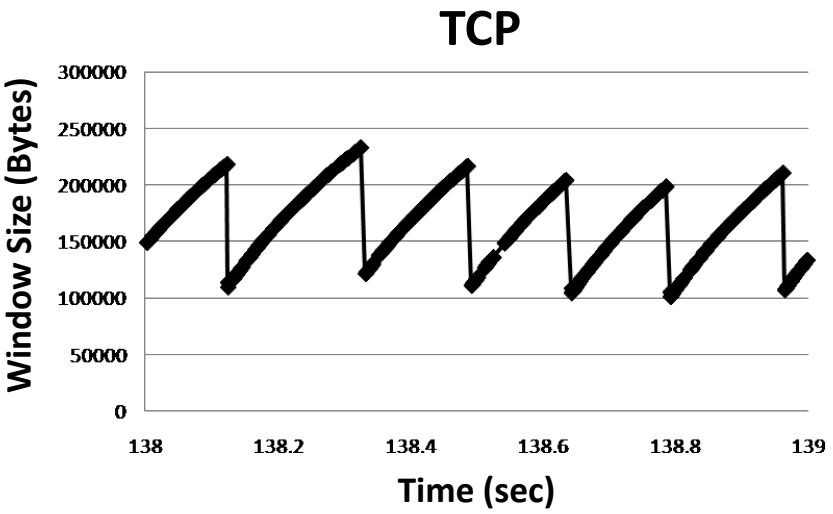
- Can't rely on stat-mux benefit in the DC.
  - Measurements show typically **only 1-2 large flows** at each server

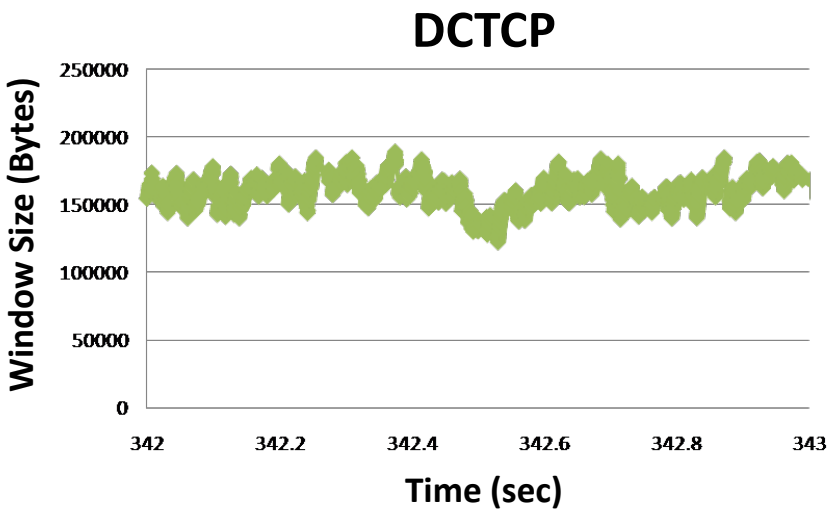
Key Observation:
Low variance in sending rate → Small buffers suffice

DCTCP: Main Idea

- Extract multi-bit feedback from single-bit stream of ECN marks
  - Reduce window size based on **fraction** of marked packets.

ECN Marks	TCP	DCTCP
1 0 1 1 1 1 0 1 1 1	Cut window by 50%	Cut window by 40%
0 0 0 0 0 0 0 0 0 1	Cut window by 50%	Cut window by 5%

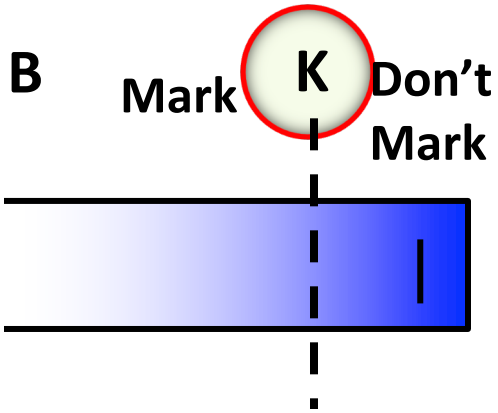




DCTCP: Algorithm

Switch side:

– Mark packets when **Queue Length > K.**



Sender side:

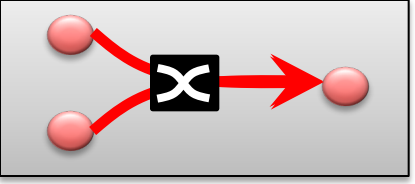
– **Maintain running average of *fraction* of packets marked (α).**

$$\text{each RTT}: \; F = \frac{\#\text{ of marked ACKs}}{\text{Total \# of ACKs}} \;\Rightarrow\; \alpha \leftarrow (1-g)\alpha + gF$$

$$W \leftarrow (1-\frac{\alpha}{2})W$$

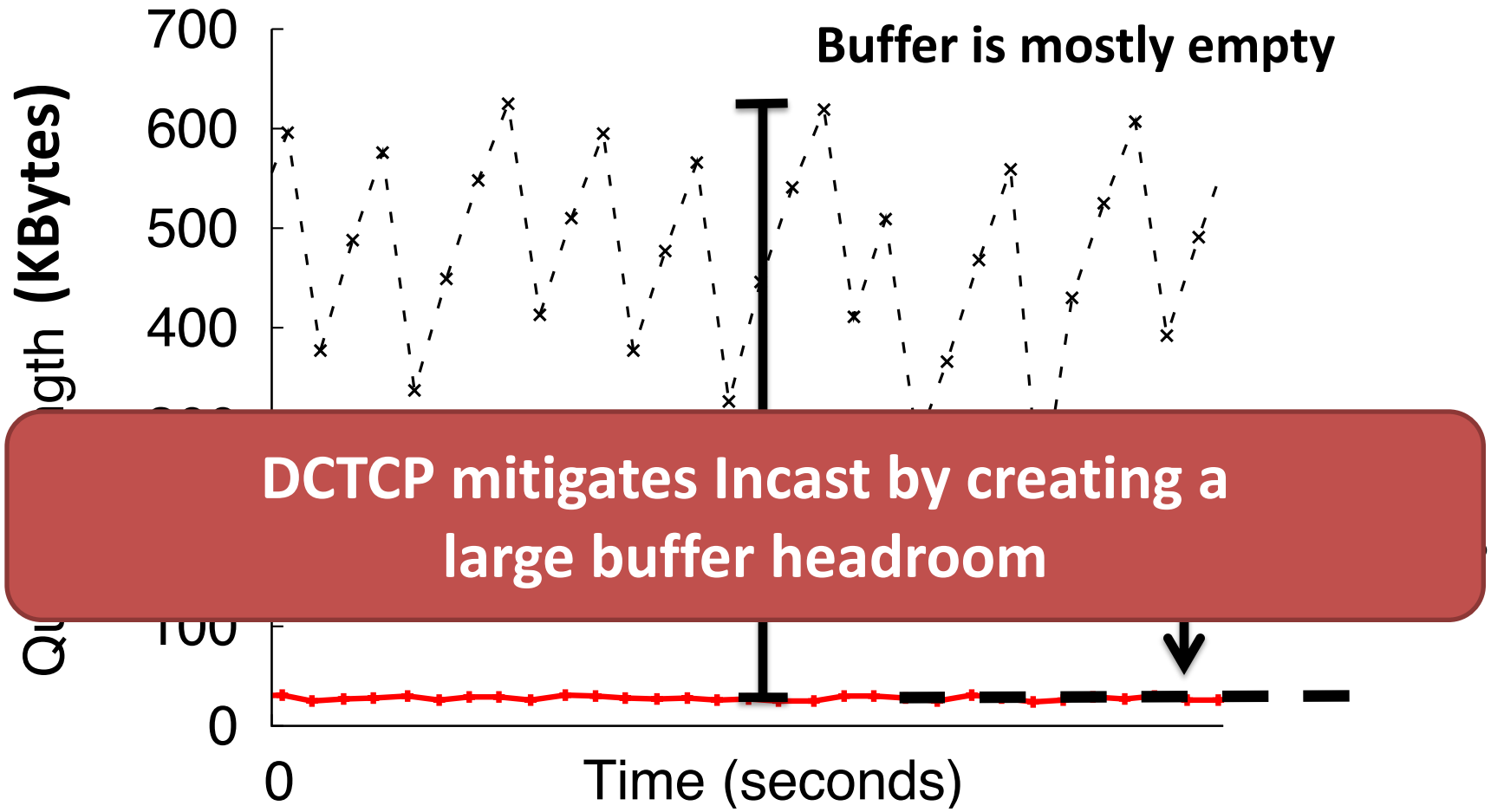
➢ **Adaptive window decreases:**

– **Note: decrease factor between 1 and 2.**



DCTCP vs TCP

Experiment: 2 flows (Win 7 stack), Broadcom 1Gbps Switch



Why it Works

1. Low Latency

- ✓ **Small buffer occupancies** → low queuing delay

2. High Throughput

- ✓ **ECN averaging** → smooth rate adjustments, low variance

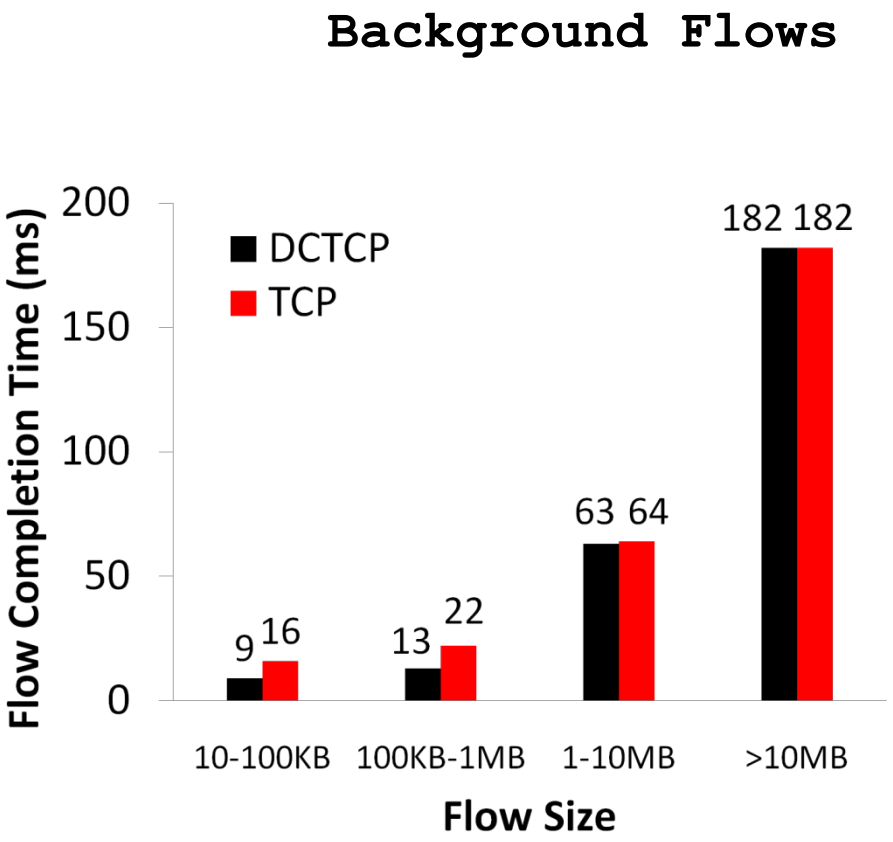
3. High Burst Tolerance

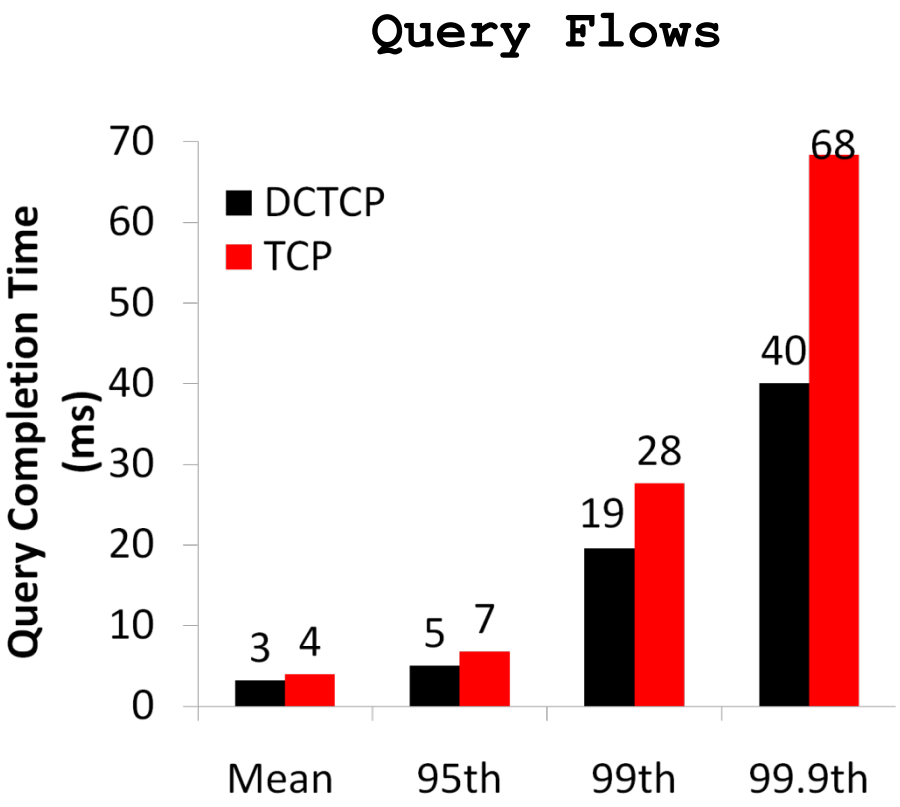
- ✓ **Large buffer headroom** → bursts fit
- ✓ **Aggressive marking** → sources react before packets are dropped

Evaluation

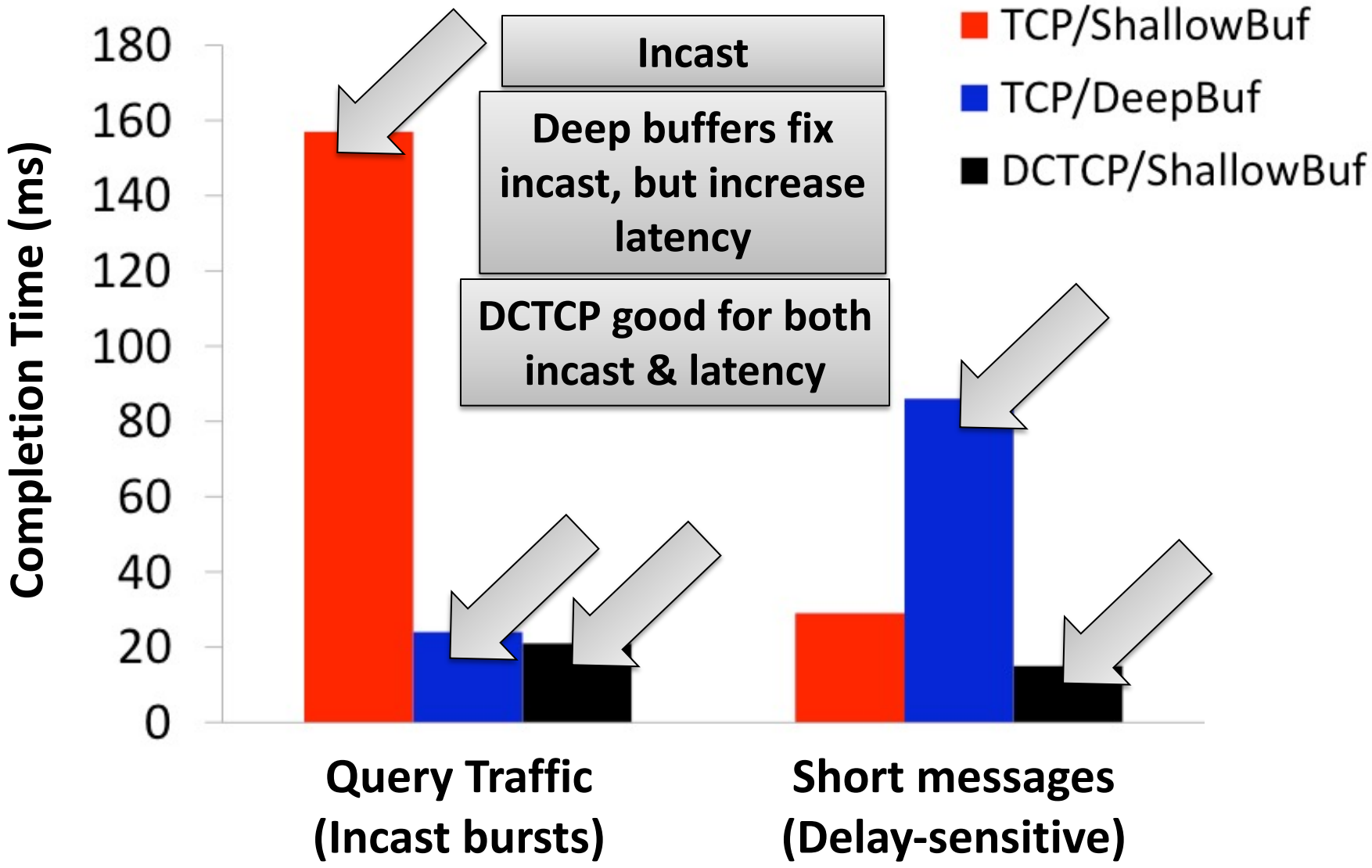
- Implemented in Windows stack.
- Real hardware, **1Gbps and 10Gbps** experiments
  - **90 server testbed**
  - **Broadcom Triumph** 48 1G ports – 4MB shared memory
  - **Cisco Cat4948** 48 1G ports – 16MB shared memory
  - **Broadcom Scorpion** 24 10G ports – 4MB shared memory
- Numerous micro-benchmarks
  - **Throughput and Queue Length**
  - **Multi-hop**
  - **Queue Buildup**
  - **Buffer Pressure**
  - **Fairness and Convergence**
  - **Incast**
  - **Static vs Dynamic Buffer Mgmt**
- **Bing cluster benchmark**

Bing Benchmark (baseline)





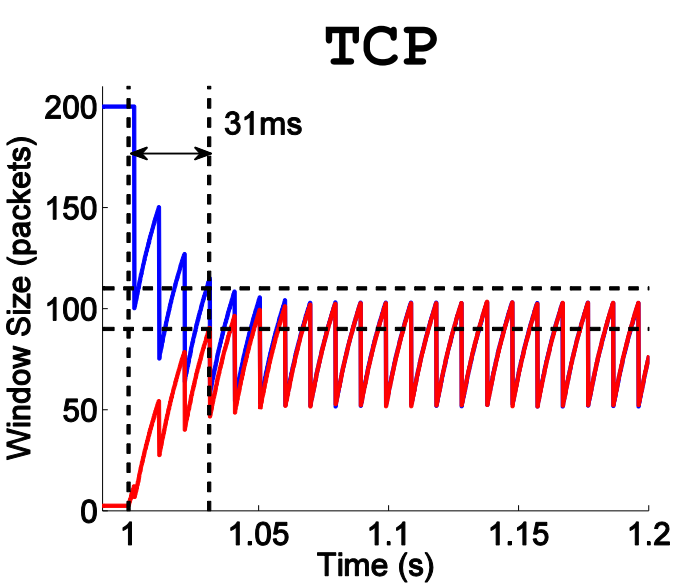
Bing Benchmark (scaled 10x)

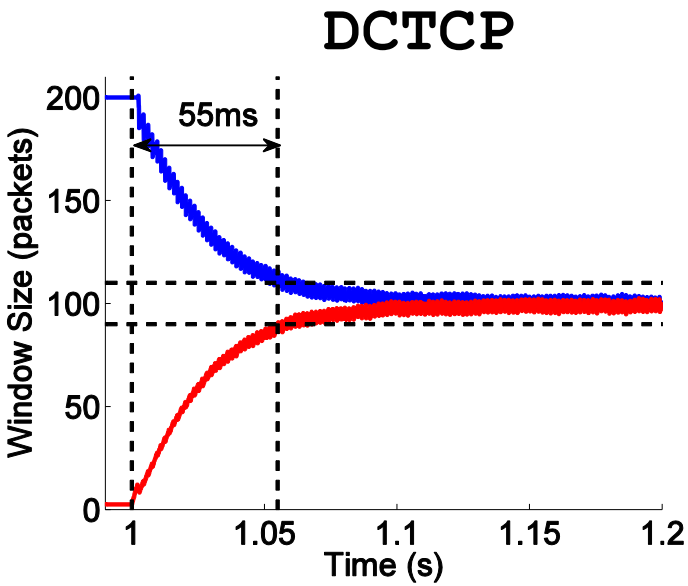


Convergence Time

- DCTCP takes at most ~40% more **RTTs** than TCP
  - "Analysis of DCTCP: Stability, Convergence, and Fairness," SIGMETRICS 2011

Intuition: DCTCP makes smaller adjustments than TCP, but makes them much more frequently





TIMELY

Qualities of RTT

- Fine-grained and informative
- Quick response time
- No switch support needed
- End-to-end metric
- Works seamlessly with QoS

Applicability in Datacenters

- RTT-based schemes discarded for WANs.
  - Compete poorly with loss-based schemes.

- **This is not a concern for the Datacenters.**

Stringent Performance Requirements

- Tightly coupled computing tasks
- Require both **high throughput** and **low latencies**
- Packet losses are too costly

However, it was too hard to measure the RTTs accurately.

While RTT was banished from WANs,

it was never even
considered for Datacenters!!

And ECN emerged as a hero instead.

ECN

DCTCP (2010)
D^2TCP (2012)
HULL (2012)
TCP-Bolt (2014)
DCQCN (2015)

Contributions

1. Show that accurate RTT measurements are possible.
2. Demonstrate the goodness of RTT as a congestion signal.
3. Develop TIMELY: RTT-based congestion control for the datacenters.

Accurate RTT Measurement

Hardware Assisted RTT Measurement

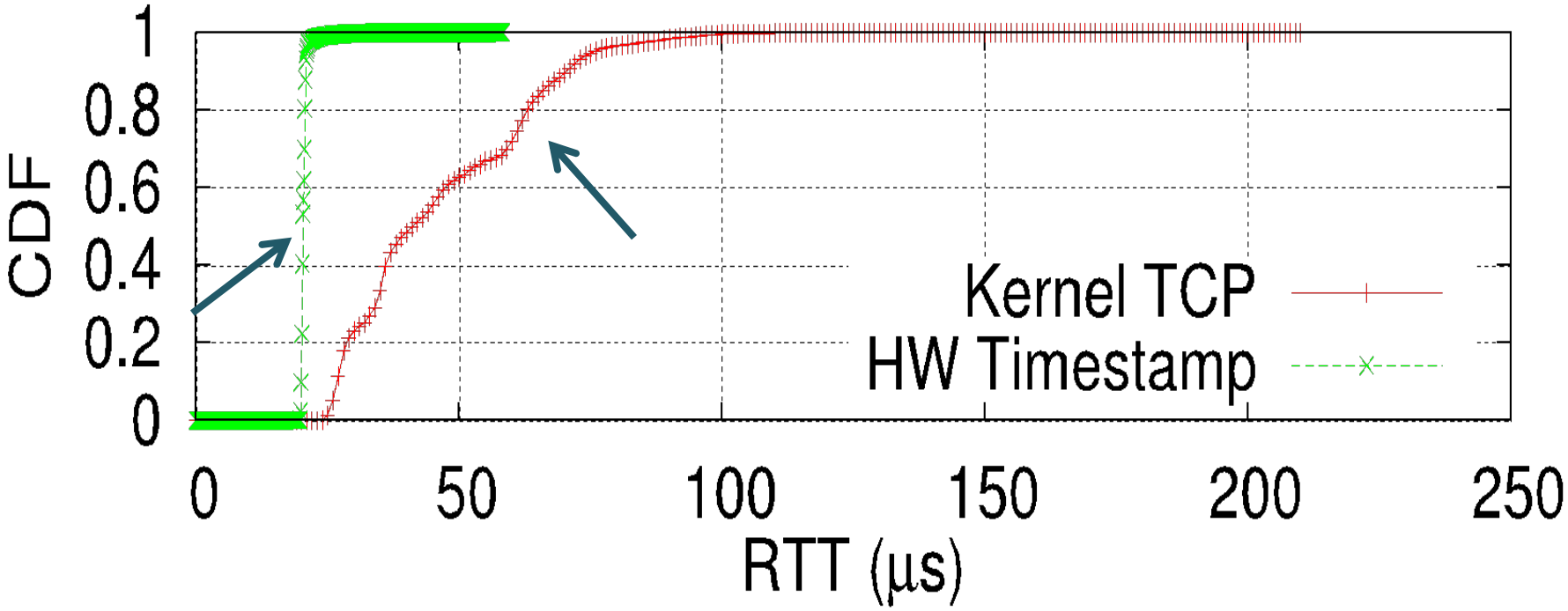
Hardware Timestamps

- mitigate noise in measurements

Hardware Acknowledgements

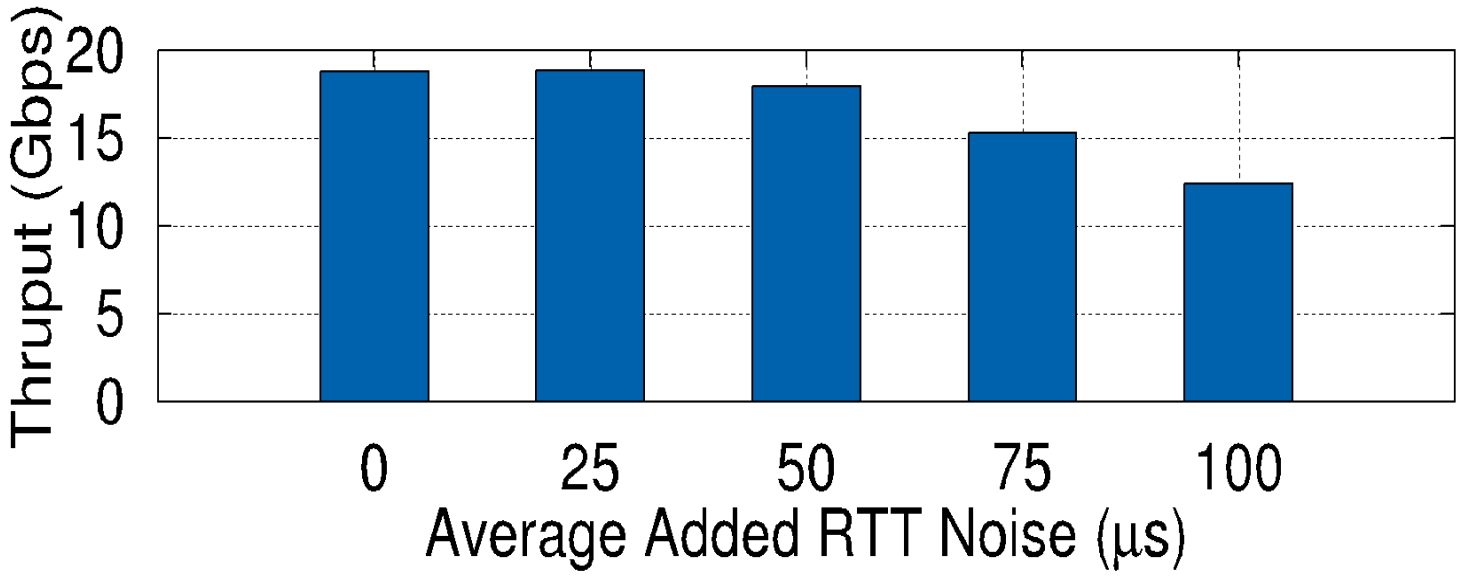
- avoid processing overhead

Hardware vs Software Timestamps



Kernel Timestamps introduce significant noise in RTT measurements compared to HW Timestamps.

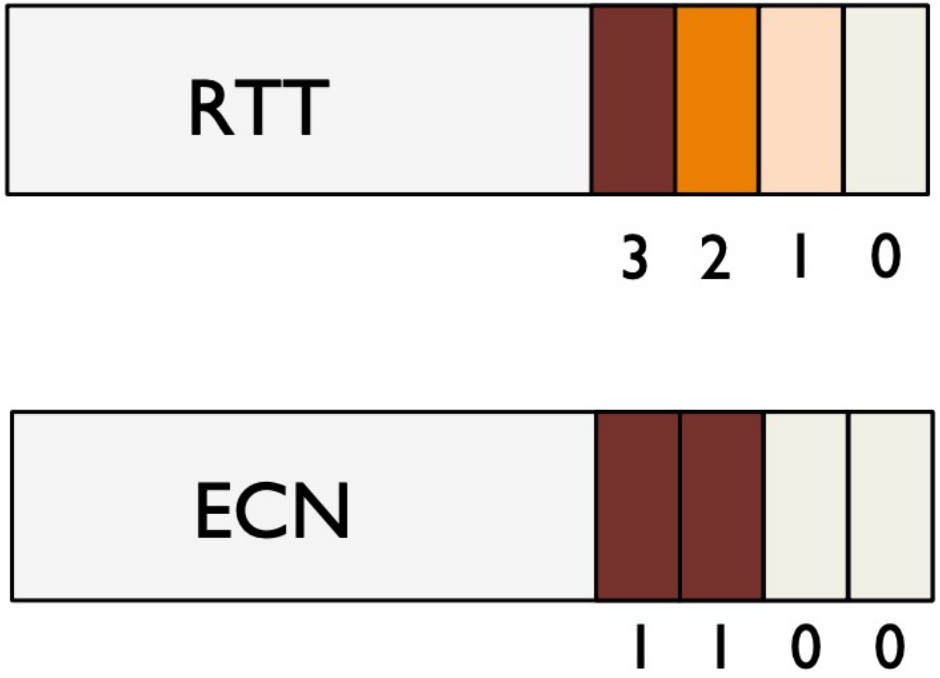
Impact of RTT Noise



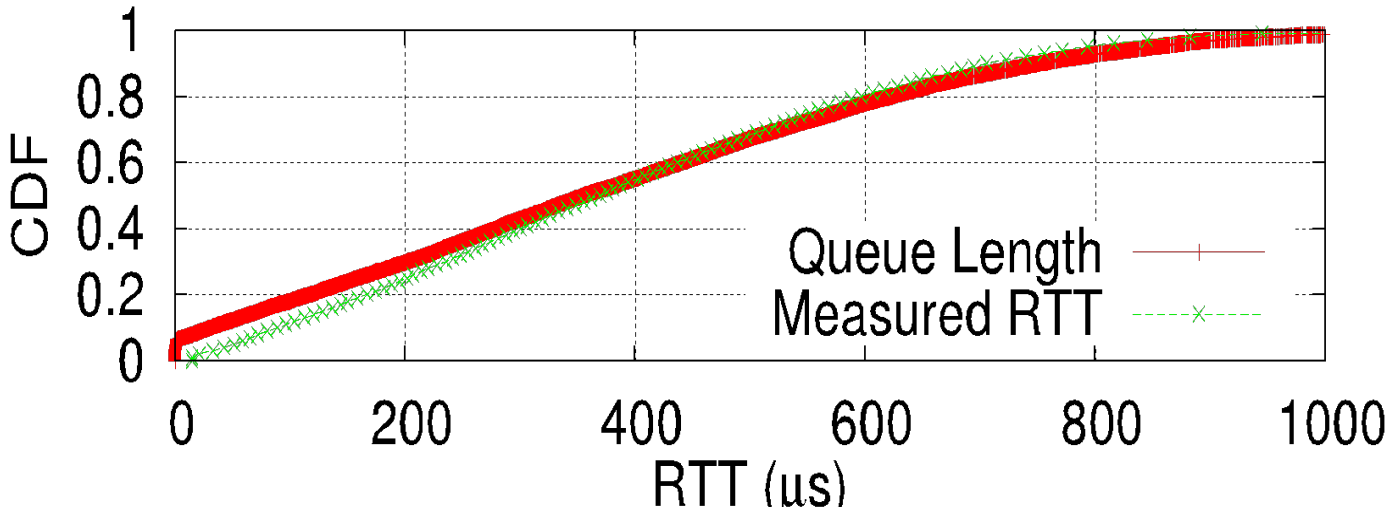
Throughput degrades with increasing noise in RTT.
Precise RTT measurement is crucial.

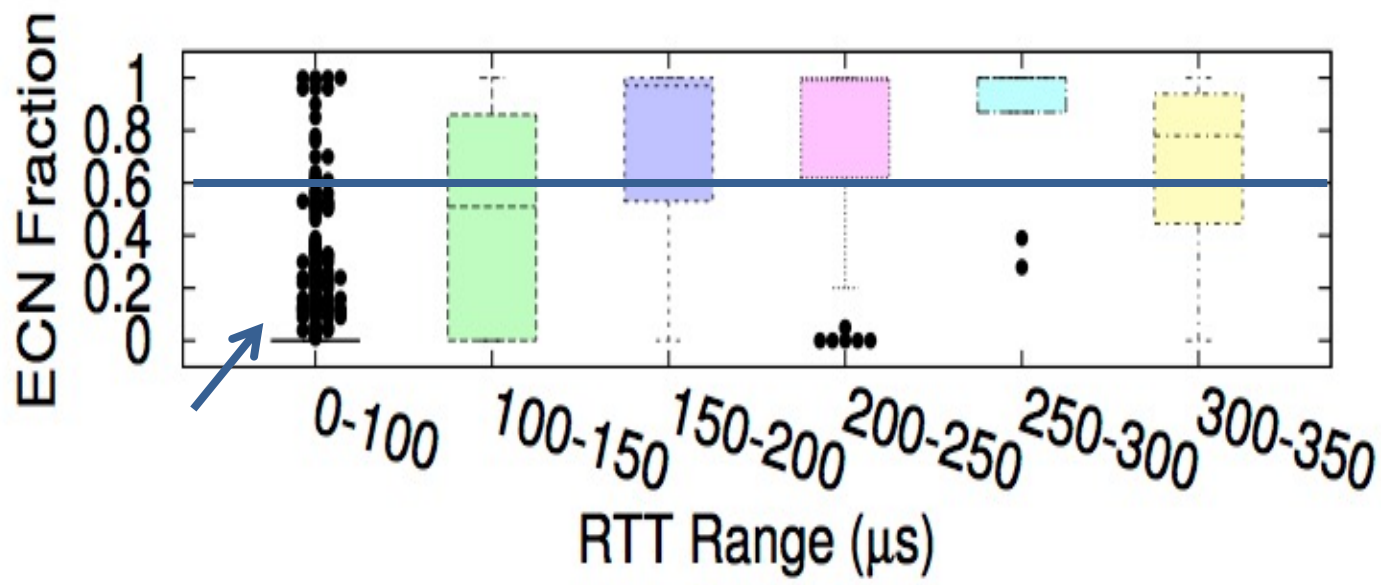
RTT as a congestion signal

RTT is a multi-bit signal



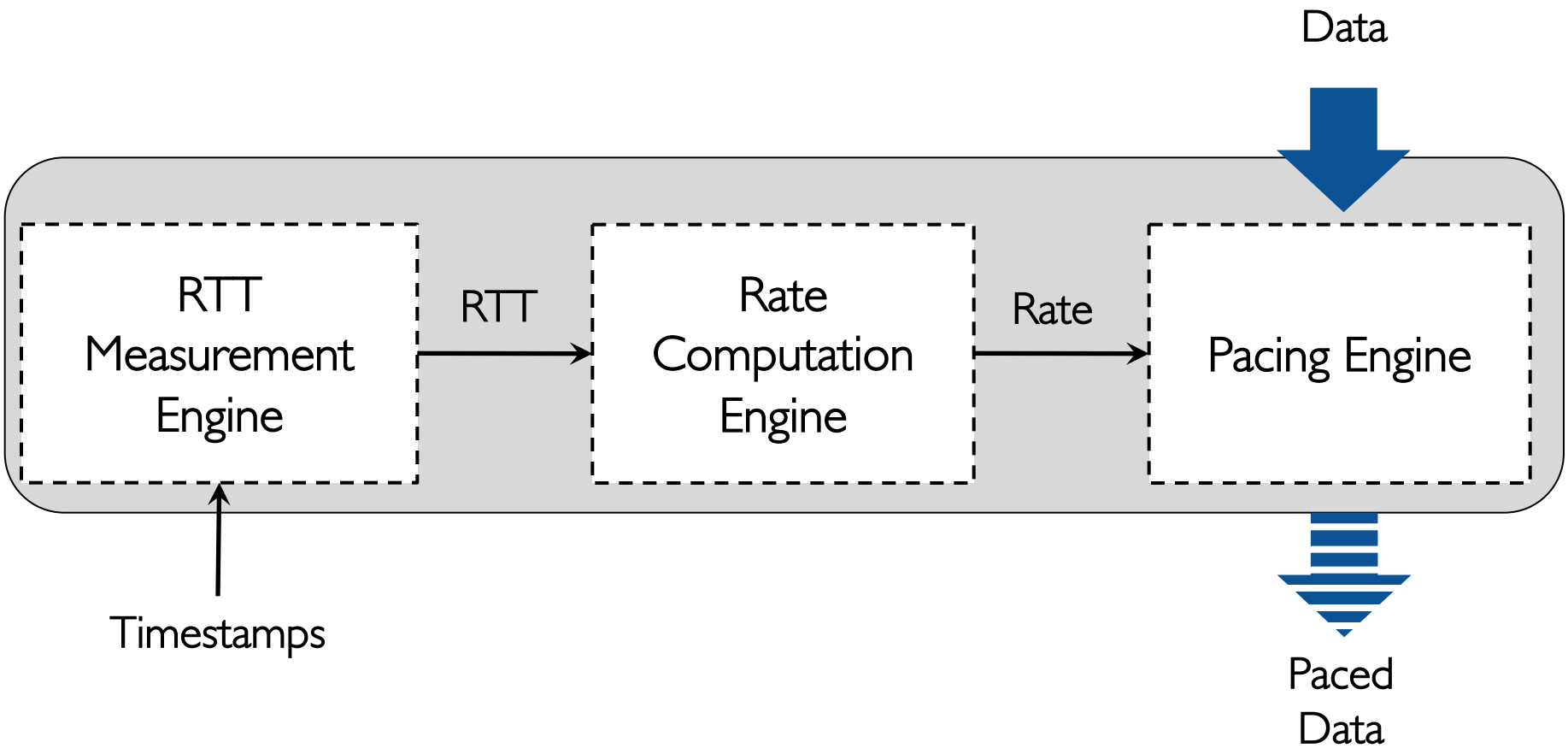
RTT correlates with queuing delay



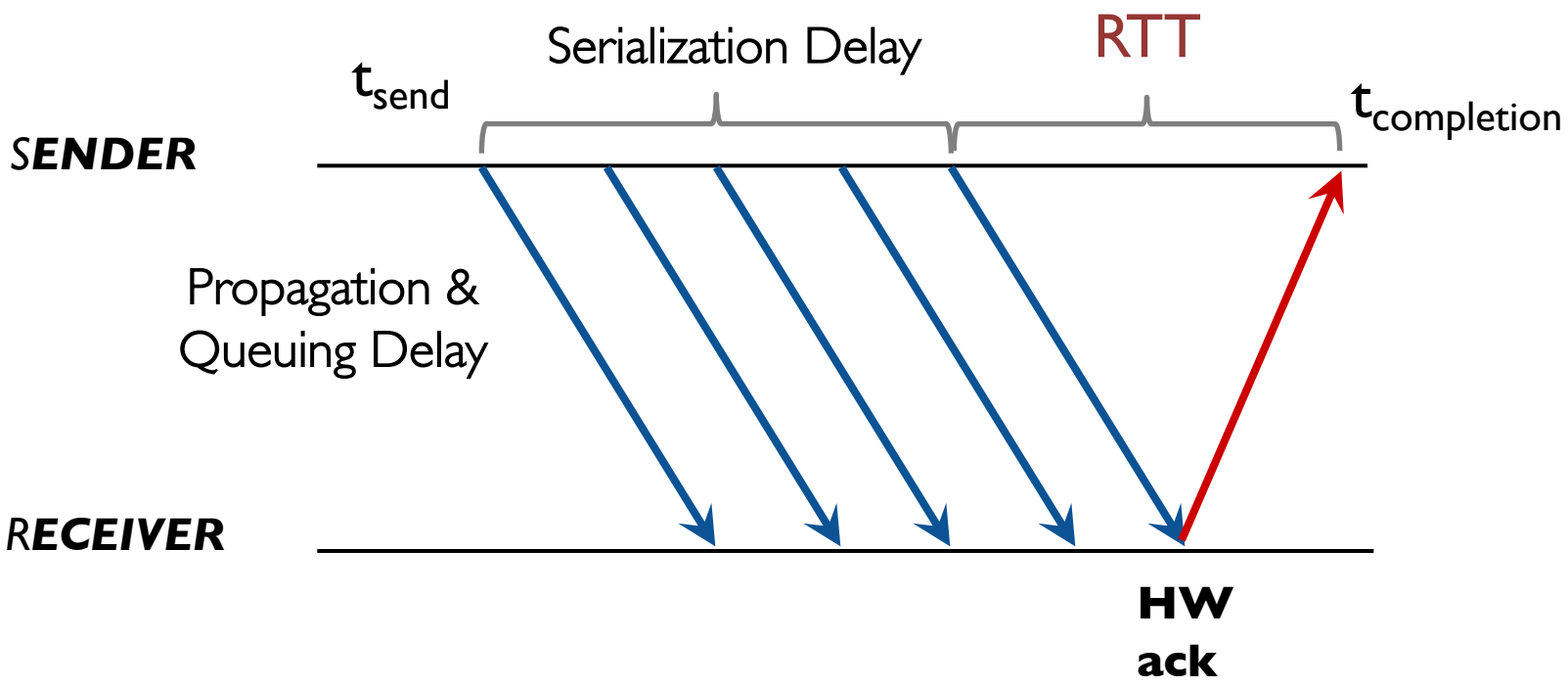


TIMELY Framework

Overview


Data
RTT Measurement Engine
RTT
Rate Computation Engine
Rate
Pacing Engine
Timestamps
Paced Data

RTT Measurement Engine



RTT = $t_{completion} - t_{send}$ – **Serialization Delay**

Rate Computation Engine

On each segment completion event

- Input: RTT sample
- Runs the rate update algorithm
- Output: updated rate

Why do we compute the rate as opposed to a window?

- Segment sizes as high as 64KB
- (32us RTT x 10Gbps) = 40KB window size
- 40KB < 64KB : Window makes no sense

Pacing Engine

Computes the send time of a segment using

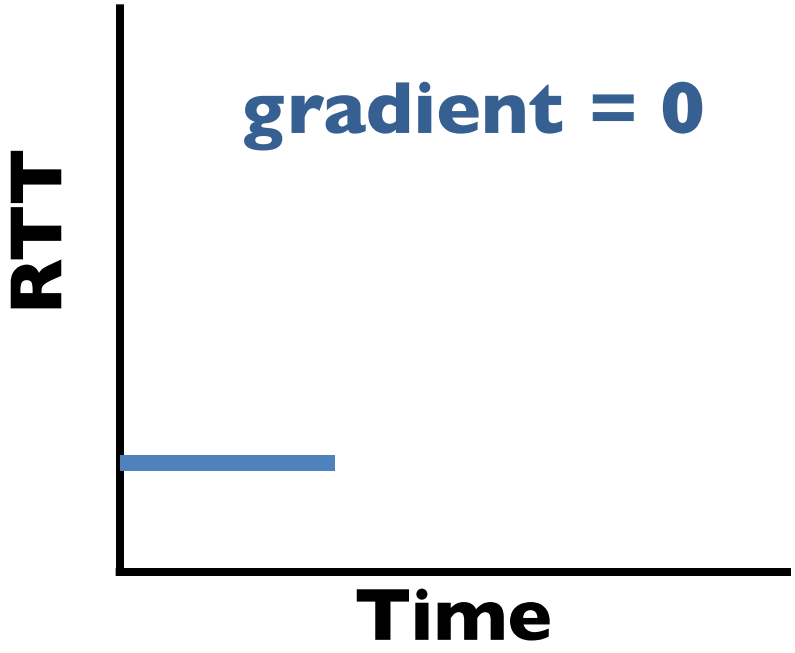
- segment size
- computed flow rate
- time of last transmission

Algorithm Overview

Gradient-based
Increase / Decrease

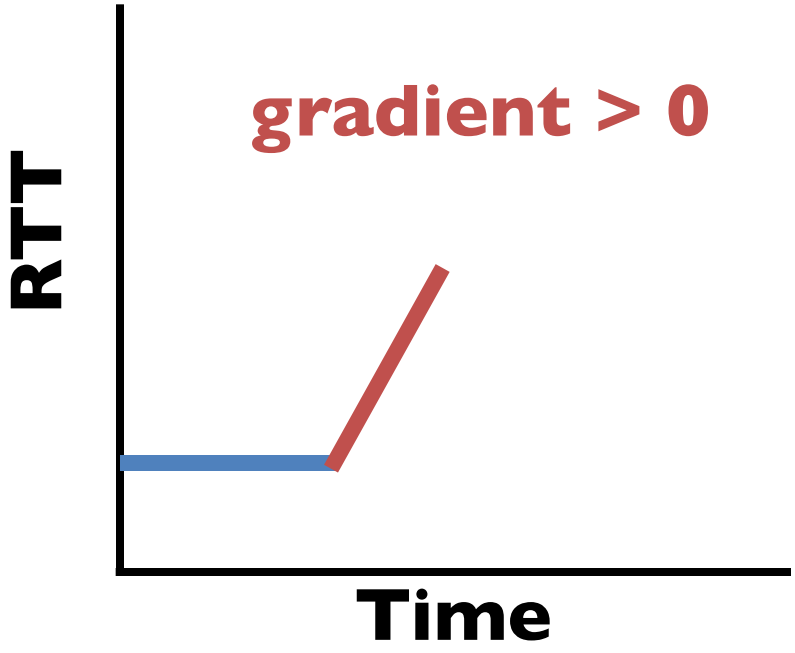
Algorithm Overview

Gradient-based Increase / Decrease



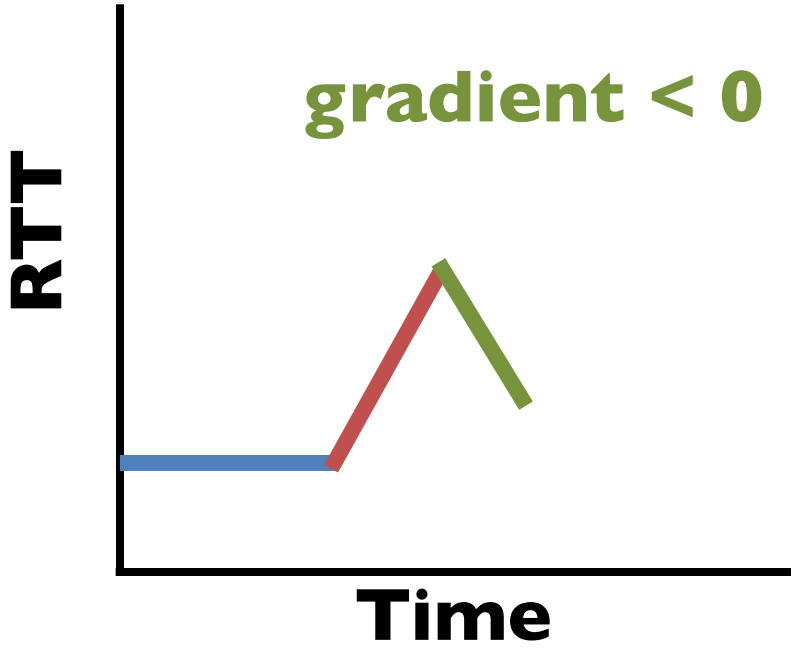
Algorithm Overview

Gradient-based Increase / Decrease



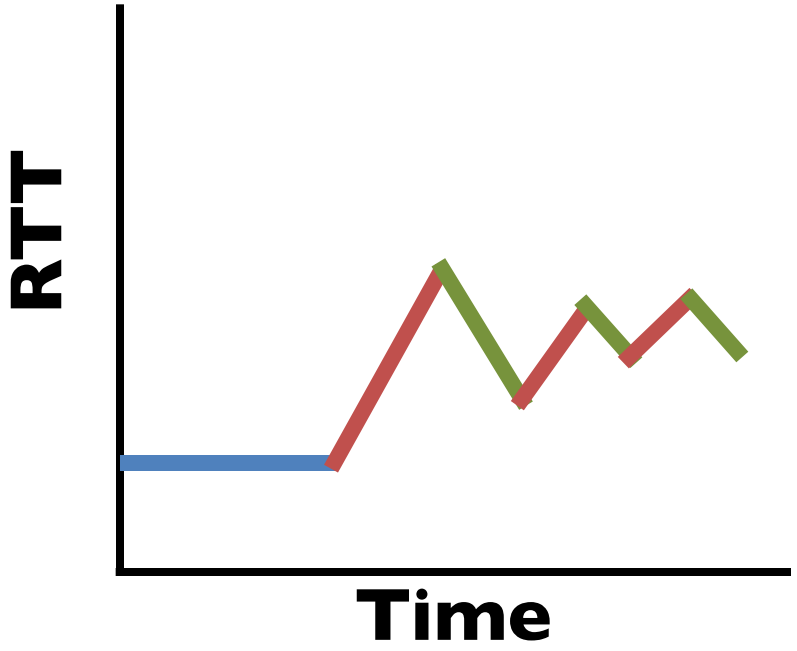
Algorithm Overview

Gradient-based Increase / Decrease

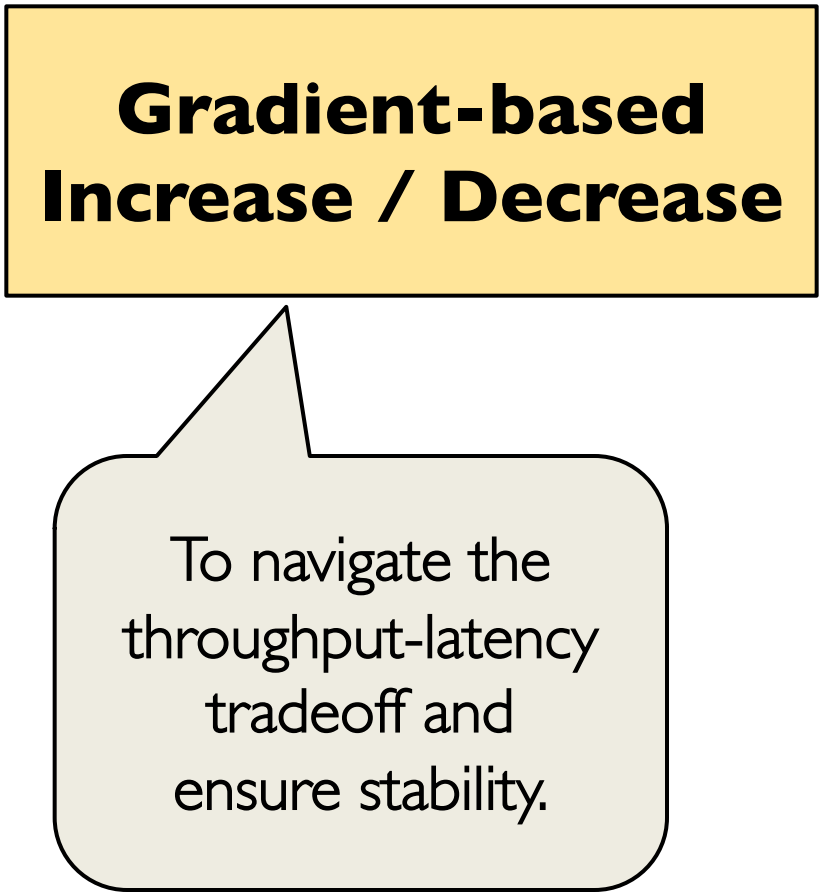


Algorithm Overview

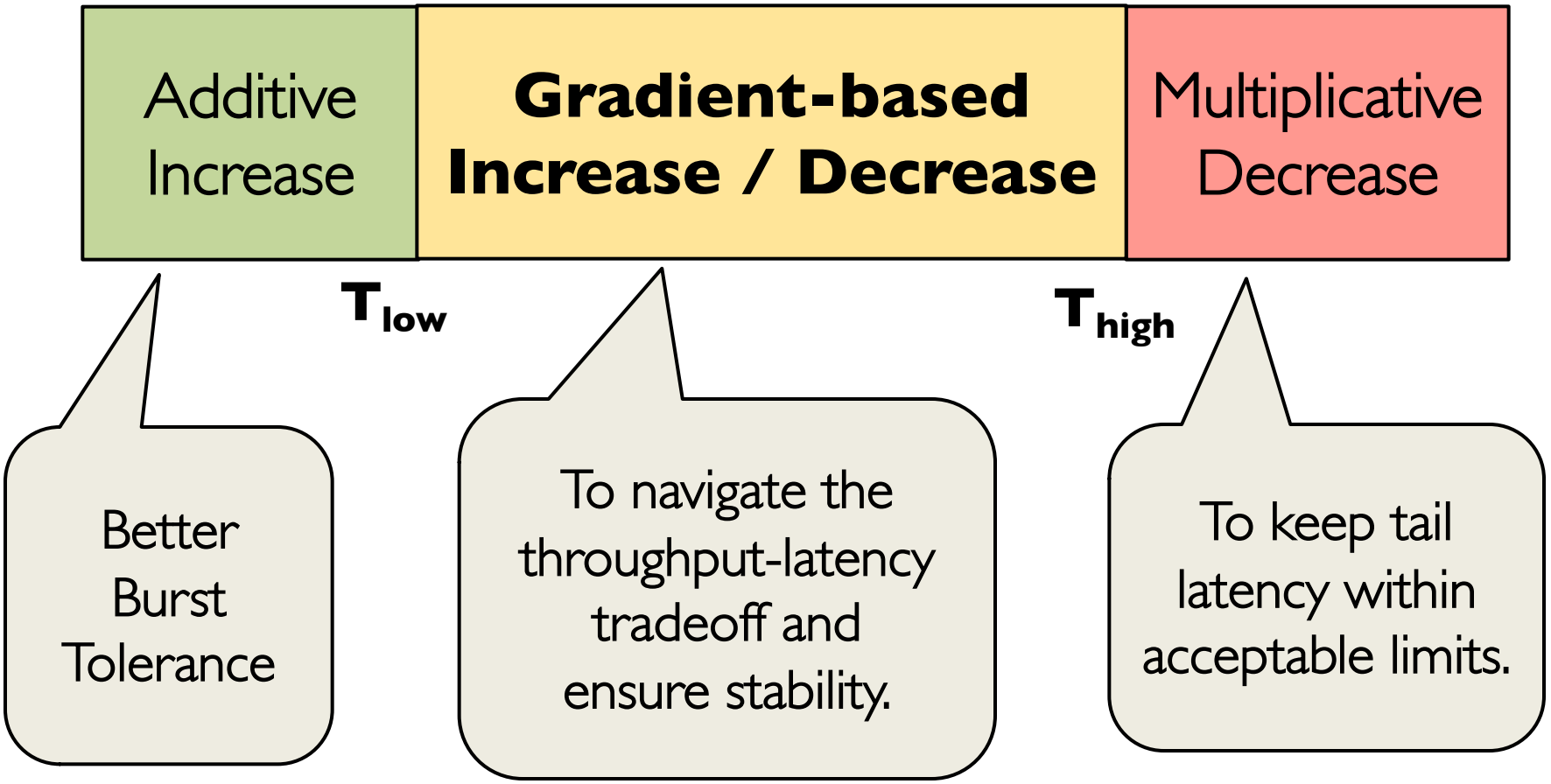
Gradient-based Increase / Decrease



Algorithm Overview



Algorithm Overview

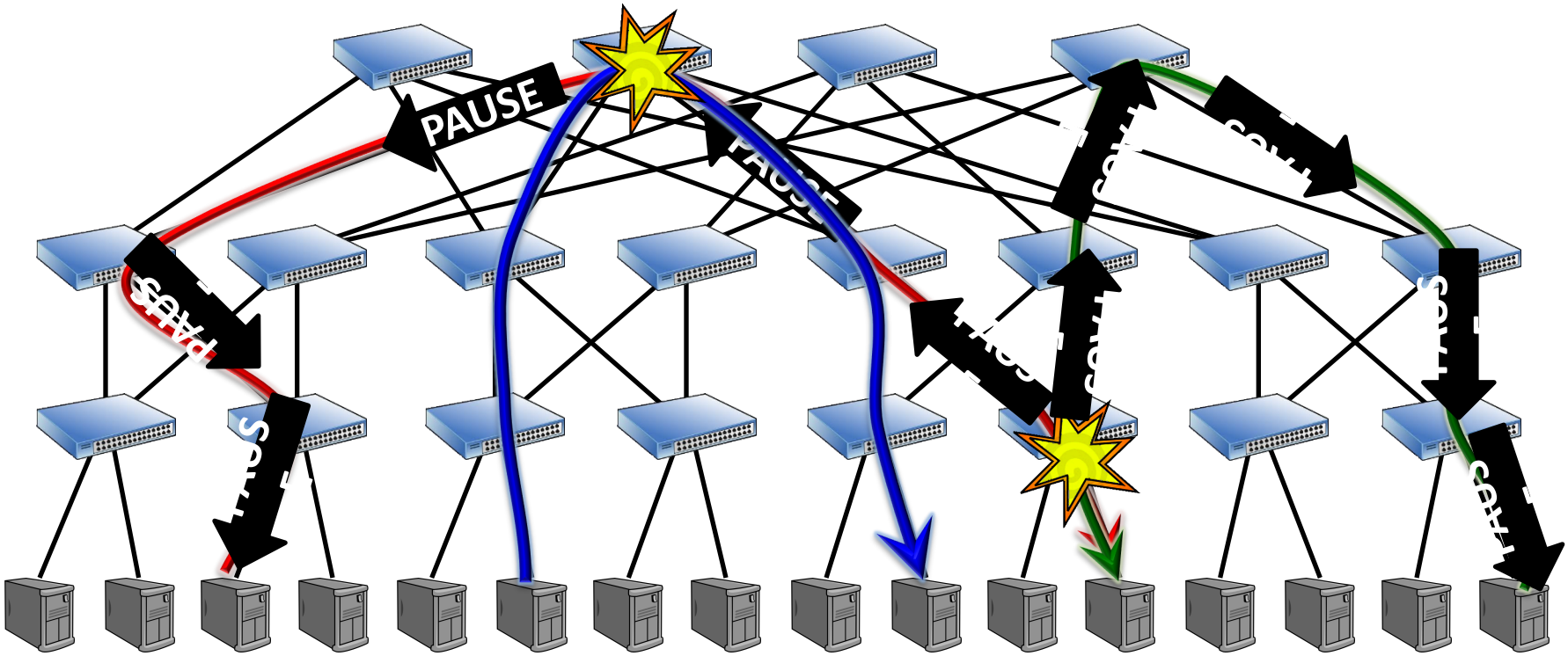


Implementation Set-up

- TIMELY is implemented in the context of RDMA.
  - RDMA write and read primitives used to invoke NIC services.

- Priority Flow Control is enabled in the network fabric.
  - RDMA transport in the NIC is sensitive to packet drops.
  - PFC sends out pause frames to ensure lossless network.

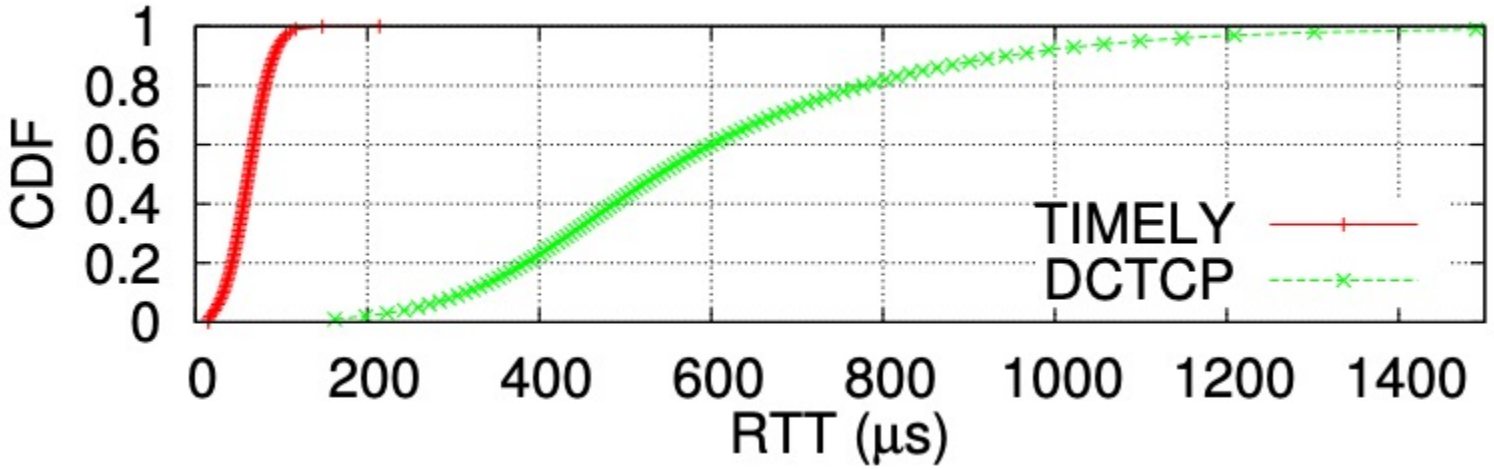
“Congestion Spreading” in Lossless Networks



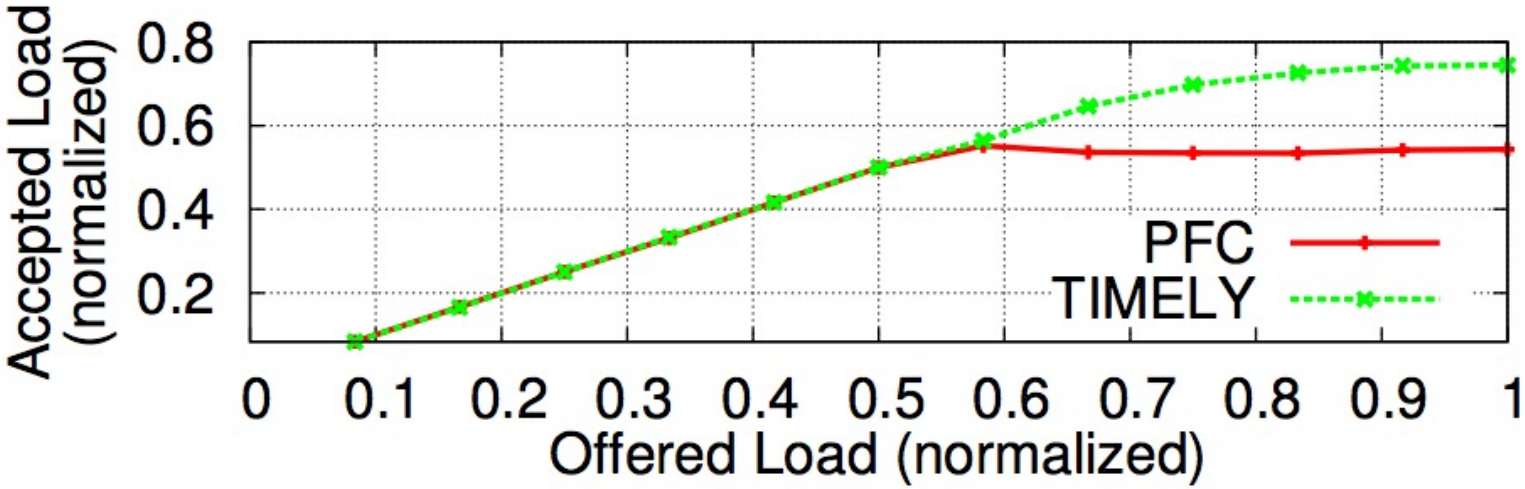
TIMELY vs DCTCP

Metric	DCTCP	FAST*			PFC	TIMELY
		10M	50M	100M		
Total Throughput (Gbps)	19.5	7.5	12.5	17.5	19.5	19.4
Avg. RTT (us)	598	19	120	354	658	61
99-percentile RTT (us)	1490	49	280	460	1036	116

Table 1: Overall performance comparison. FAST* is shown with network buffered traffic parameter in Mbps.



TIMELY vs PFC



TIMELY vs PFC