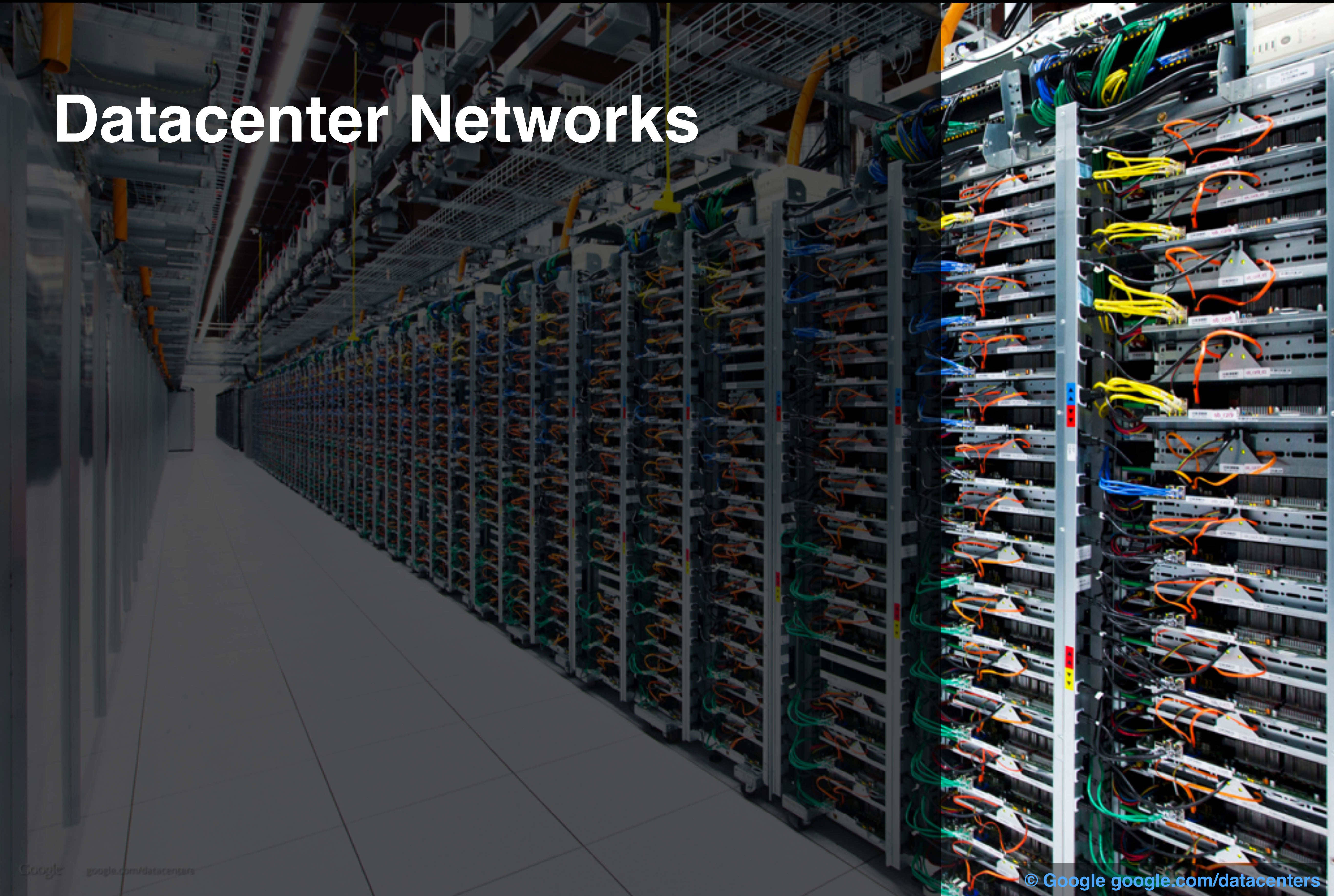
UNIVERSITY OF CAMBRIDGE

Queues don't matter when you can JUMP them

Matthew P. Grosvenor Malte Schwarzkopf Ionel Gog
Andrew W. Moore Robert N. M. Watson Steven Hand Jon Crowcroft

Context

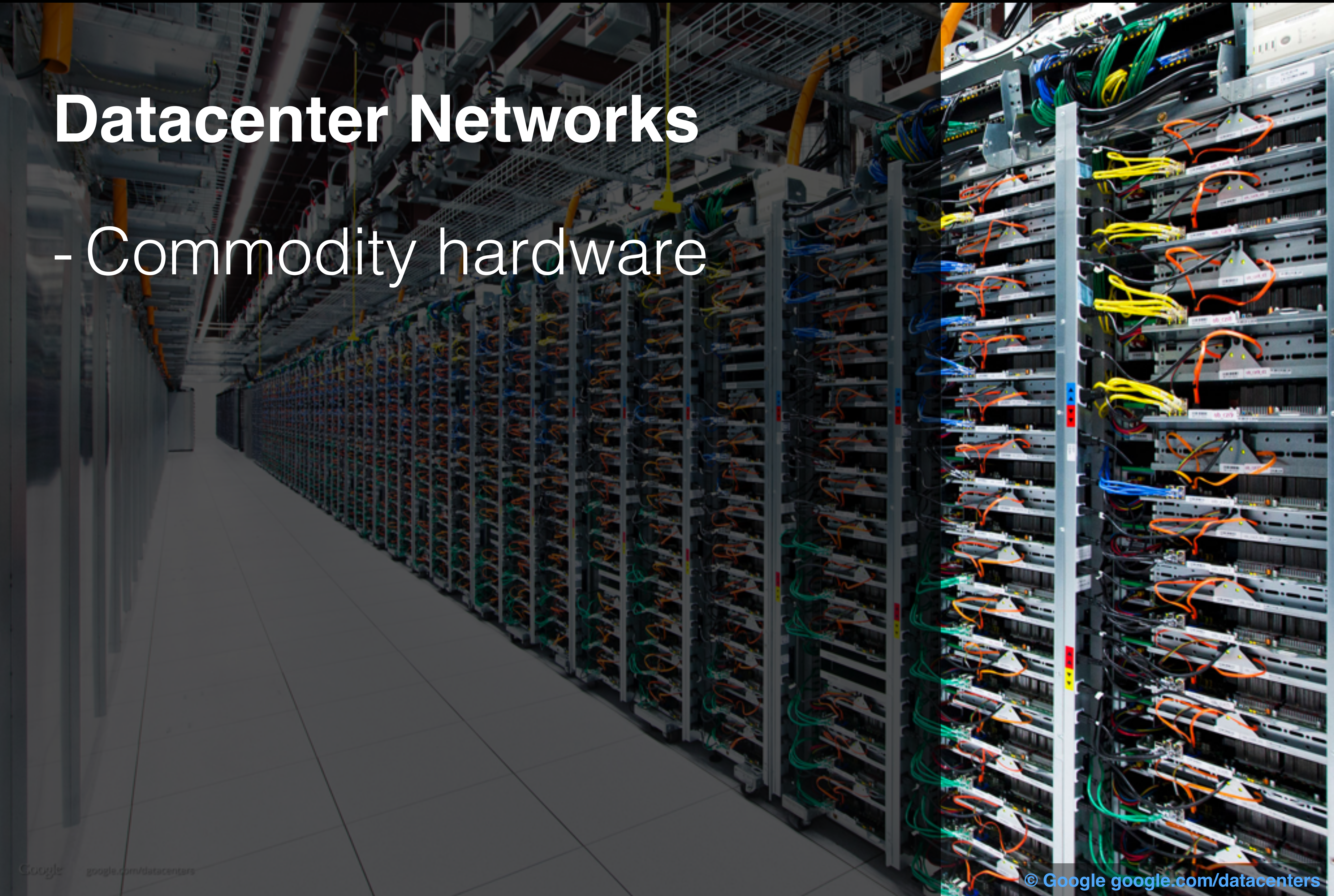
Datacenter Networks



Context

Datacenter Networks

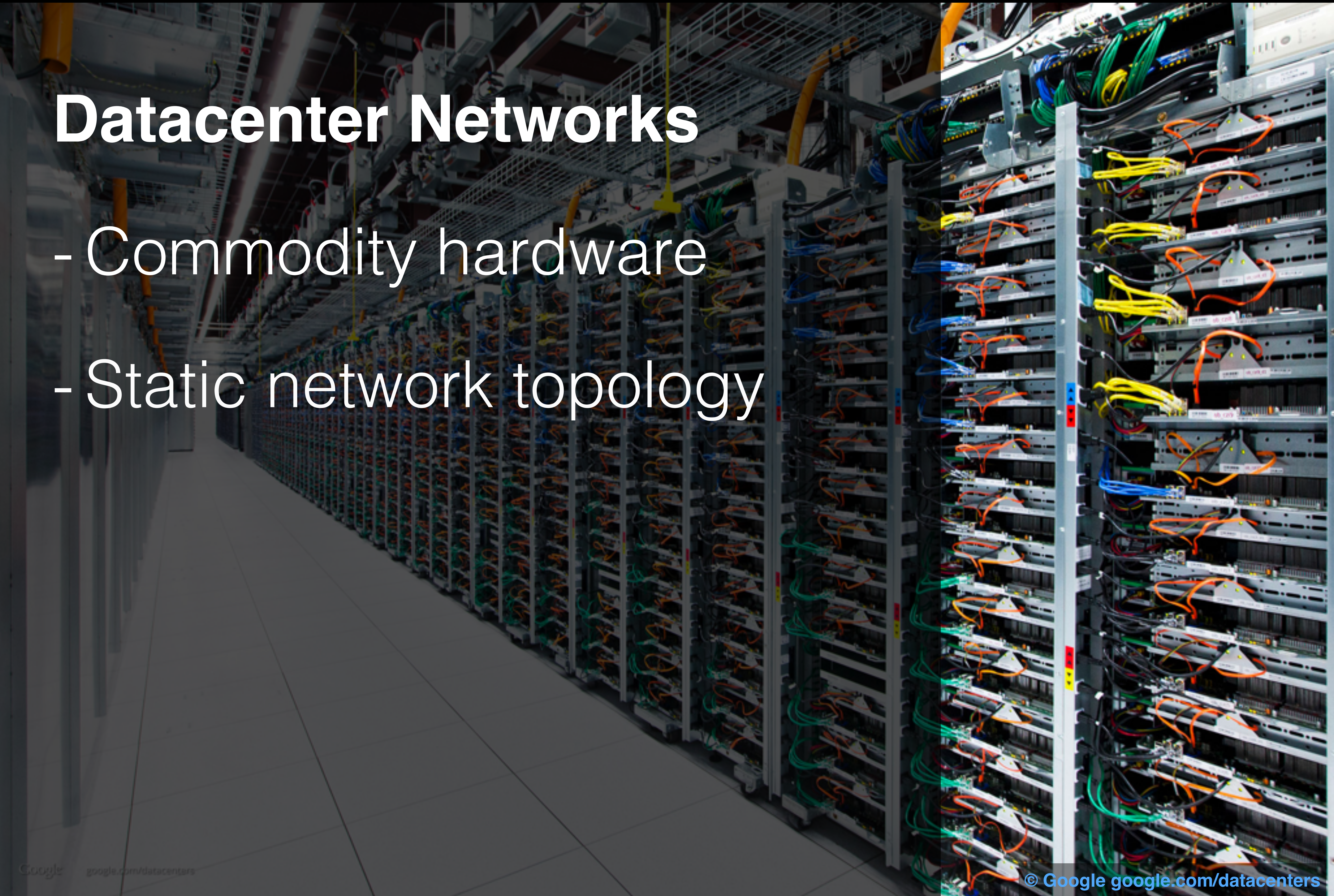
- Commodity hardware



Context

Datacenter Networks

- Commodity hardware
- Static network topology



© Google google.com/datacenters

Context

Datacenter Networks

- Commodity hardware
- Static network topology
- Single administrative domain

Context

Datacenter Networks

- Commodity hardware
- Static network topology
- Single administrative domain
- Some level of cooperation

Context

Datacenter Networks

- Commodity hardware
- Static network topology
- Single administrative domain
- Some level of cooperation
- Statistically Multiplexed

Context

Datacenter Networks

- Commodity hardware
- Static network topology
- Single administrative domain
- Some level of cooperation
- **Statistically Multiplexed**

Context

Datacenter Networks

- Commodity hardware
- Static
- Si ative domain
- evel of cooperation
- **Statistically Multiplexed**

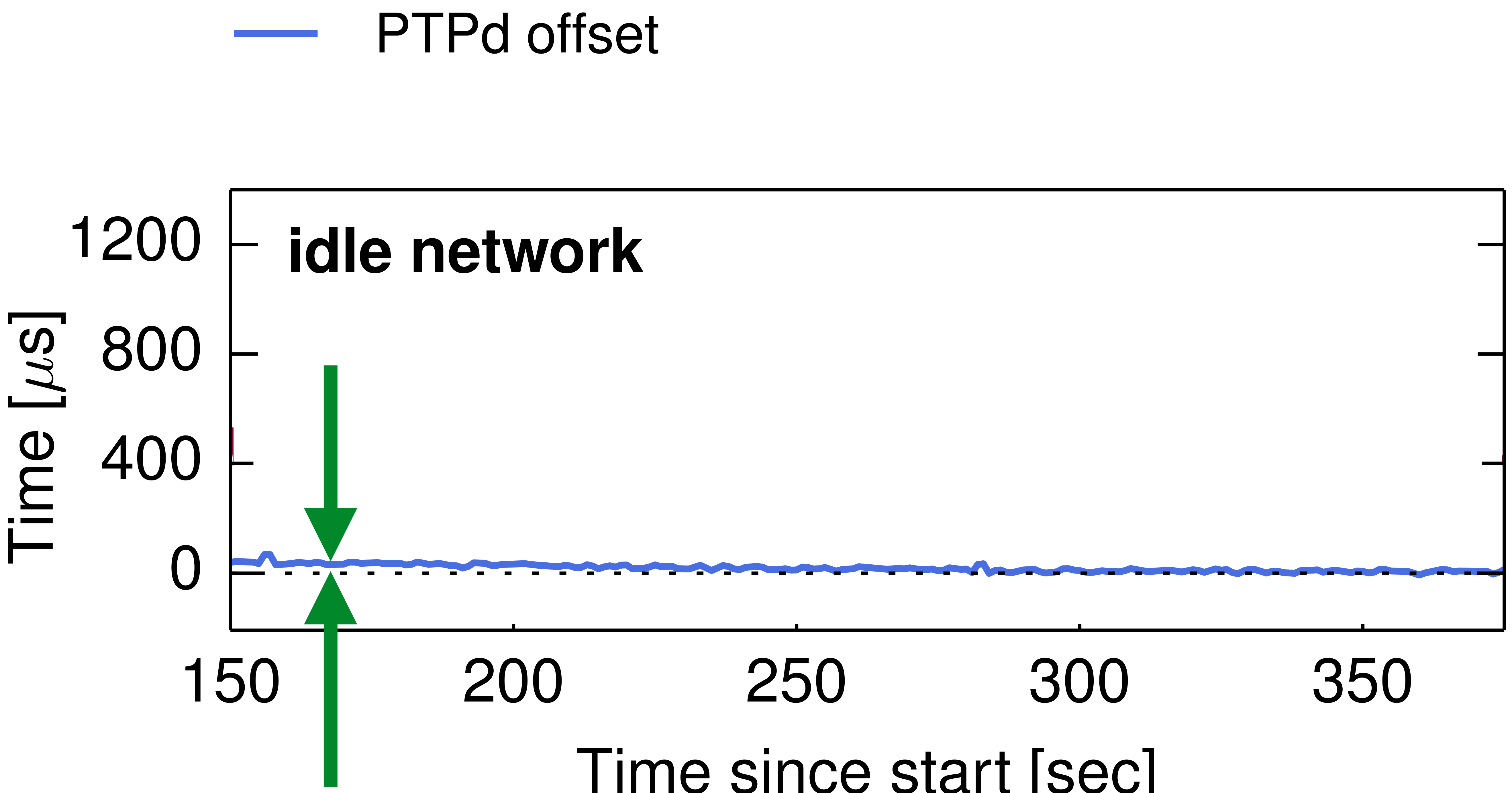
NO ^latency GUARANTEES

Application impact

Illustrative experiment

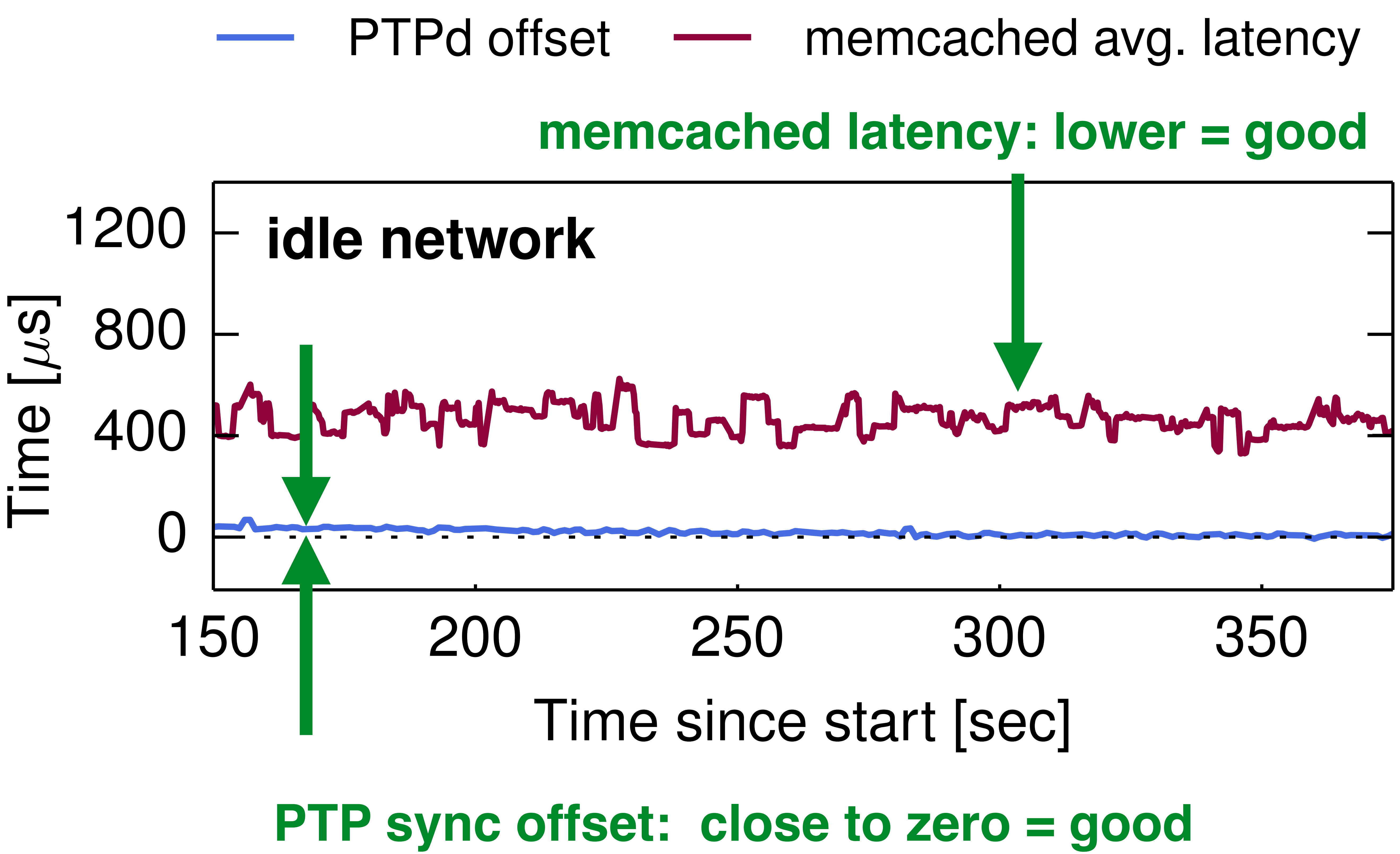
- 12 node 10G test cluster
  - 8 nodes *Hadoop MR*
  - 2 nodes *PTPd*
  - 2 nodes *memcached*

Application impact



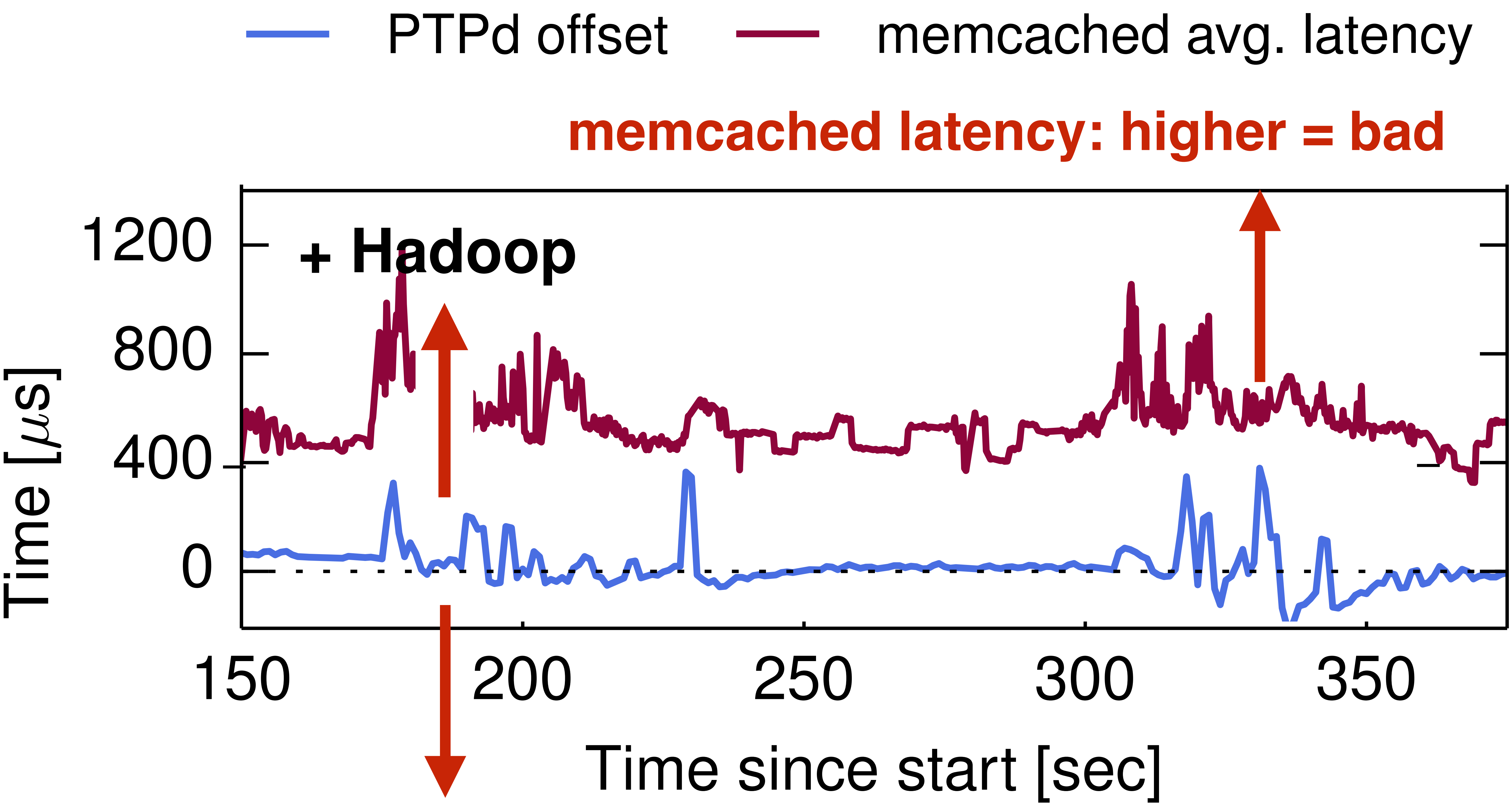
PTP sync offset: close to zero = good

Application impact



Application impact

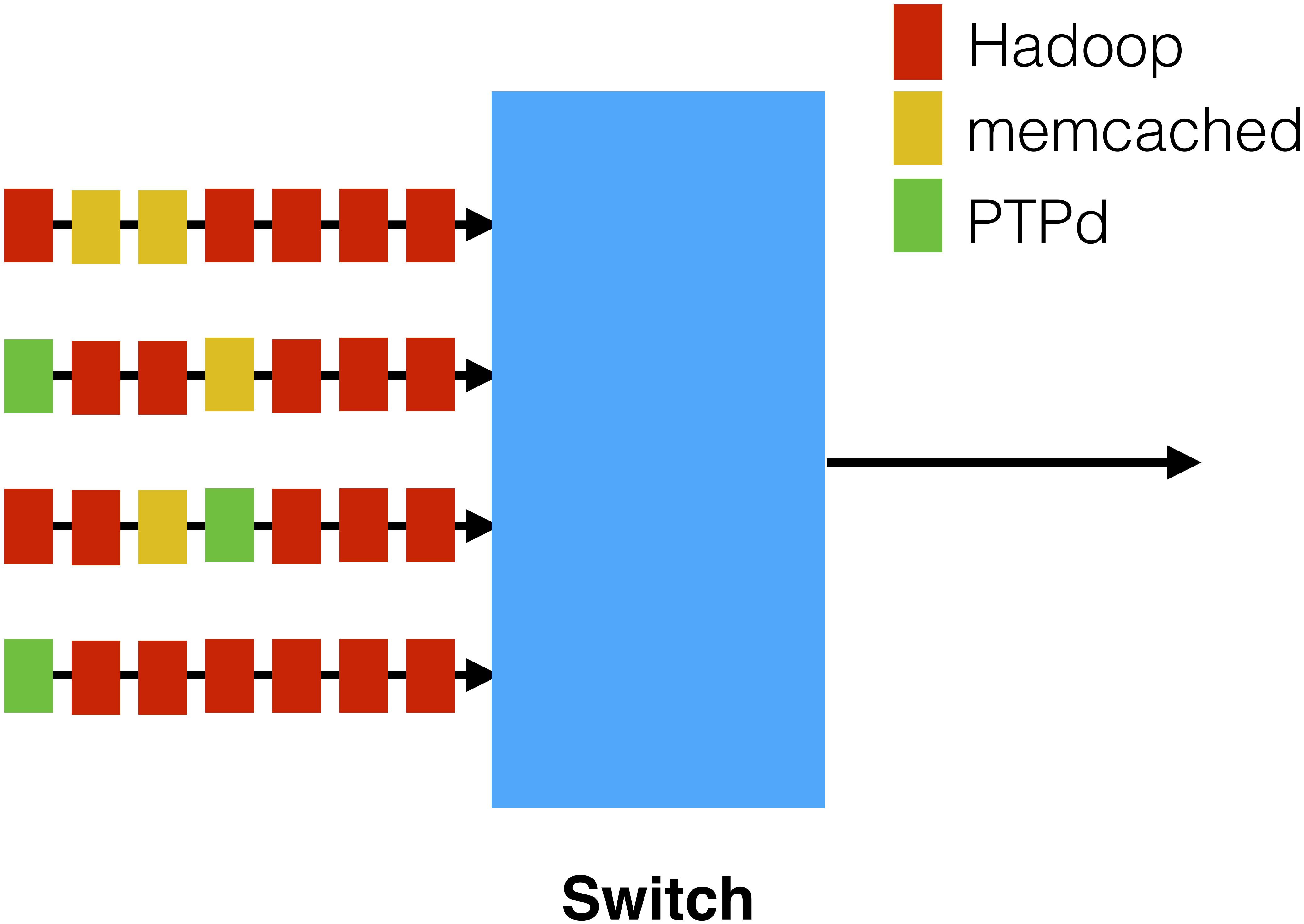
memcached latency: higher = bad



PTP sync offset: away from zero = bad

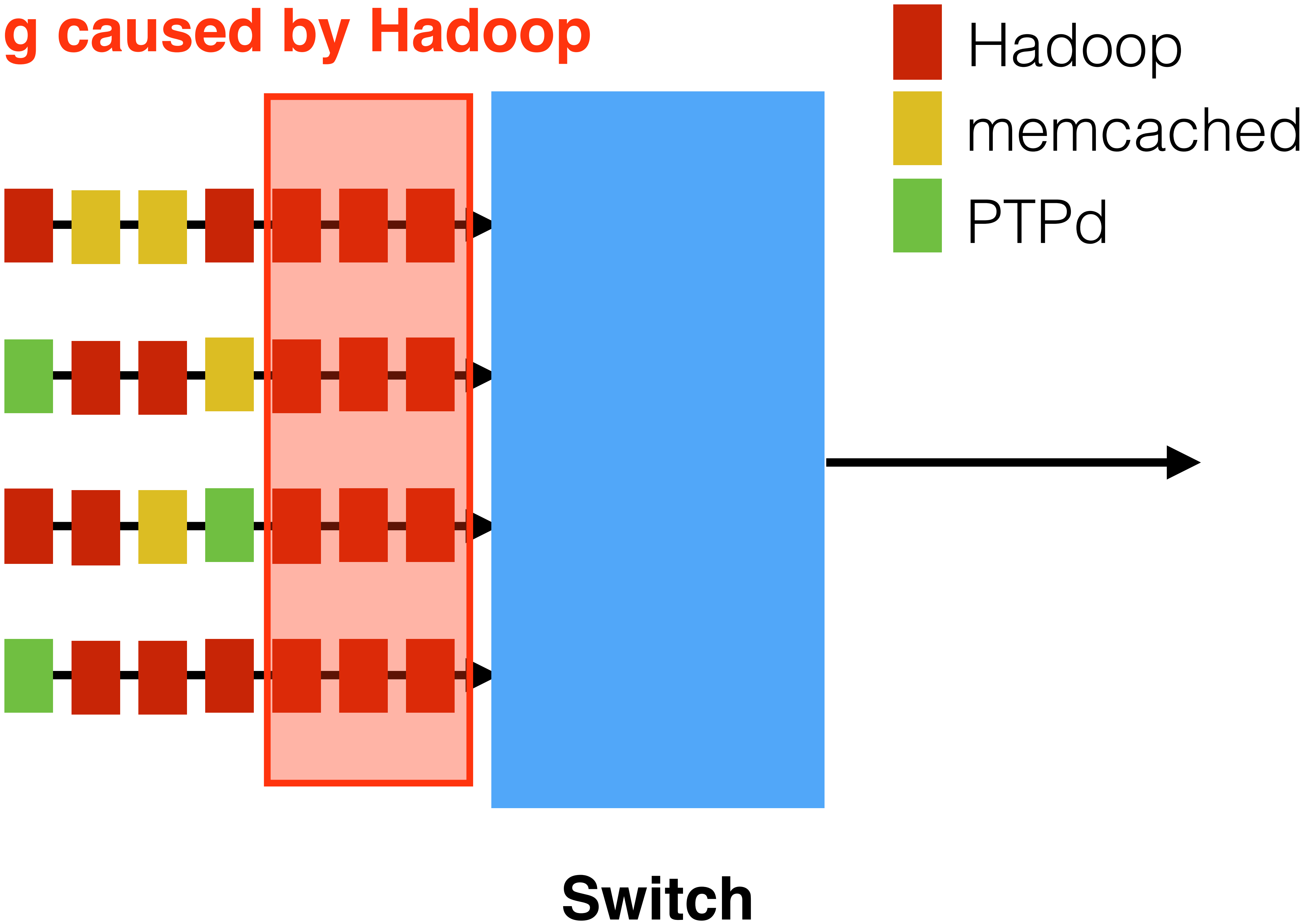
What's the problem?

Network Interference

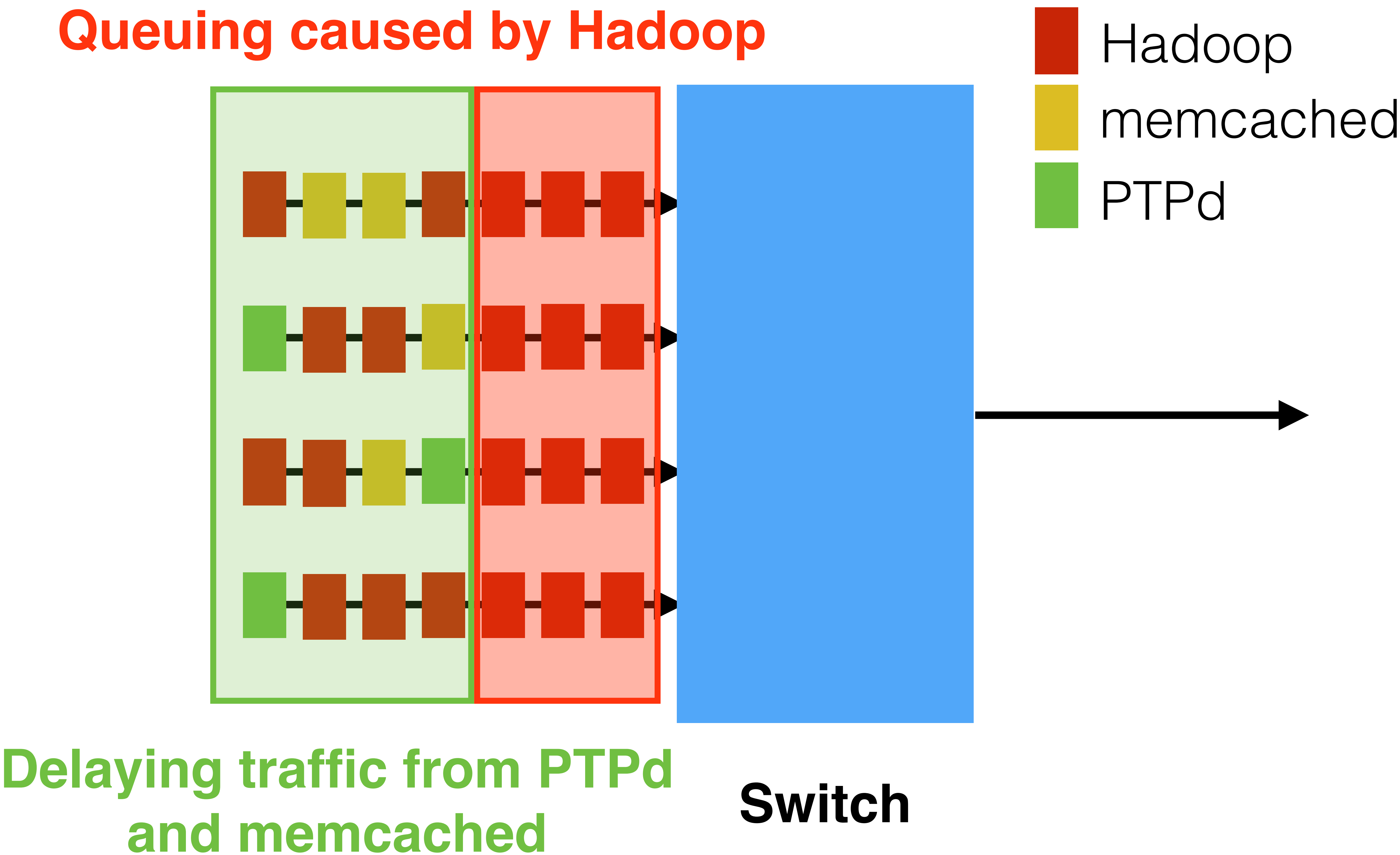


Network Interference

Queuing caused by Hadoop



Network Interference



Key Idea

Network Interference:

Congestion from one application causes queuing that delays traffic from another* application.

*possibly related

Solving network interference?

Borrow some old ideas

Packet by Packet Generalised Processor Sharing (PGPS)

(Weighted) Fair Queuing (WFQ)

Differentiated Service Classes (diff-serv)

Parekh-Gallager Theorem

Solving network interference?

Borrow some old ideas

- Packet by Packet Generalised Processor Sharing (PGPS)
- (Weighted) Fair Queuing (WFQ)
- Differentiated Service Classes (diff-serv)
- Parekh-Gallager Theorem

Apply in a new context : Datacenters

Opportunities & Constraints

Datacenter Opportunities

Opportunities & Constraints

Datacenter Opportunities

- Static network
- Single admin domain
- Cooperation

Opportunities & Constraints

Datacenter Opportunities

- Static network
- Single admin domain
- Cooperation

Deployability Constraints

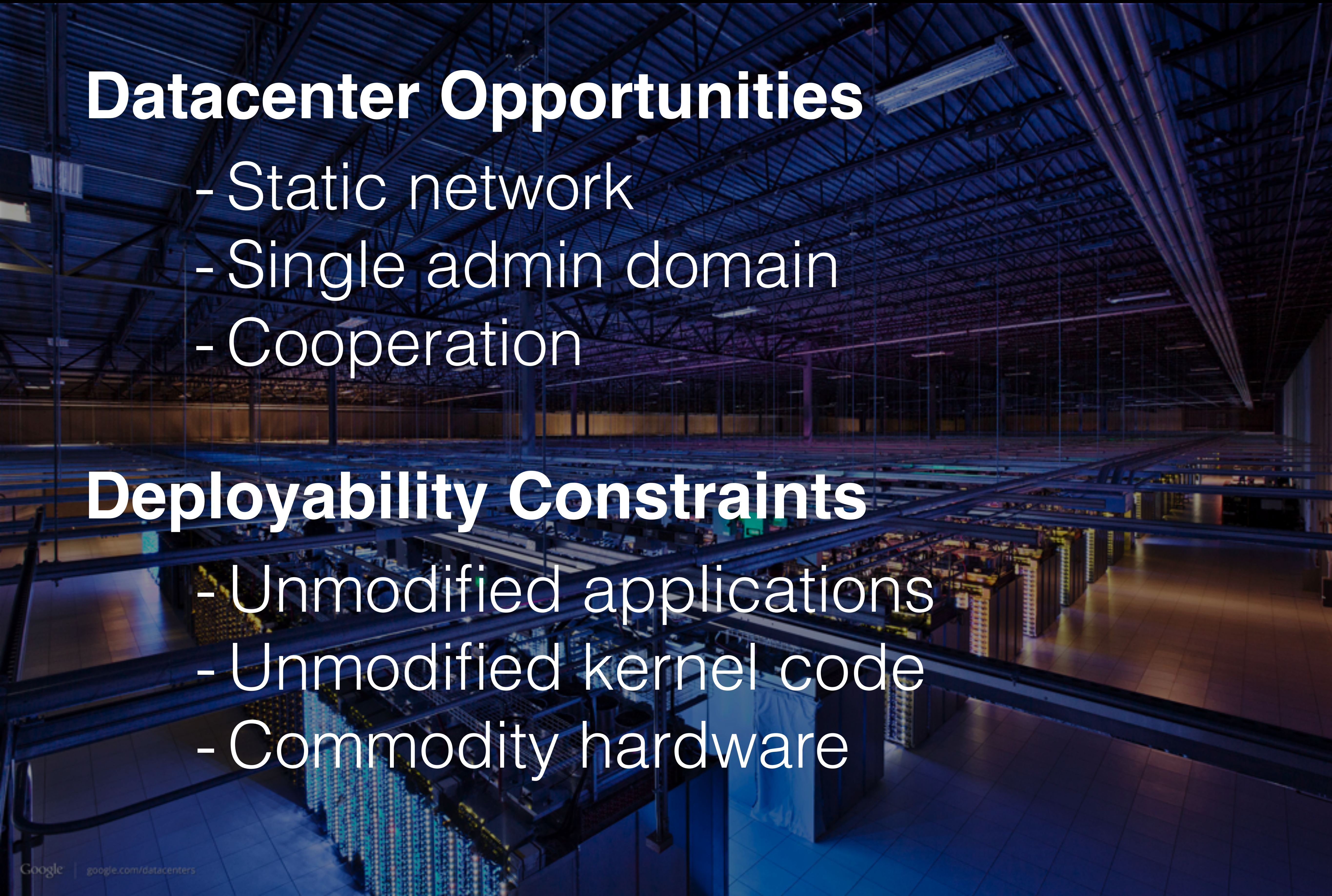
Opportunities & Constraints

Datacenter Opportunities

- Static network
- Single admin domain
- Cooperation

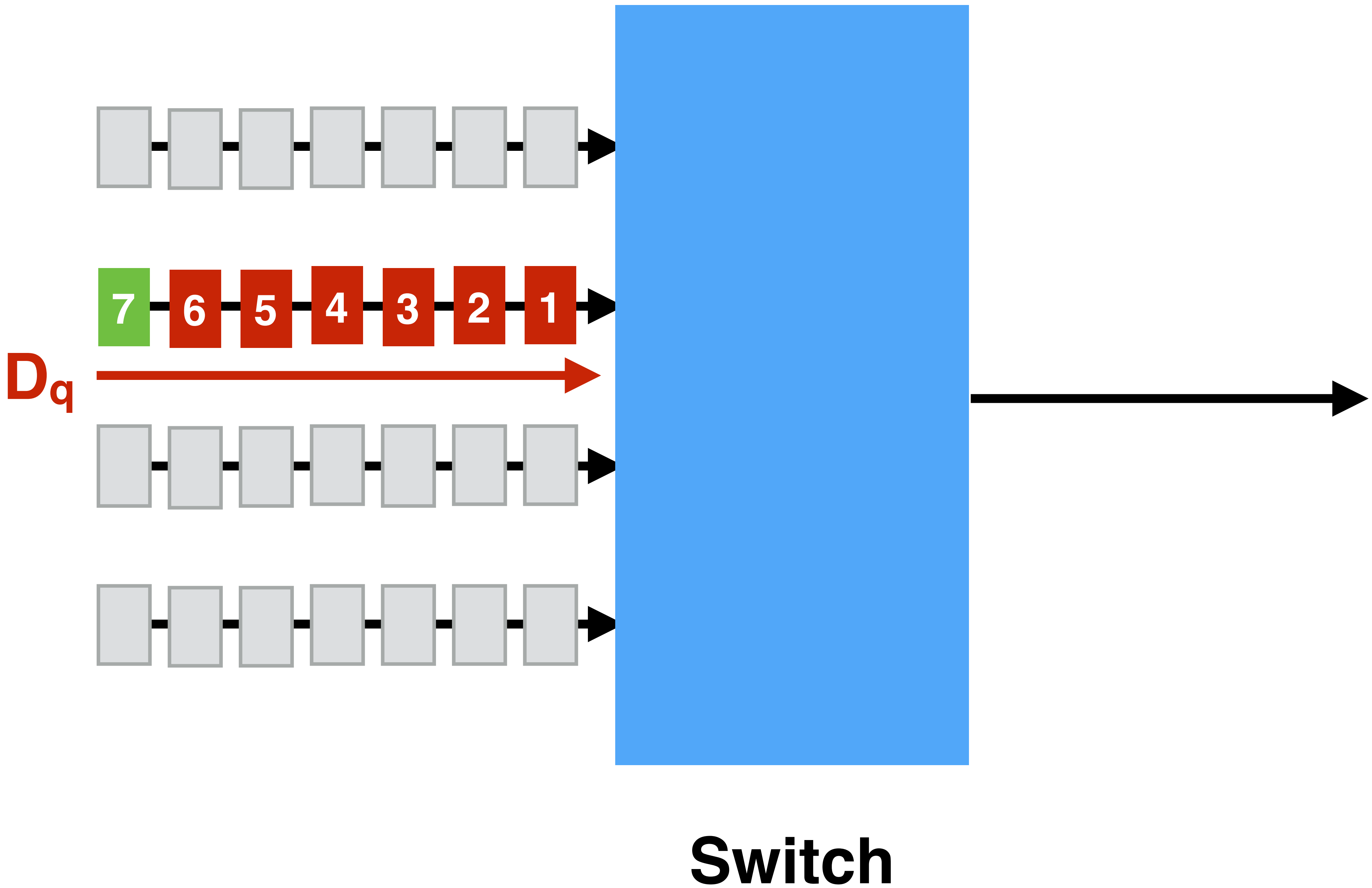
Deployability Constraints

- Unmodified applications
- Unmodified kernel code
- Commodity hardware



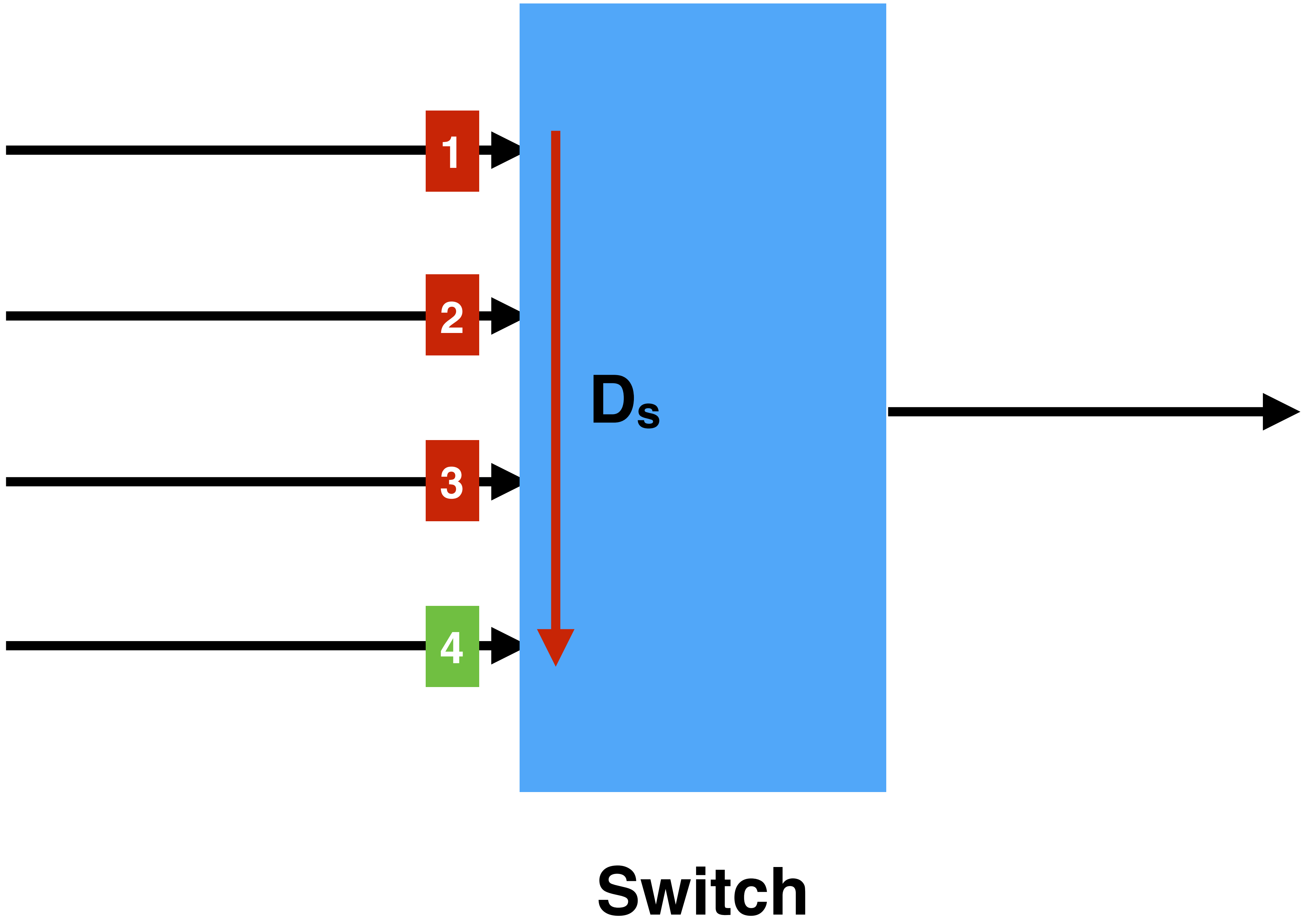
Understanding delays

Delay type I - Queuing Delay (D_q)



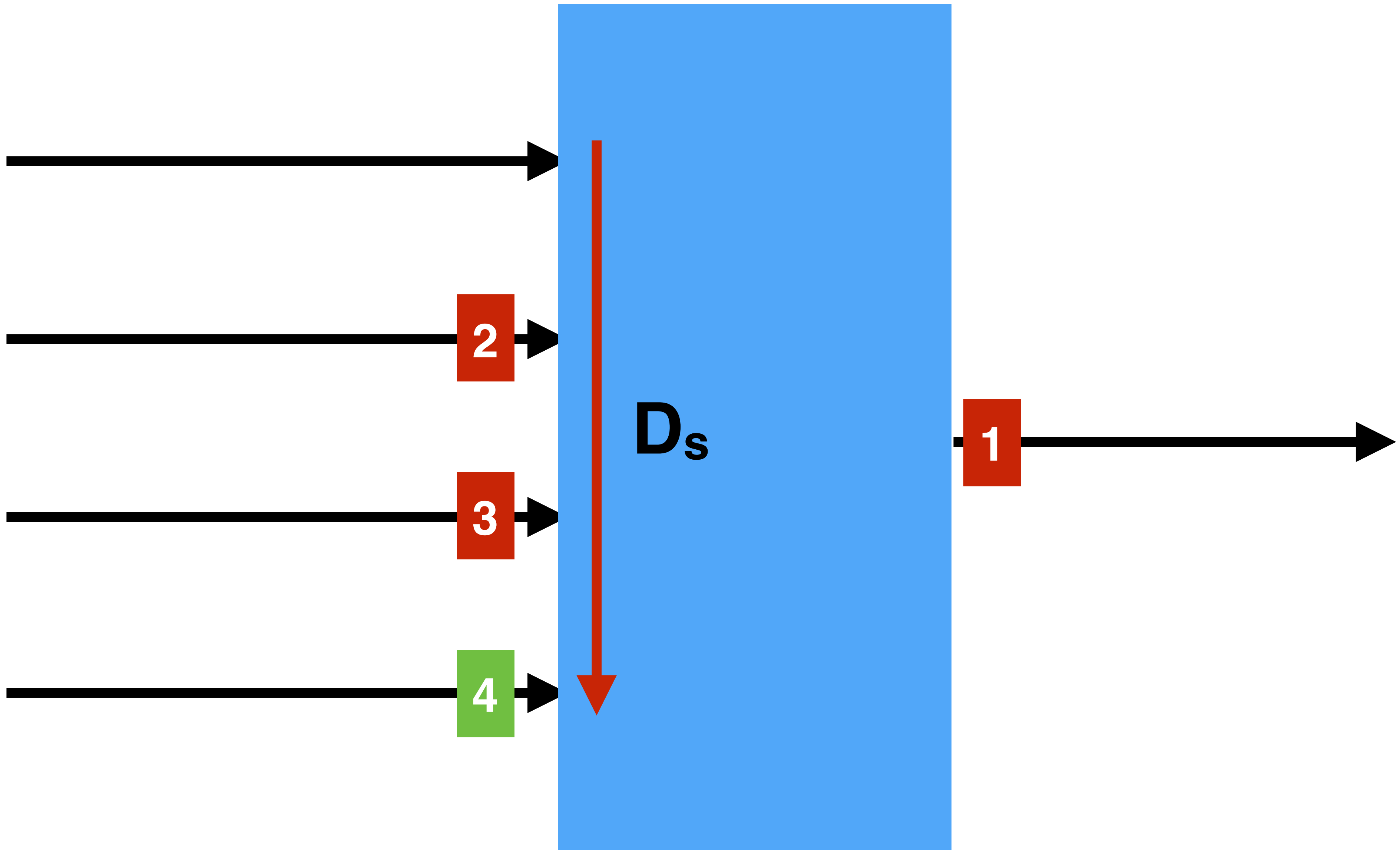
Understanding delays

Delay type II - Servicing Delay (D_s)



Understanding delays

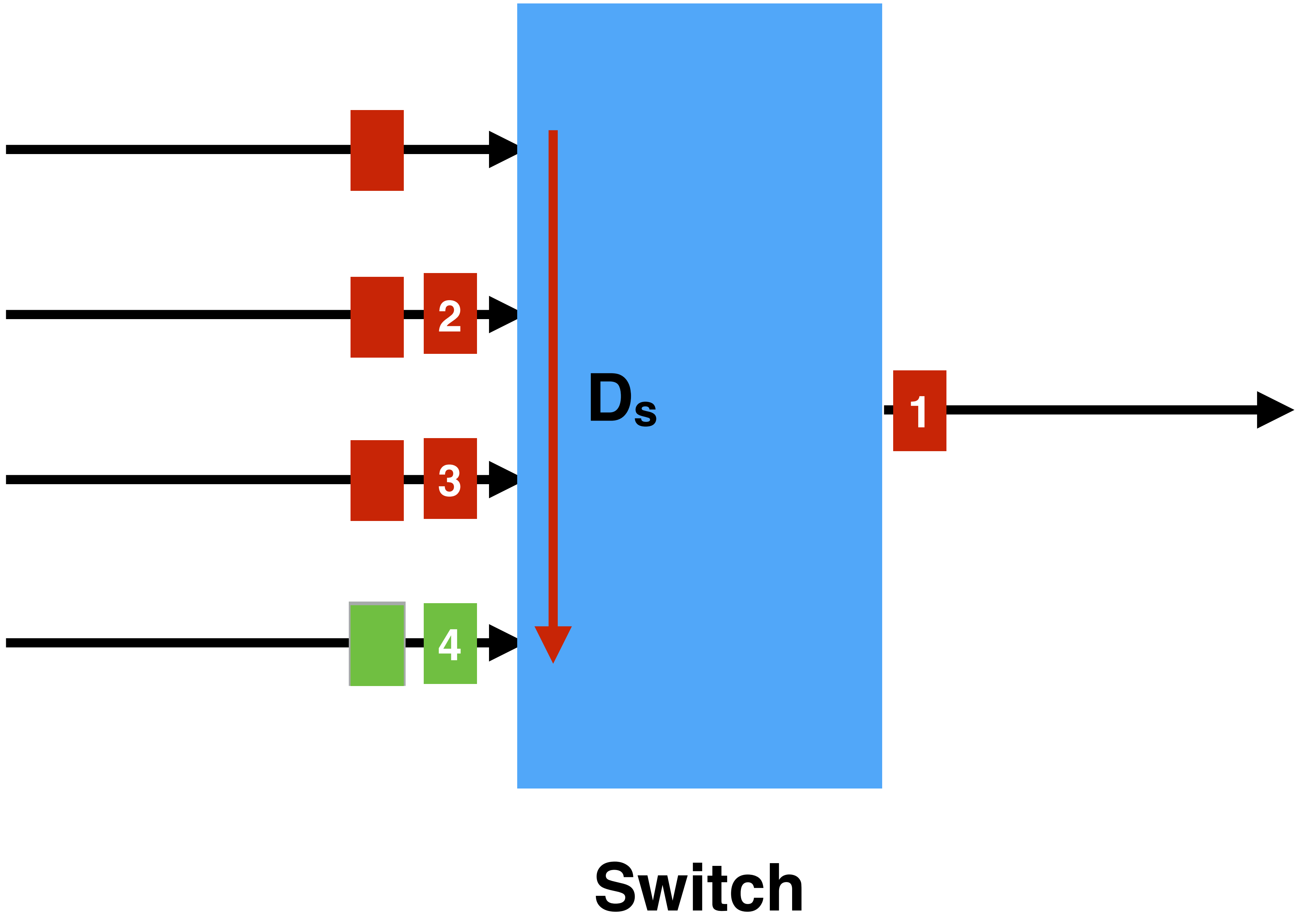
Delay type II - Servicing Delay (D_s)



Switch

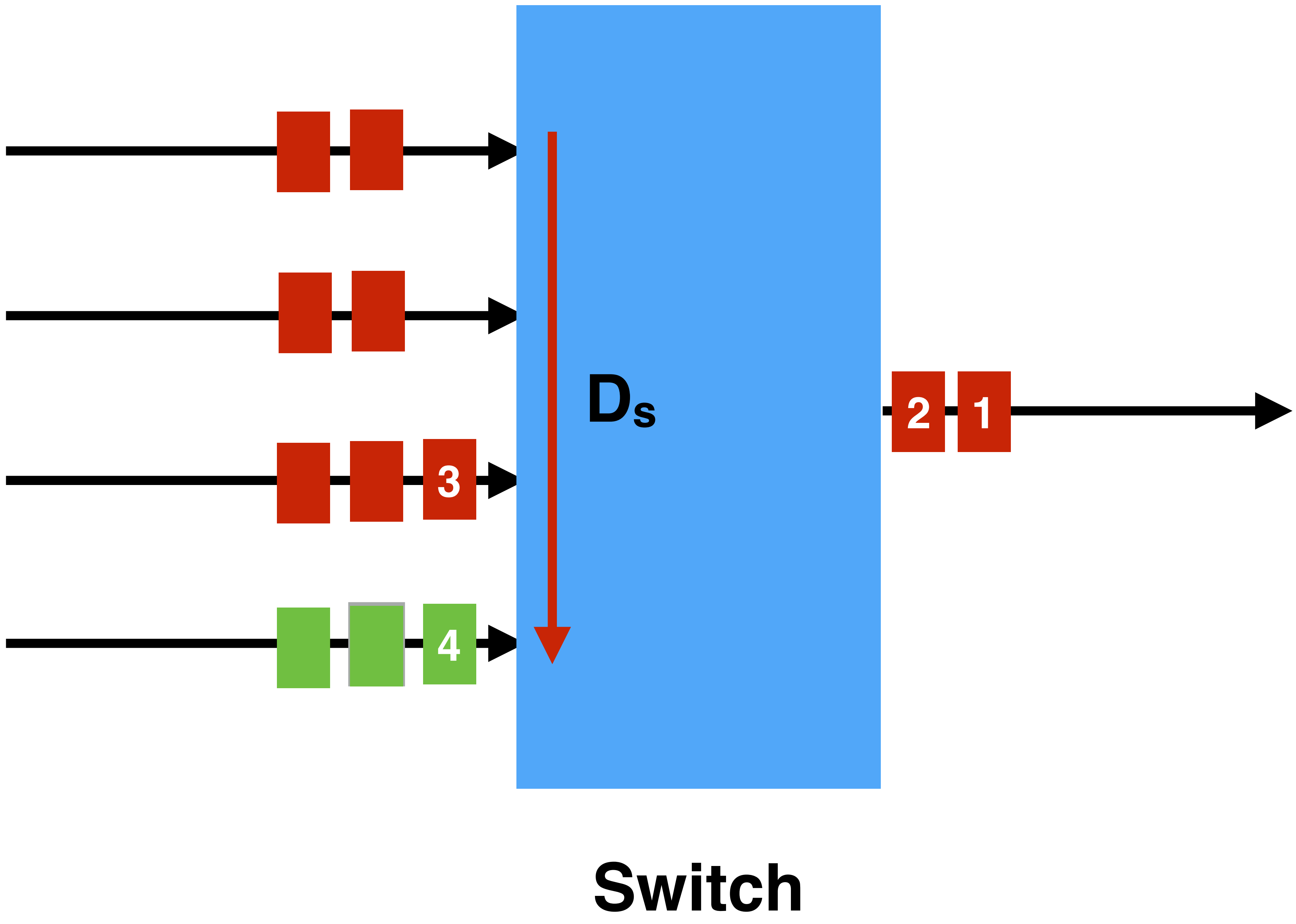
Understanding delays

Delay type II - Servicing Delay (D_s)



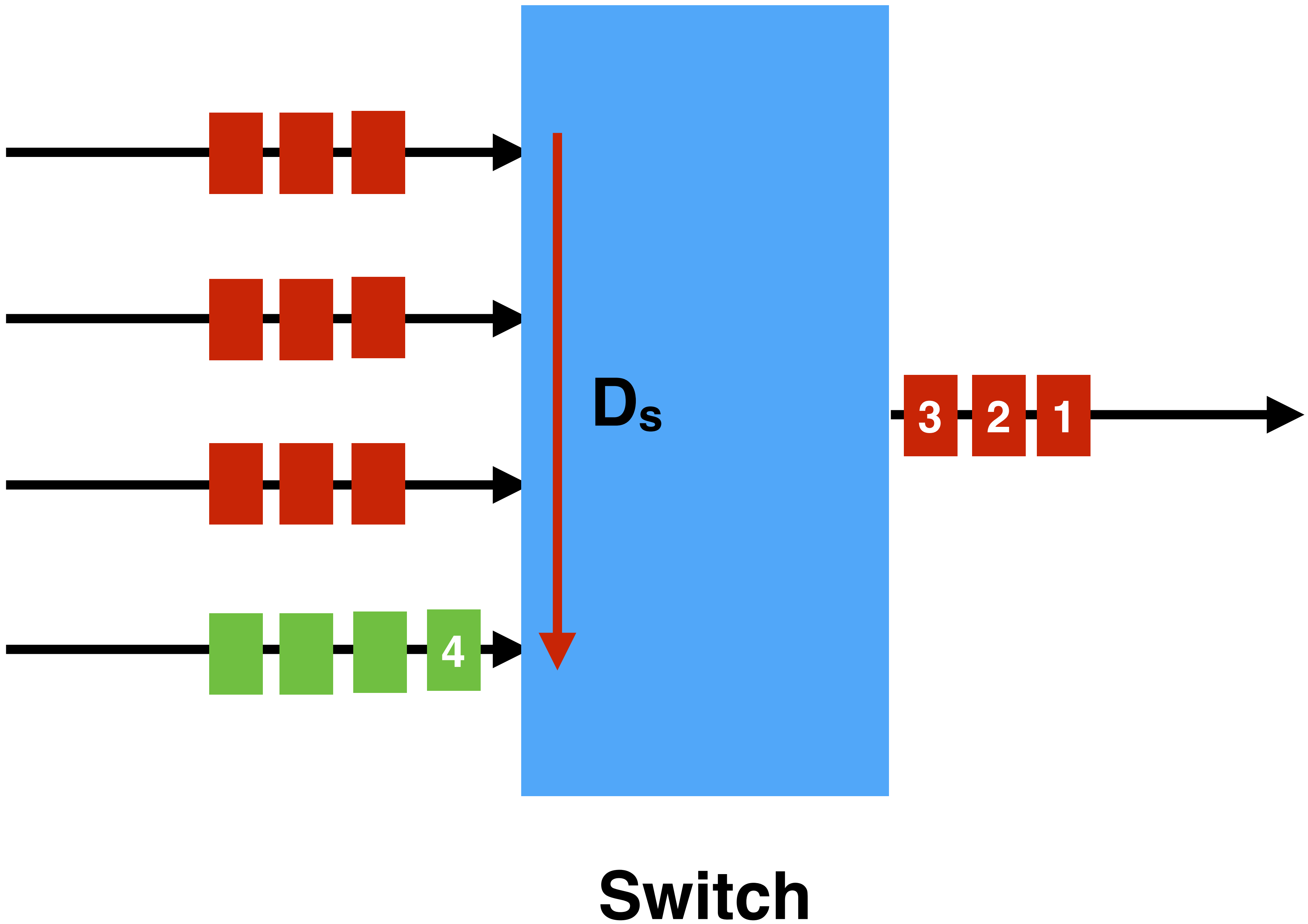
Understanding delays

Delay type II - Servicing Delay (D_s)

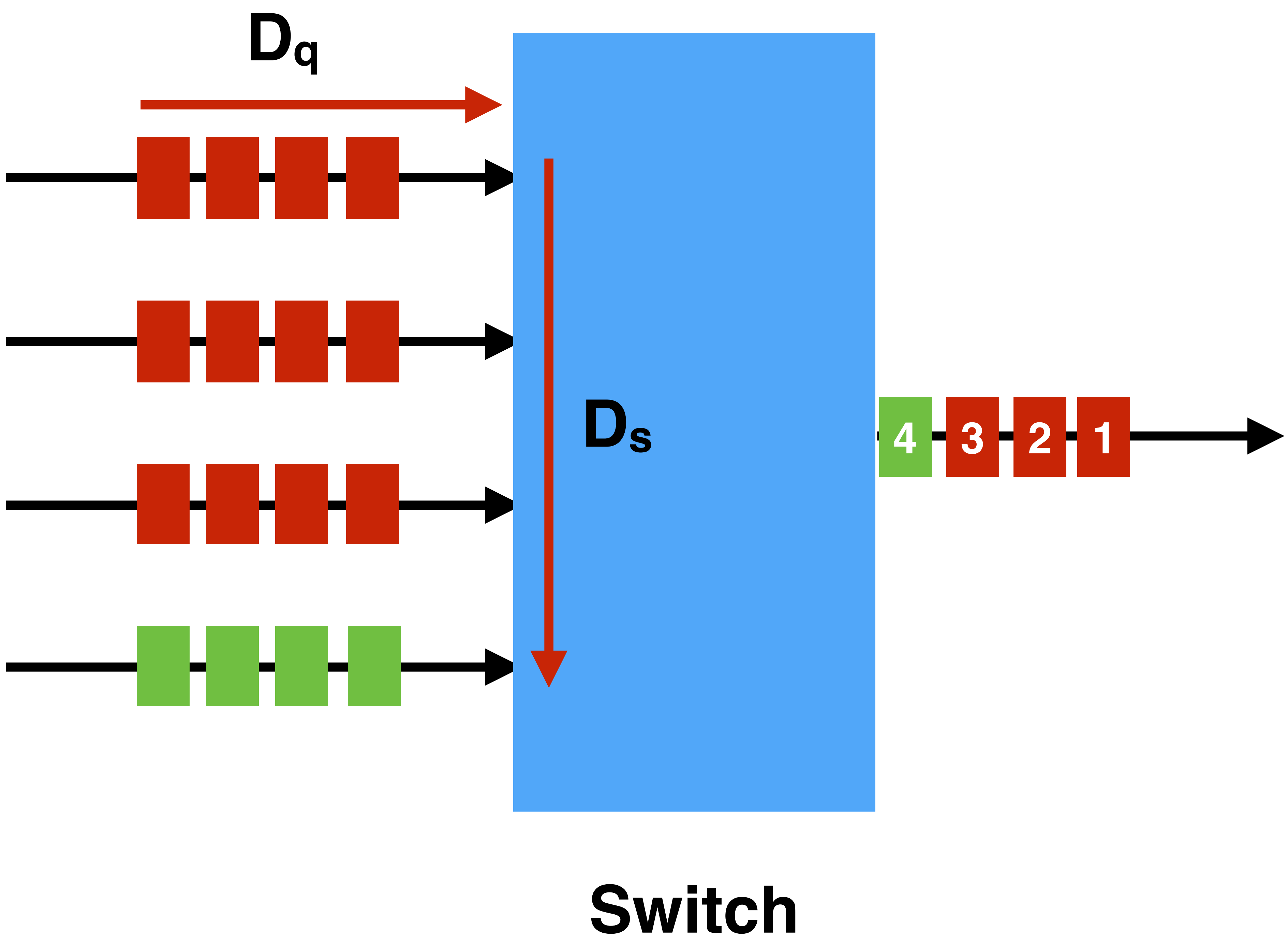


Understanding delays

Delay type II - Servicing Delay (D_s)

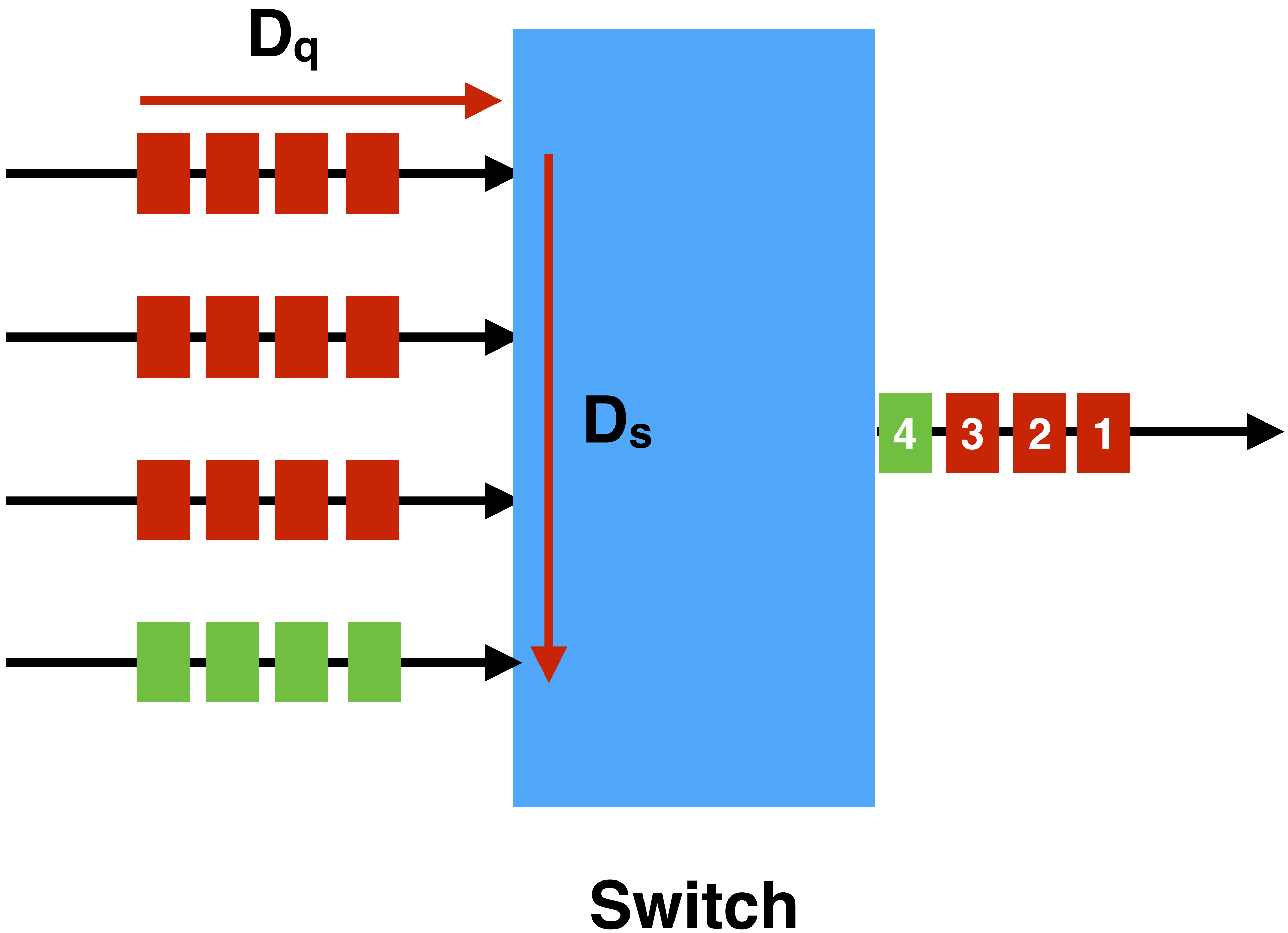


Understanding delays

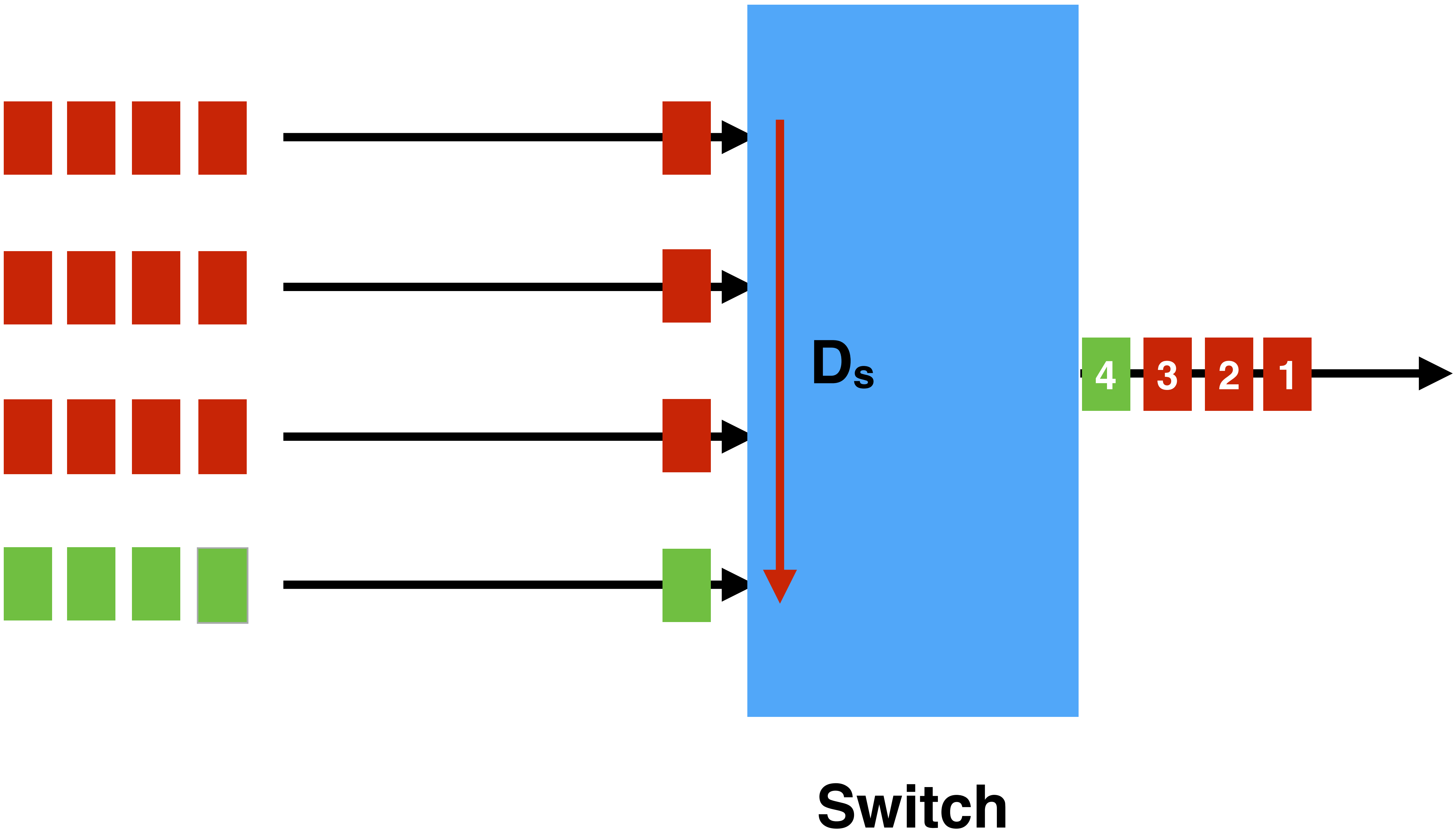


Understanding delays

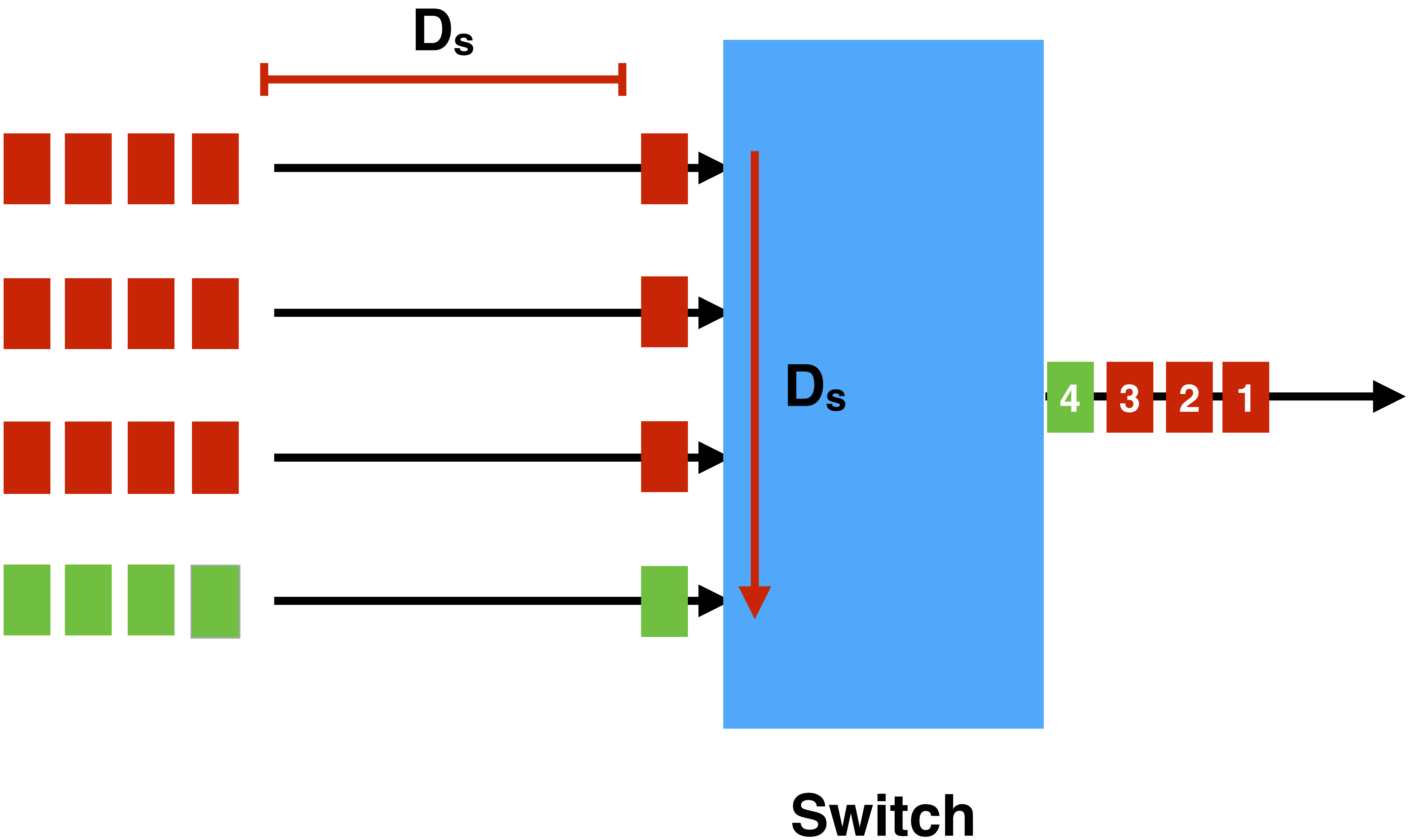
Servicing delay causes queuing delay



Eliminating Queuing Delay



Eliminating Queuing Delay

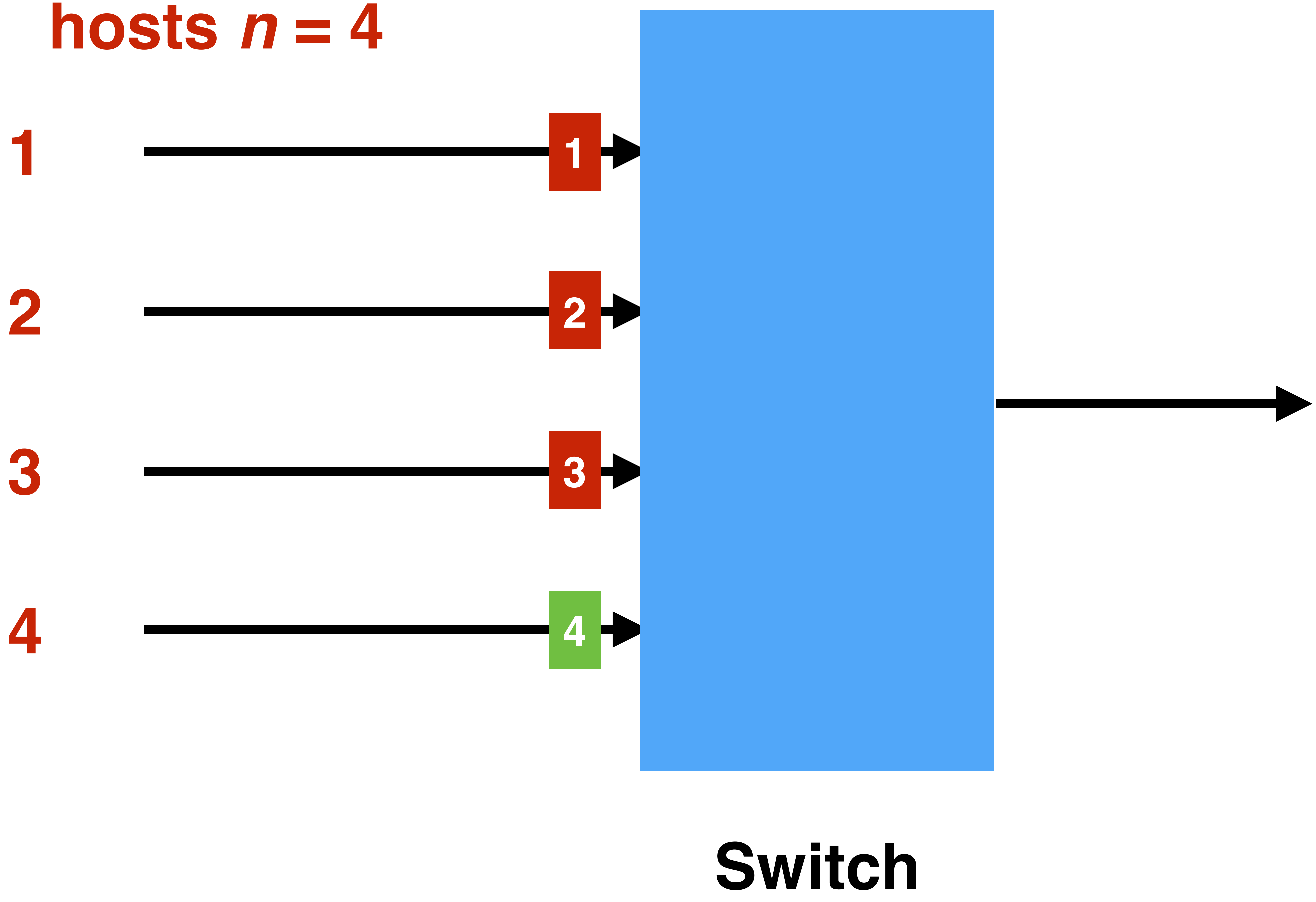


Rate-Limiting

If we can find a bound for servicing delay, we can rate-limit hosts so that they never experience queuing delay

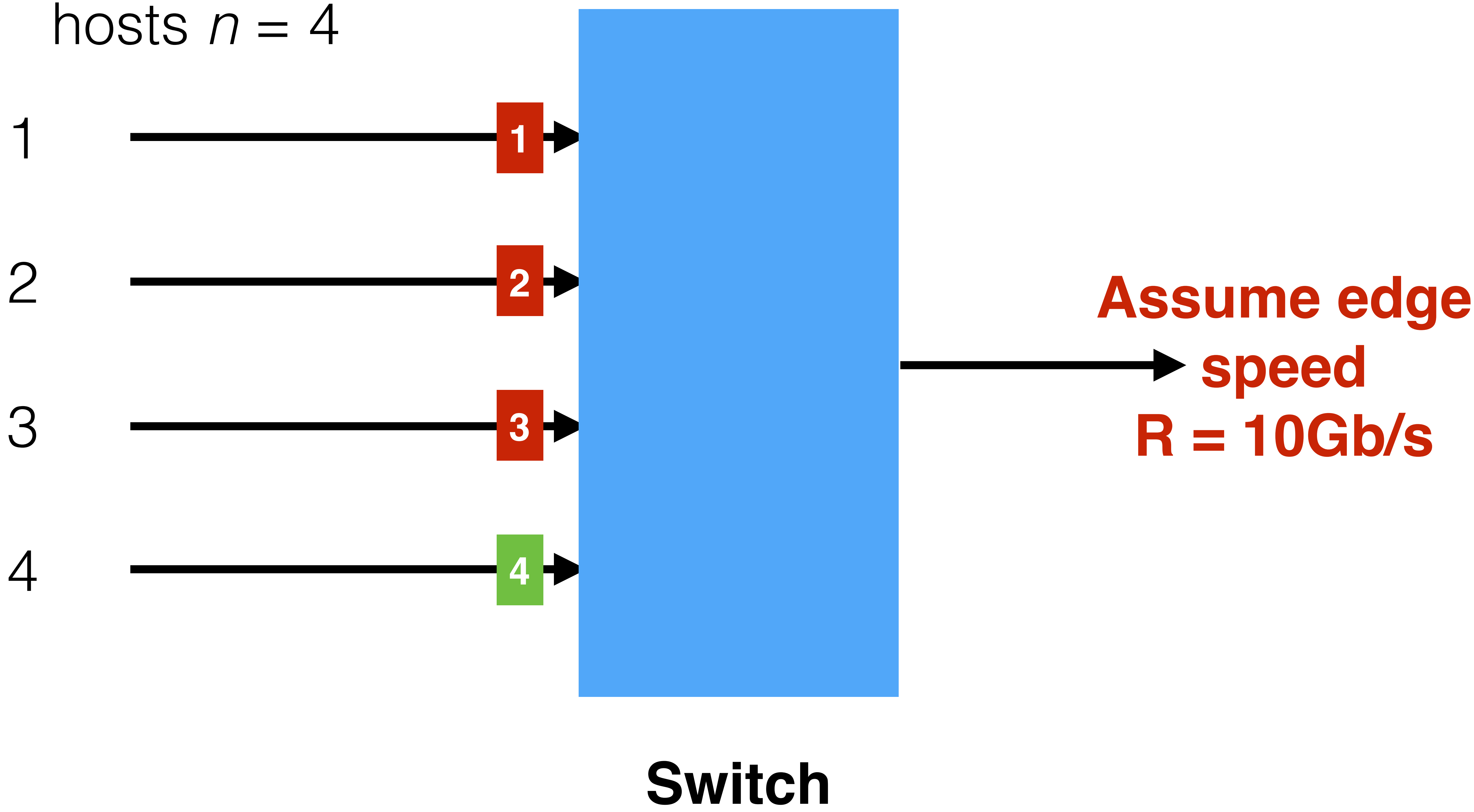
Calculating Service Delay

Assume sending hosts $n = 4$



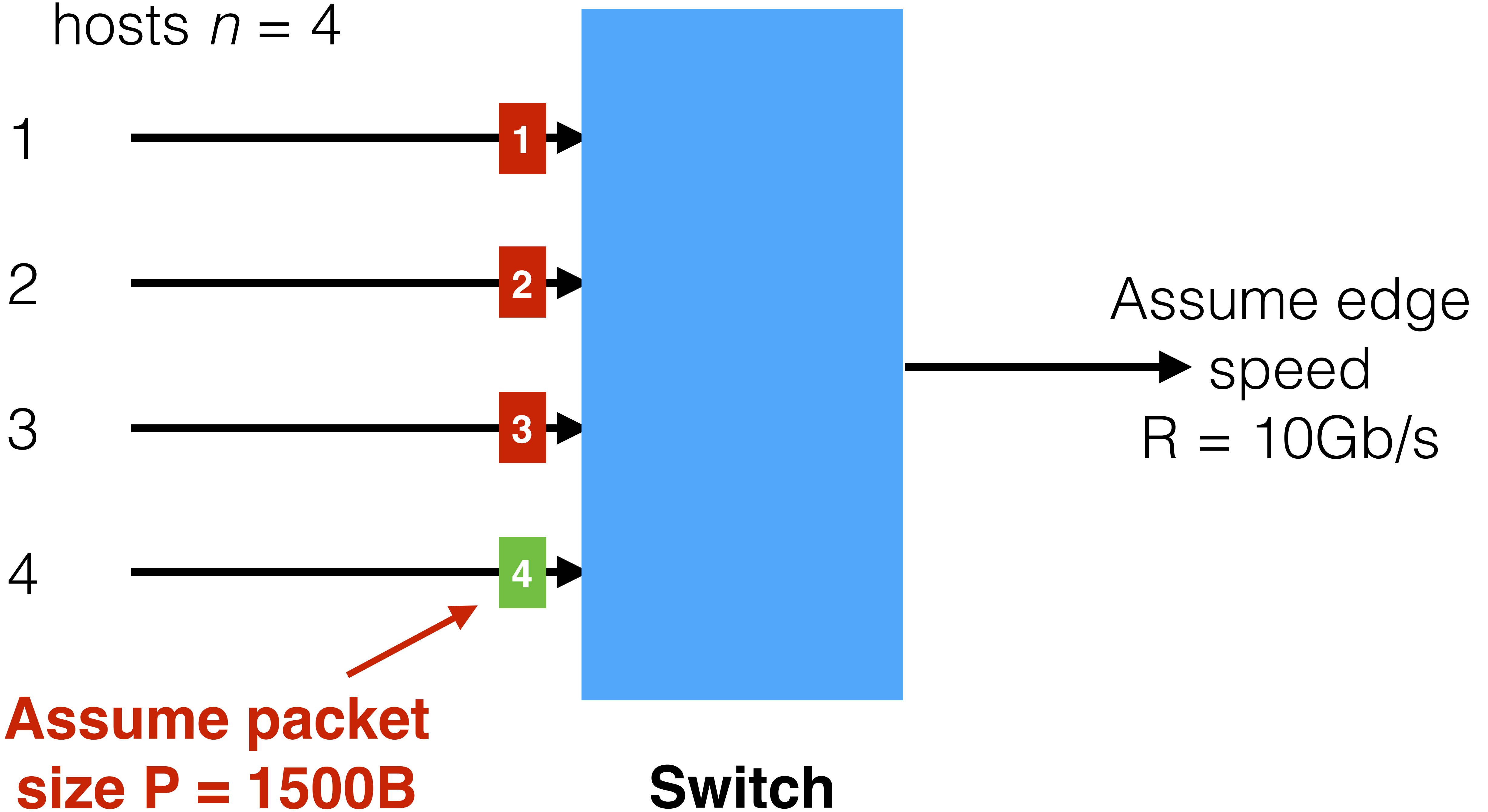
Calculating Service Delay

Assume sending hosts $n = 4$



Calculating Service Delay

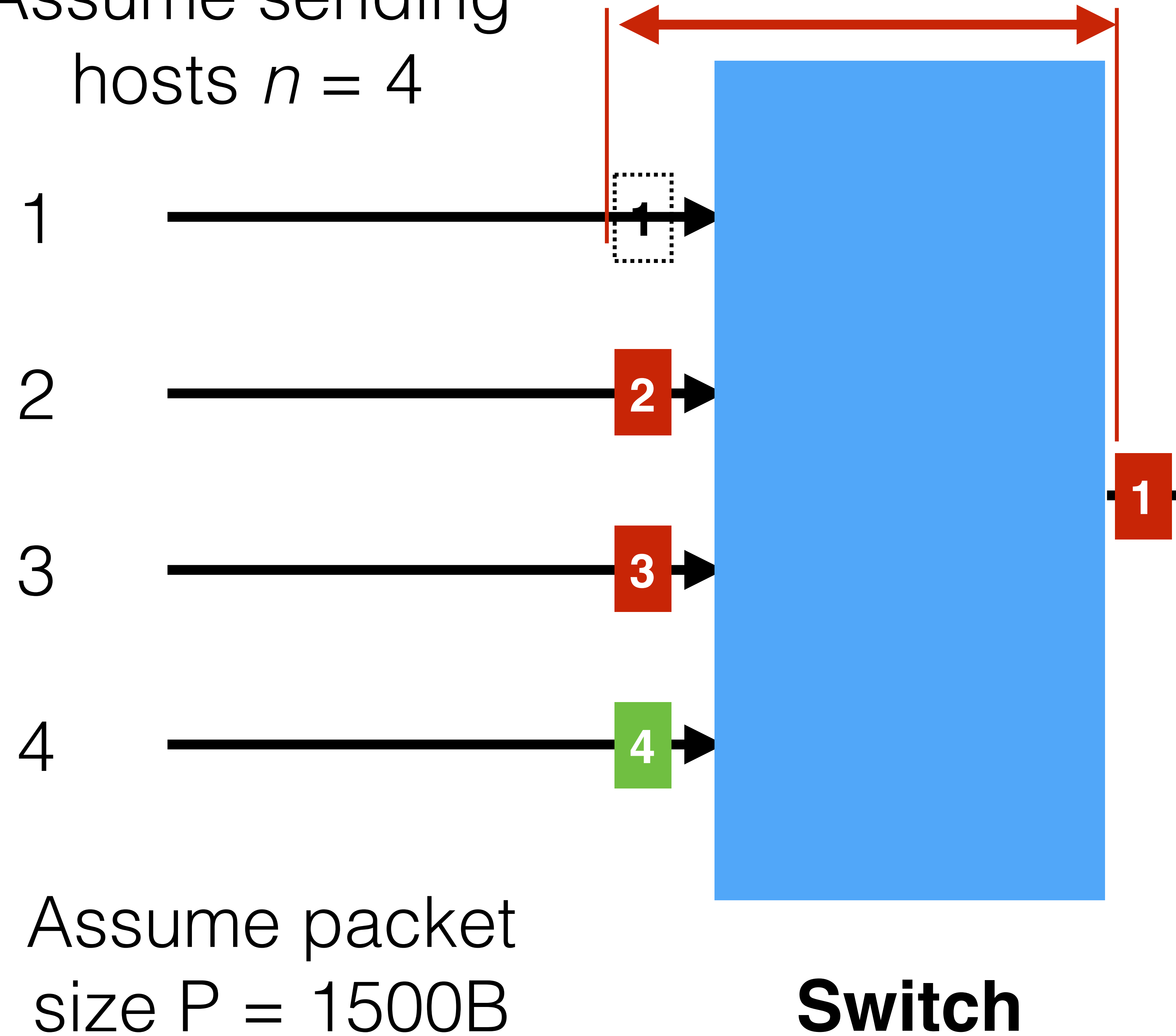
Assume sending hosts $n = 4$



Assume packet size P = 1500B

Calculating Service Delay

Assume sending hosts $n = 4$



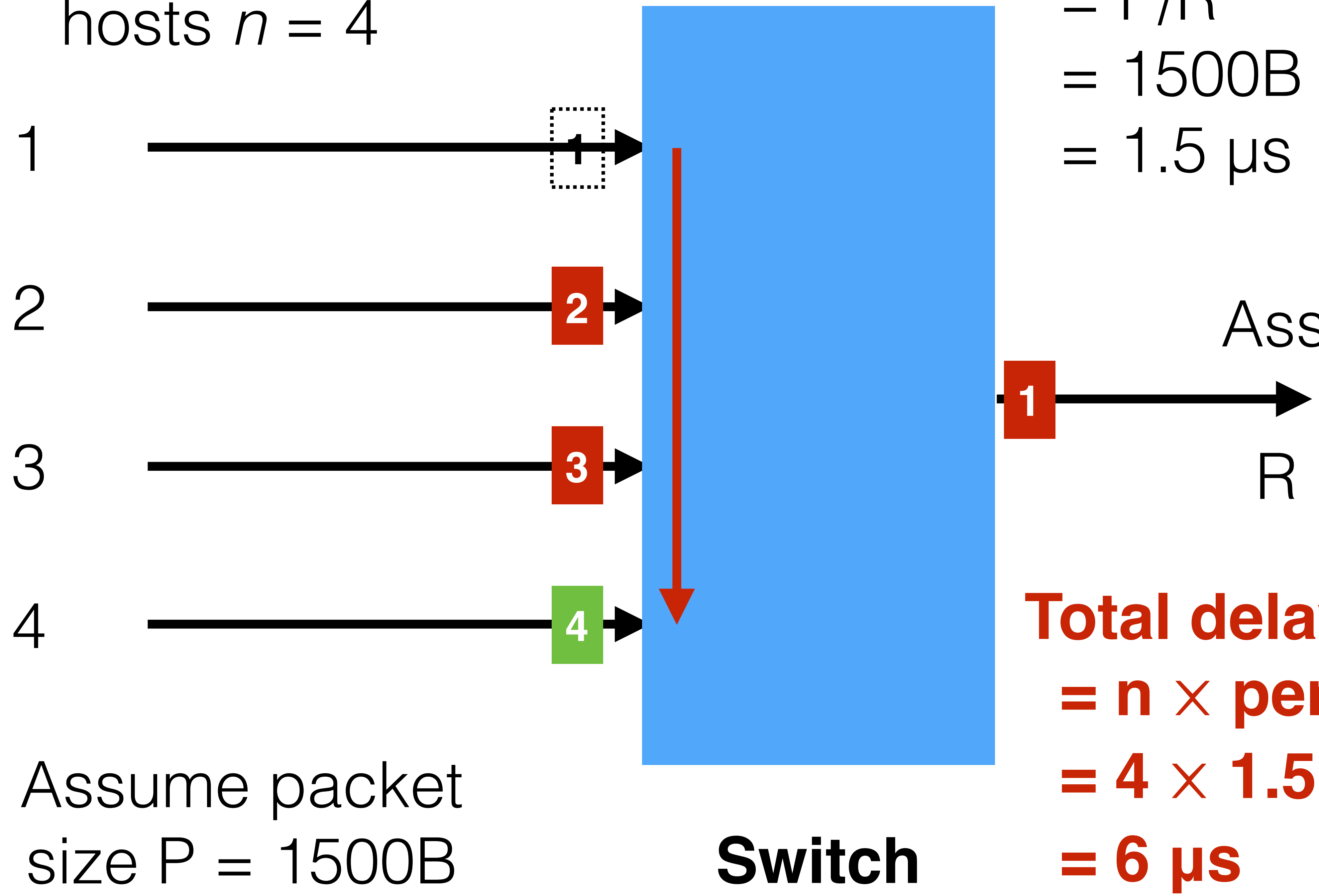
Delay per packet
= P/R
= 1500B / 10Gb/s
= 1.5 µs

Assume edge speed R = 10Gb/s

Assume packet size P = 1500B

Calculating Service Delay

Assume sending hosts $n = 4$



Delay per packet
= P/R
= 1500B / 10Gb/s
= 1.5 μs

Assume edge speed R = 10Gb/s

Total delay
= n × per packet
= 4 × 1.5 μs
= 6 μs

Assume packet size P = 1500B

Calculating Servicing Delay

$$\text{servicing delay} = n \times \frac{P}{R}$$

Calculating Servicing Delay

$$\text{servicing delay}^* = n \times \frac{P}{R}$$

Where

n - number of hosts

P - bytes sent

R - edge speed

*Assuming a fair scheduler

Calculating Servicing Delay

$$\text{network}^{**}\ \text{servicing delay}^{*} = n \times \frac{P}{R}$$

Where

n - number of hosts

P - bytes sent

R - edge speed

*Assuming a fair scheduler

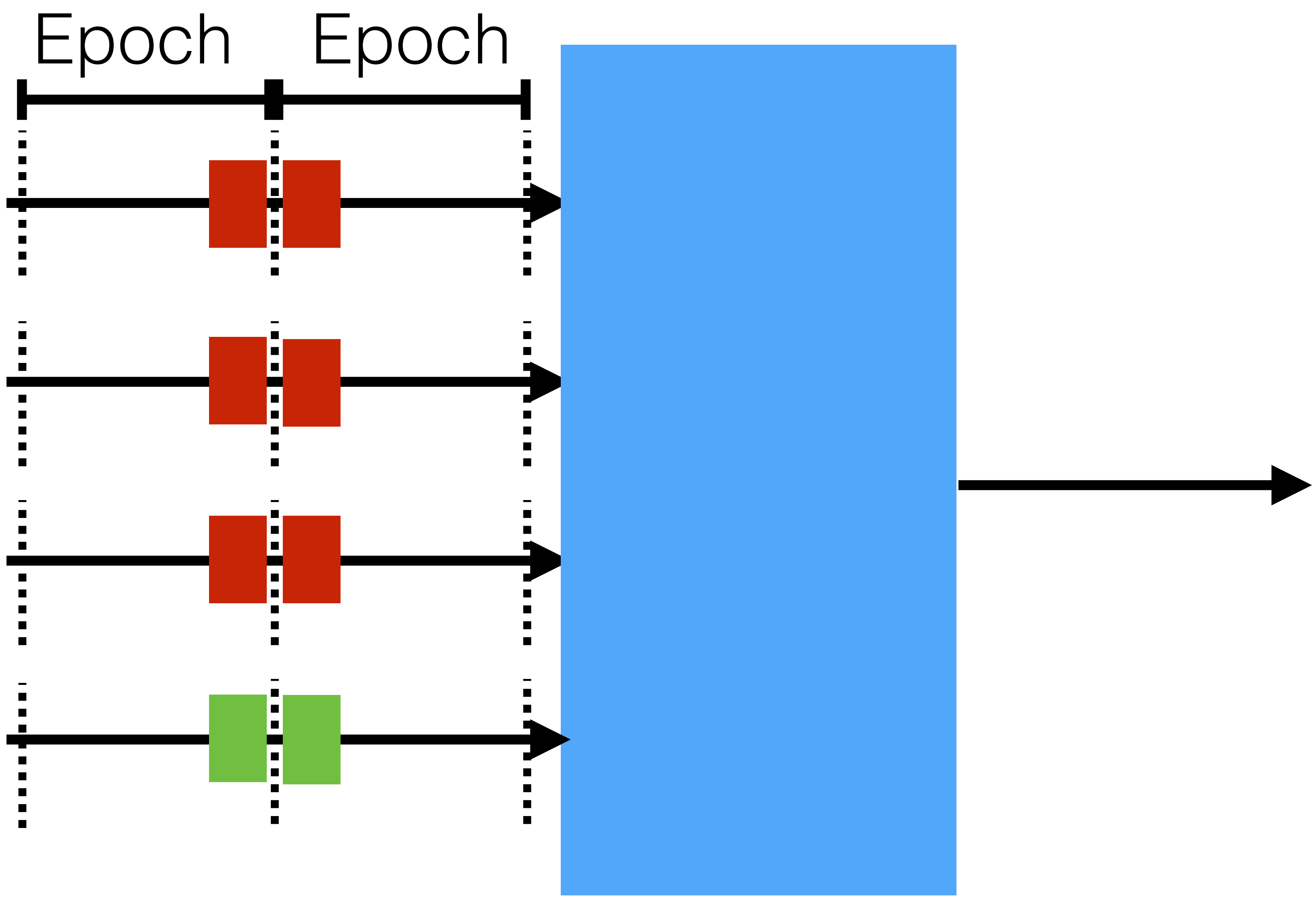
**Apply hose constraint model

Rate-Limiting

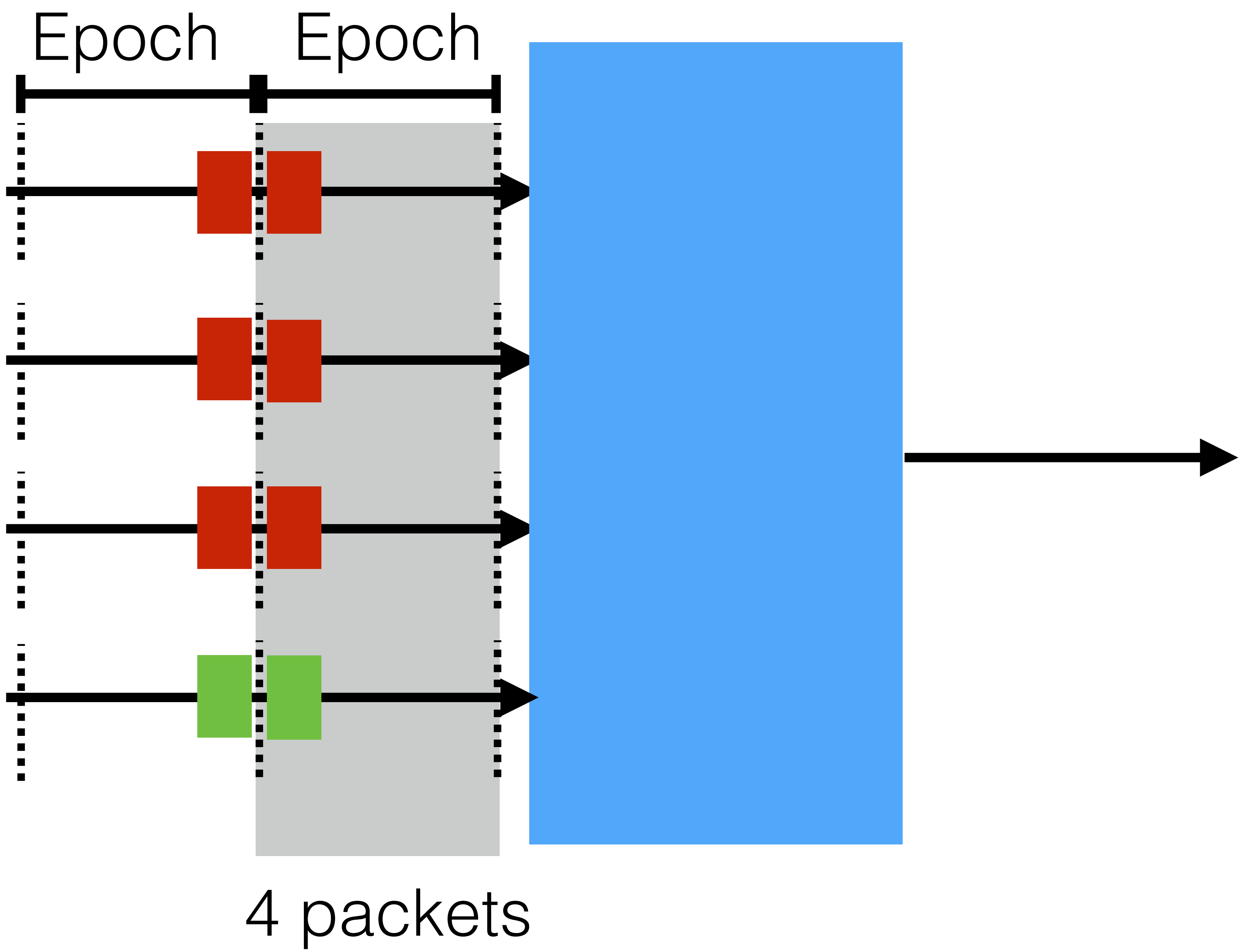
1. Network is idle
2. Hosts send ≤ *P* bytes
3. Wait (*n* x *P* / *R*) secs
4. Goto 1

} Network Epoch

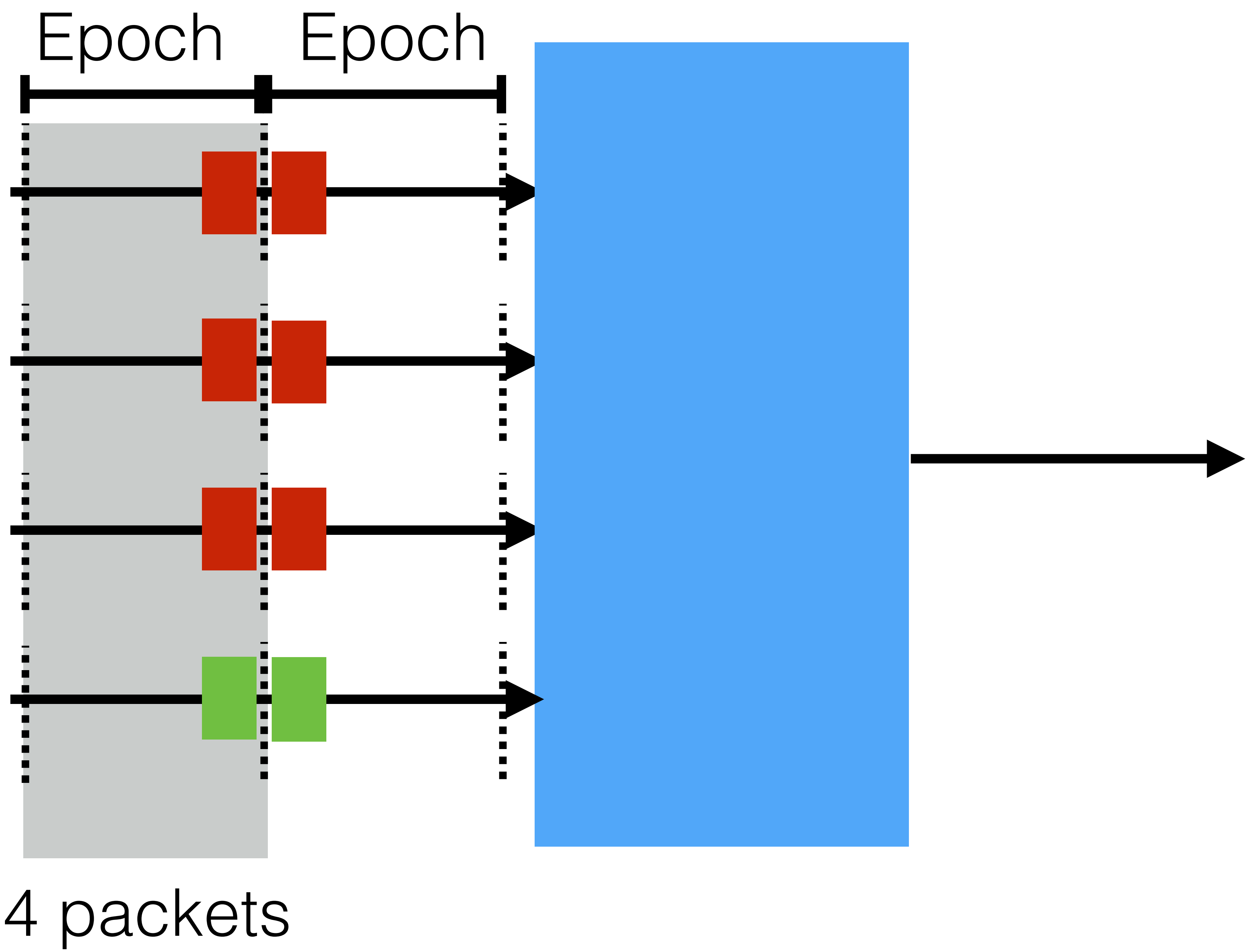
Eliminating Synchronization



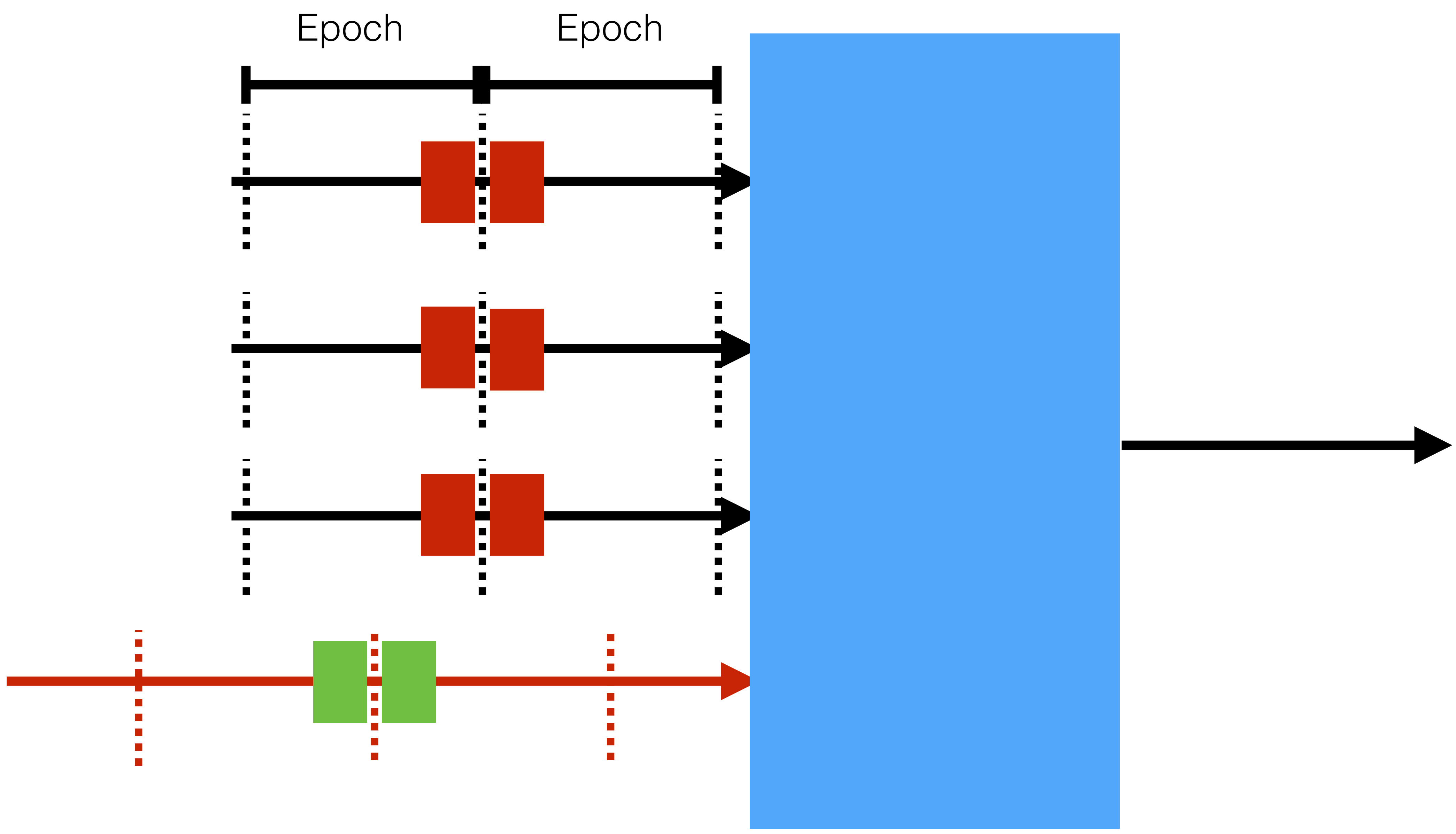
Eliminating Synchronization



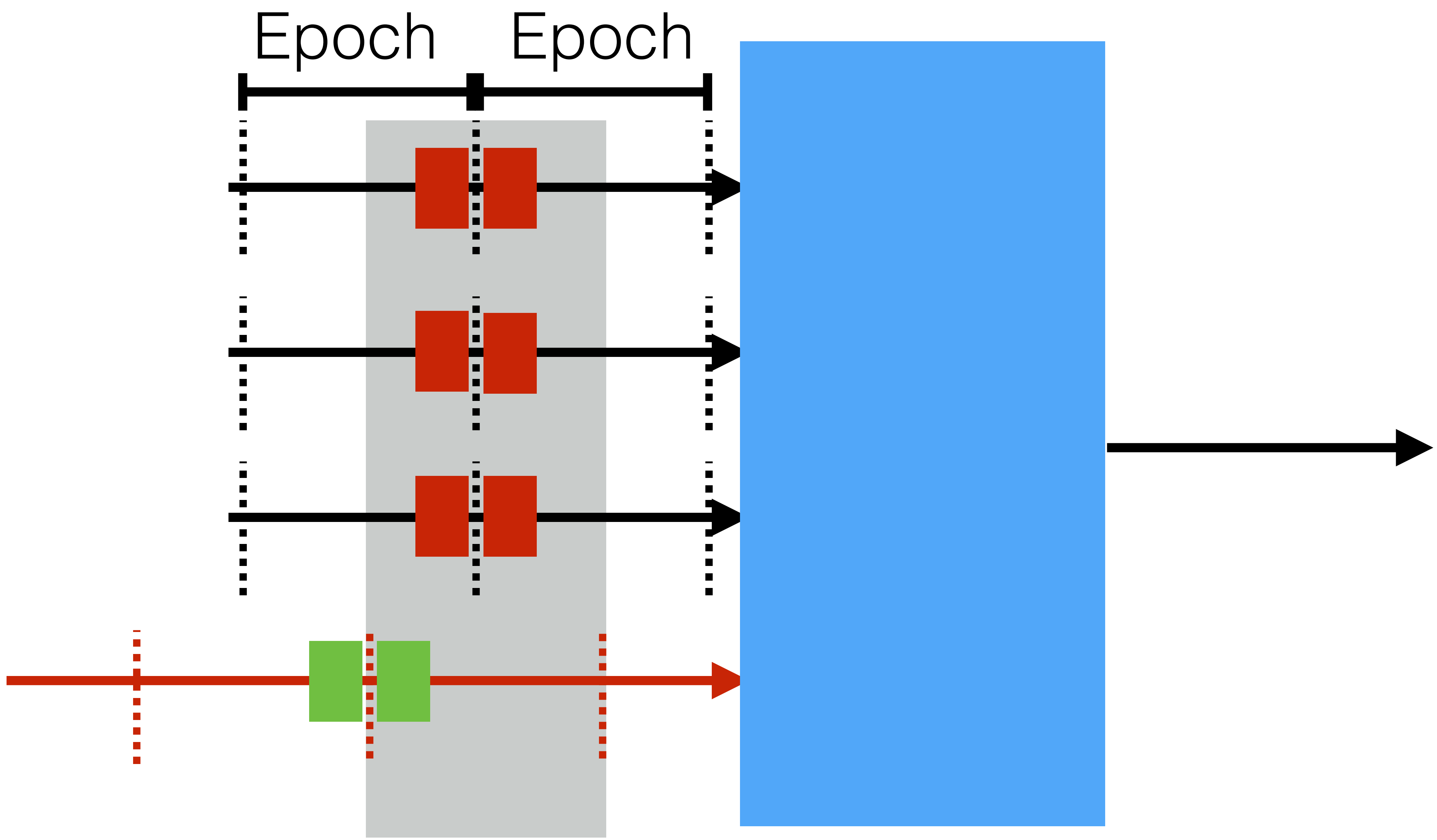
Eliminating Synchronization



Eliminating Synchronization



Eliminating Synchronization



≈ 8 packets per epoch

Eliminating Synchronization

$$\text{network epoch} = 2n \times \frac{P}{R}$$

Where

- **n - number of hosts**
- **P - bytes sent**
- **R - edge speed**
- **2 - mesochronous compensation**

The dark side of network epoch

$$\text{throughput} = \frac{R}{2n}$$

Where

n is the number of hosts

R is the edge speed

The dark side of network epoch

$$\text{throughput} = \frac{10\text{Gb/s}}{2 \times 1000} = \mathbf{5Mb/s}$$

Where

n = 1000 hosts

R = 10 Gb/s

The dark side of network epoch

$$\text{throughput}^{*} = \frac{10\text{Gb/s}}{2 \times 1000} = \mathbf{5Mb/s}$$

Where

n = 1000 hosts

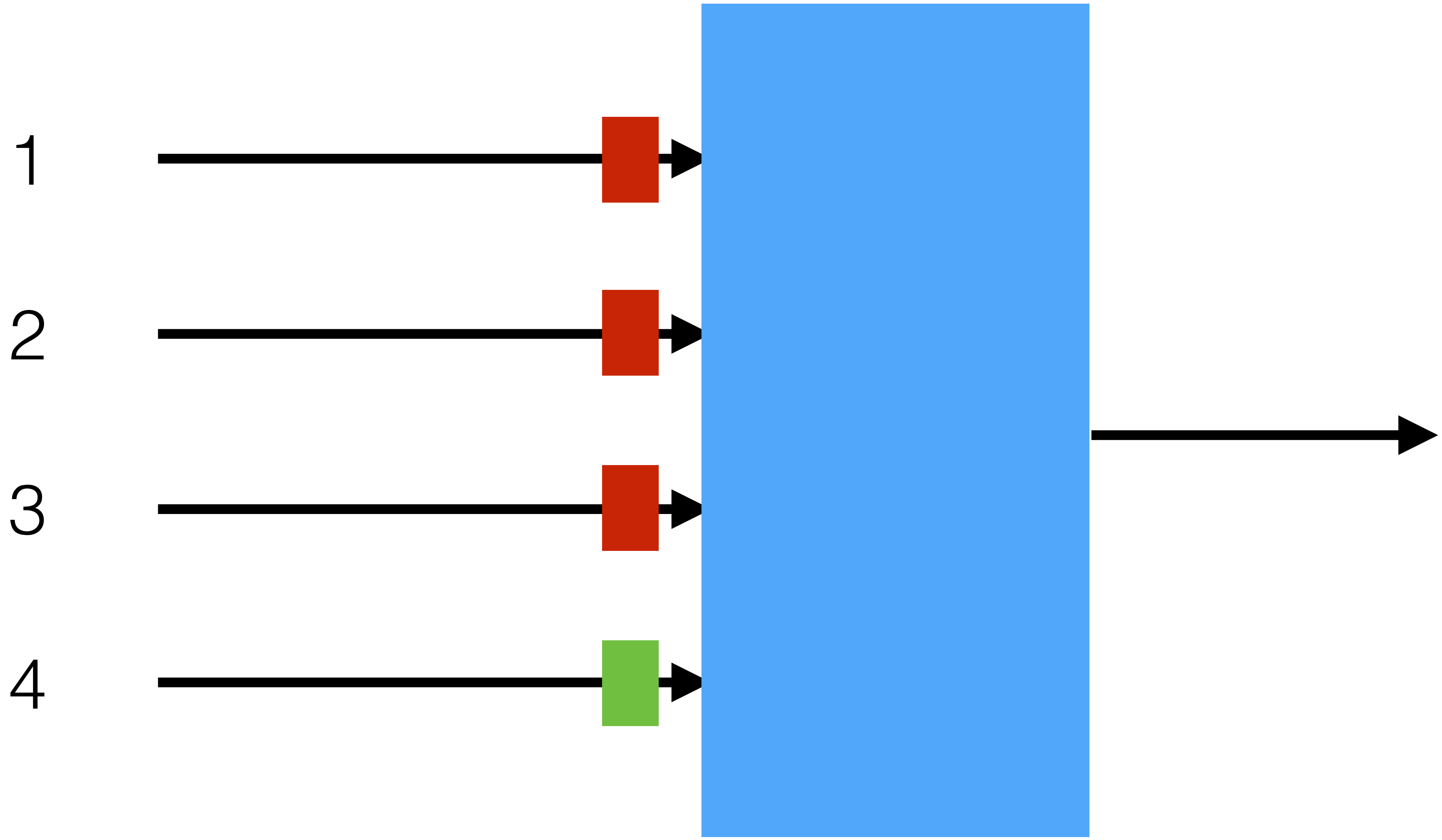
R = 10 Gb/s

*at guaranteed latency!

solution: assume there is no problem?

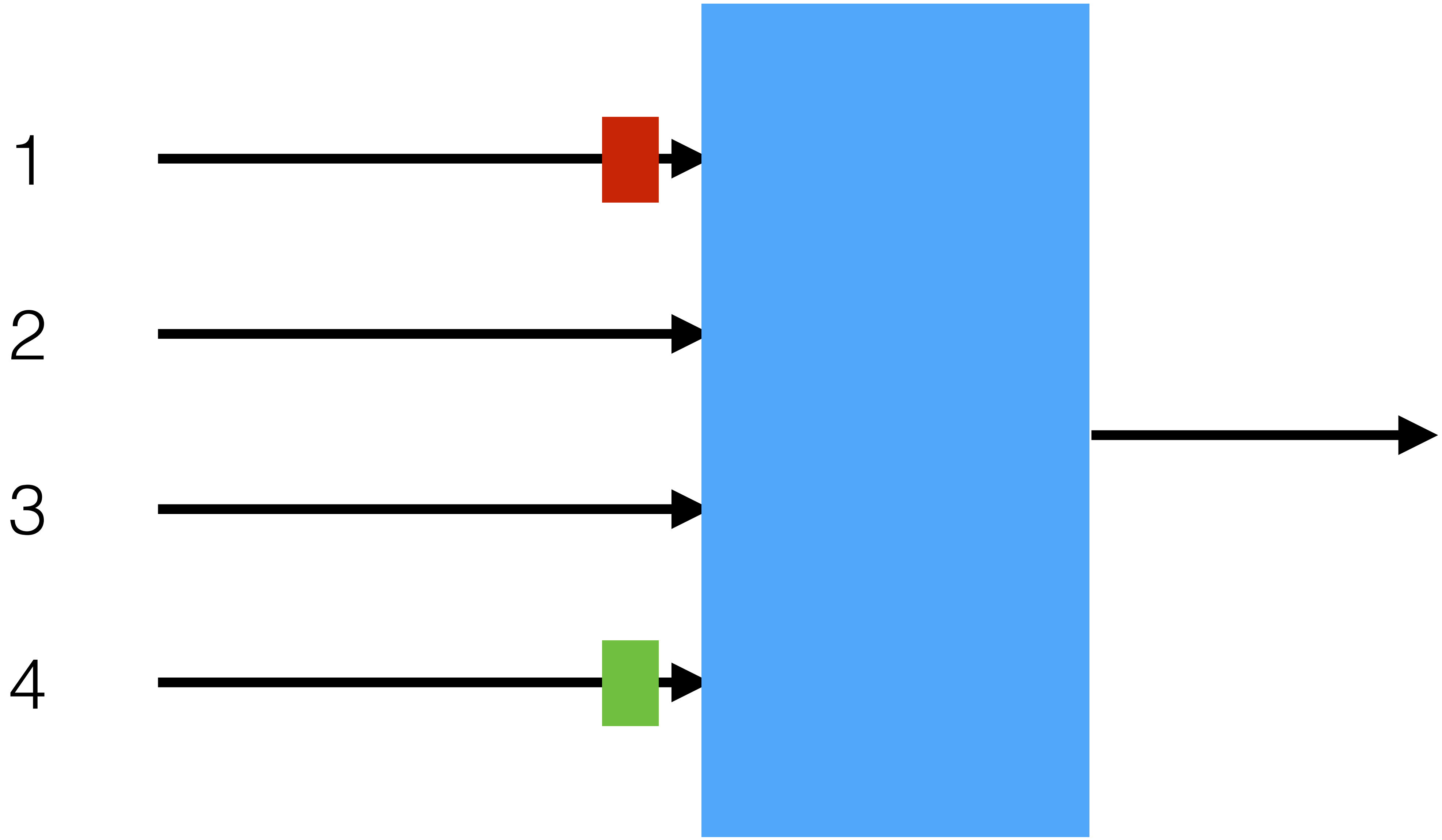
Changing the assumptions

Pessimistic assumption of 4:1



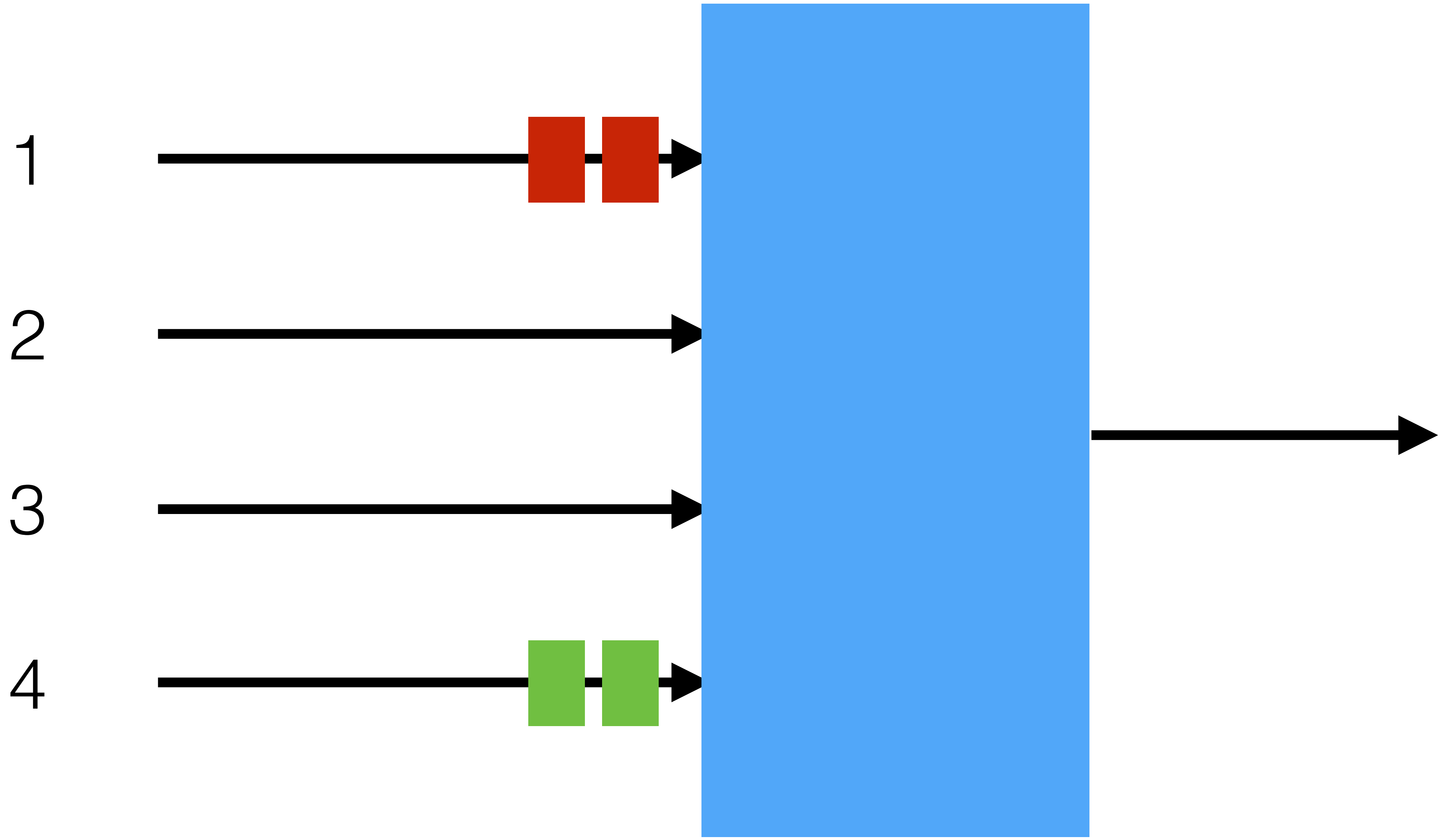
Changing the assumptions

What if we assume 2:1?



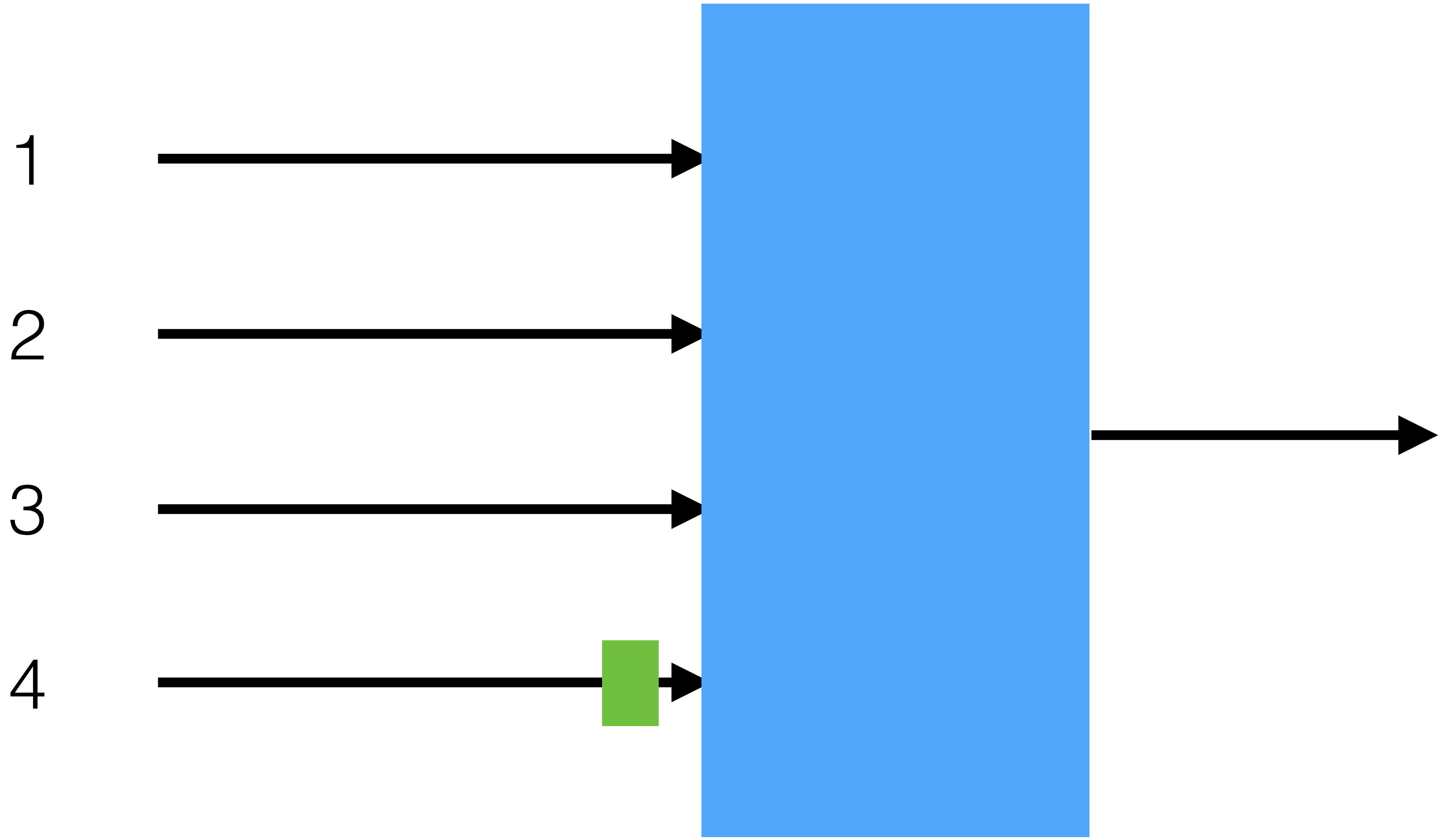
Changing the assumptions

What if we assume 2:1? Hosts can send 2x the rate!



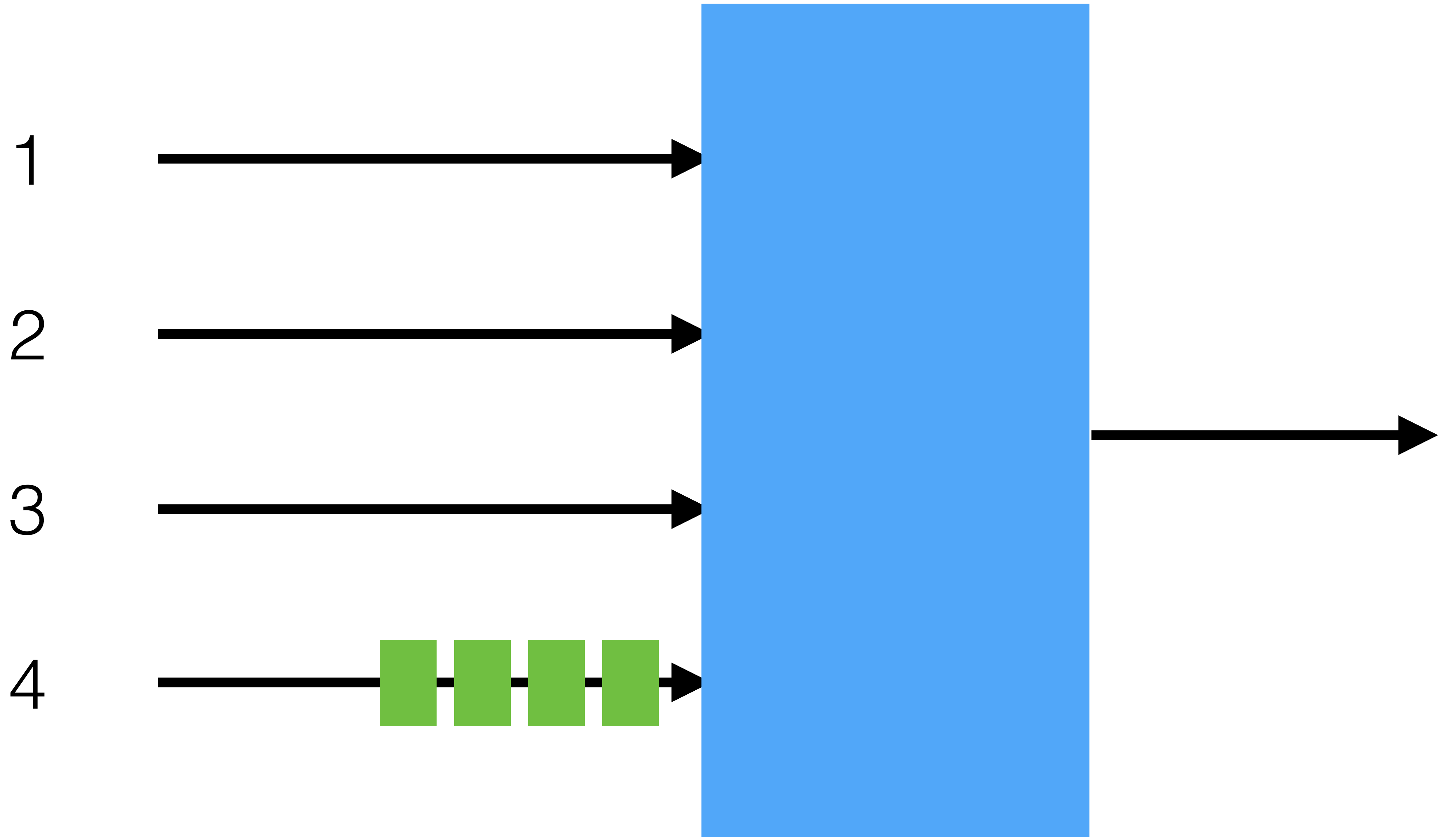
Changing the assumptions

What if we assume 1:1?



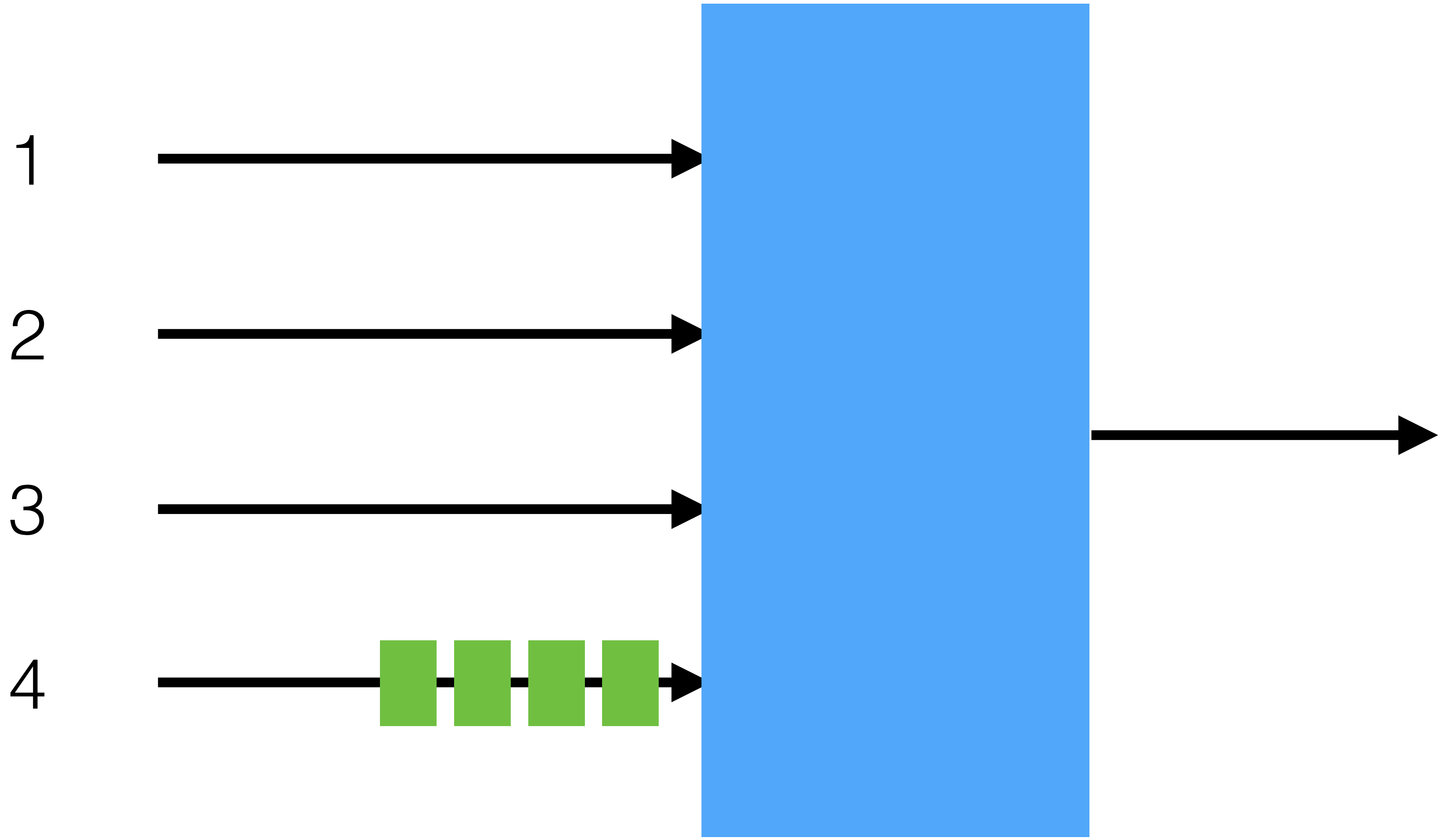
Changing the assumptions

What if we assume 1:1? Hosts can send 4x the rate!



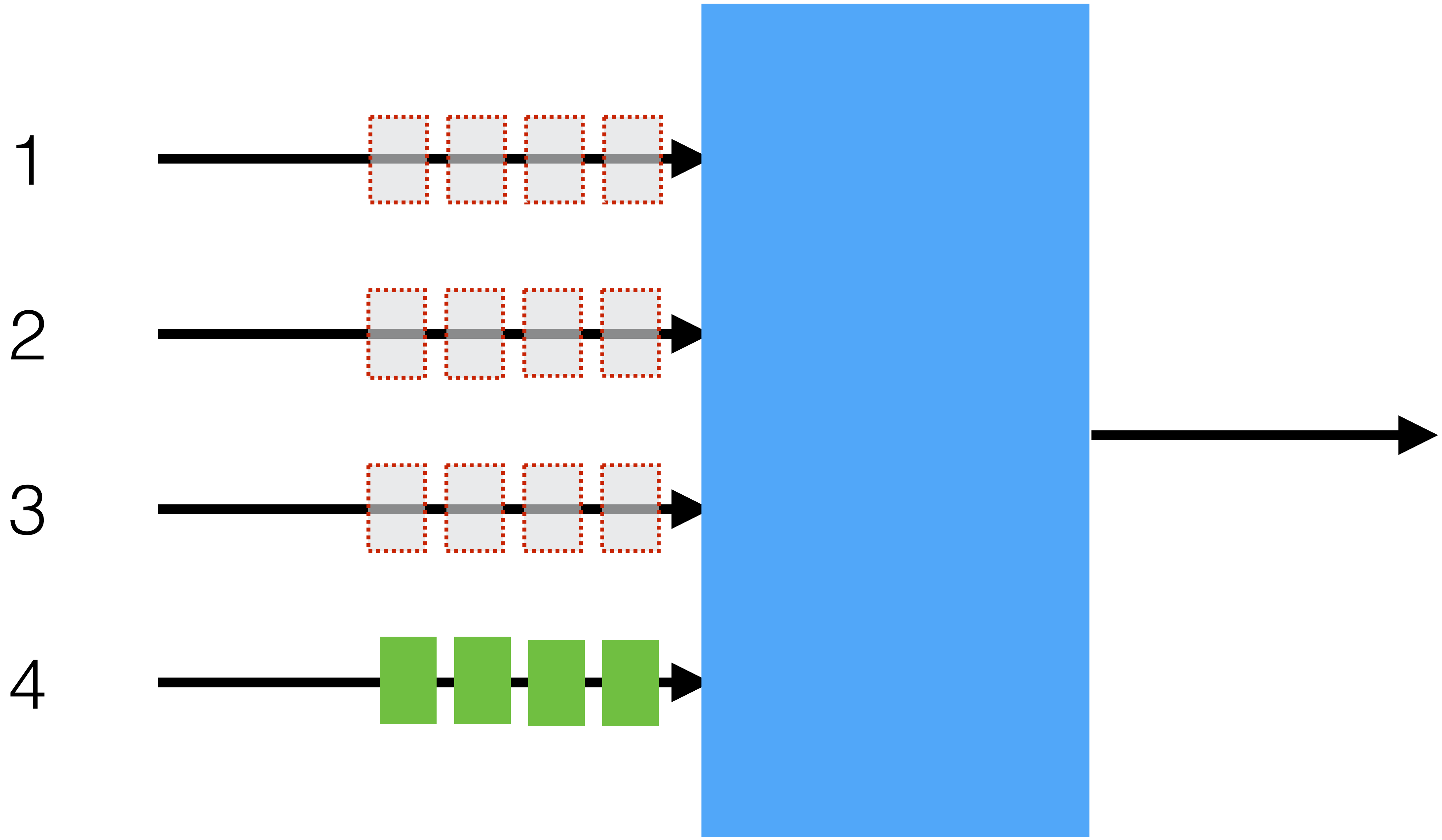
Changing the assumptions

What if assumption is wrong?

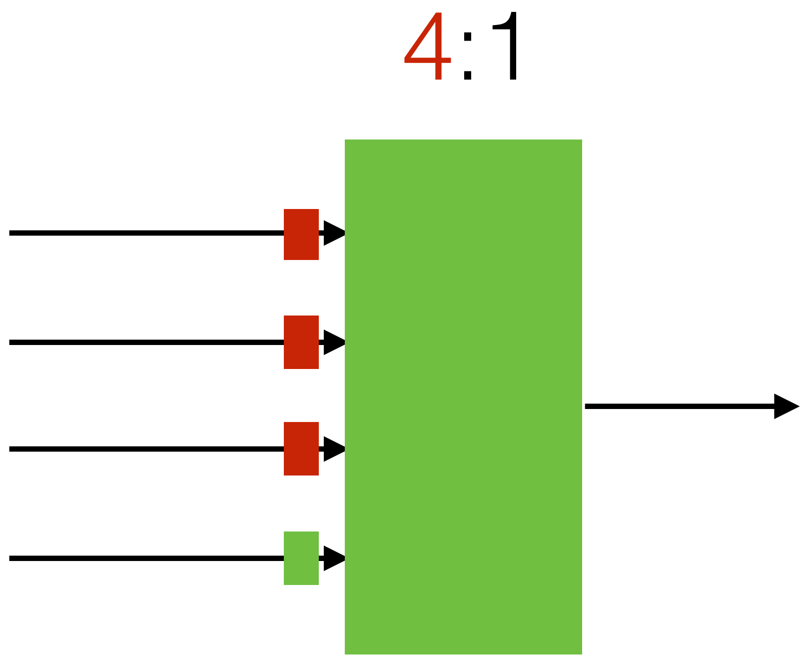


Changing the assumptions

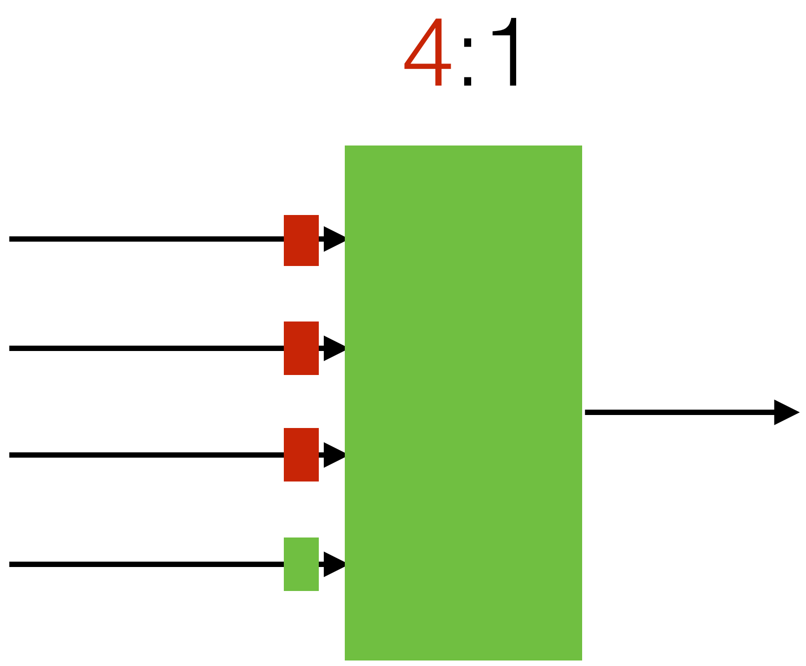
What if assumption is wrong? Queuing will happen!



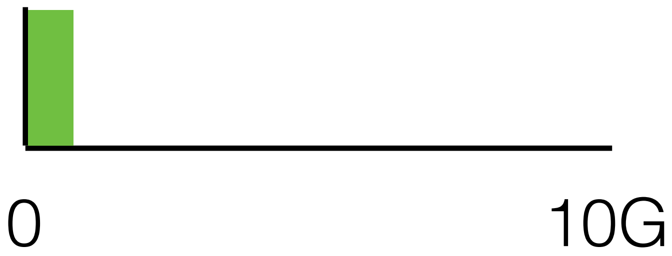
Which assumption?



Which assumption?

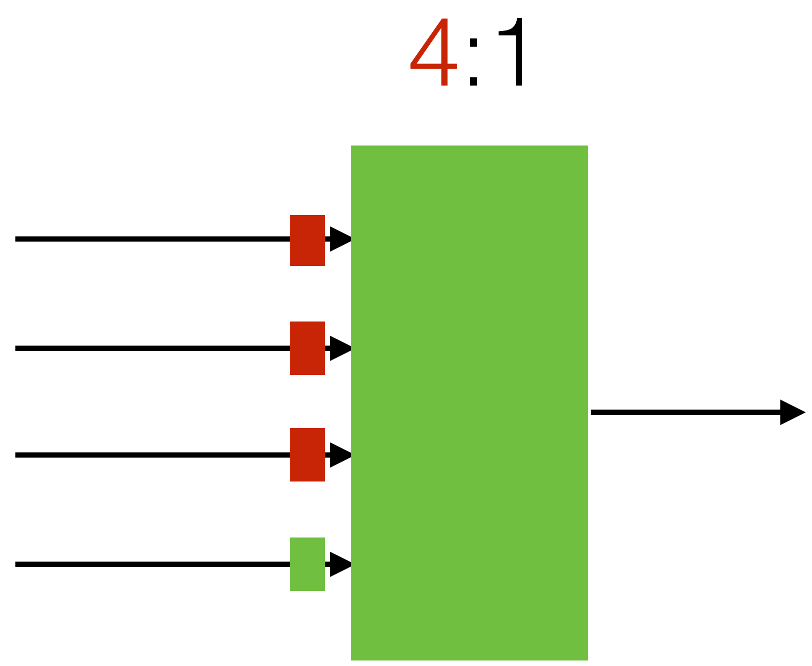


Rate limit

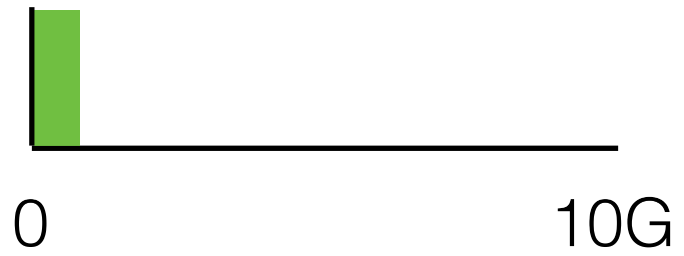


low throughput

Which assumption?

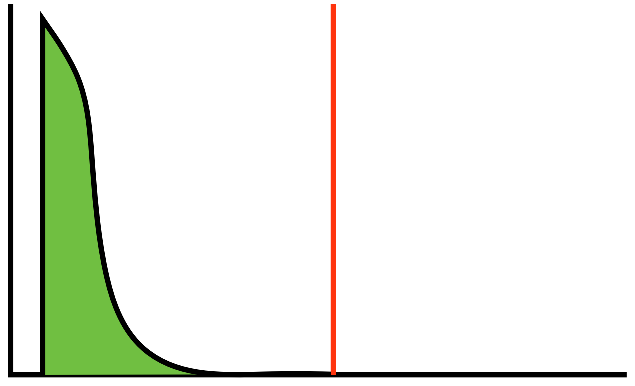


Rate limit



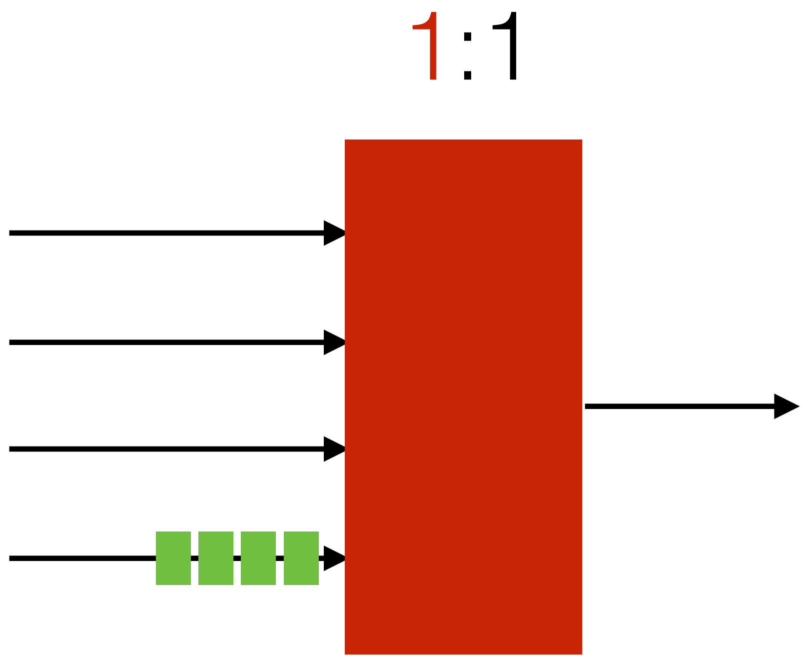
low throughput

Latency Distribution

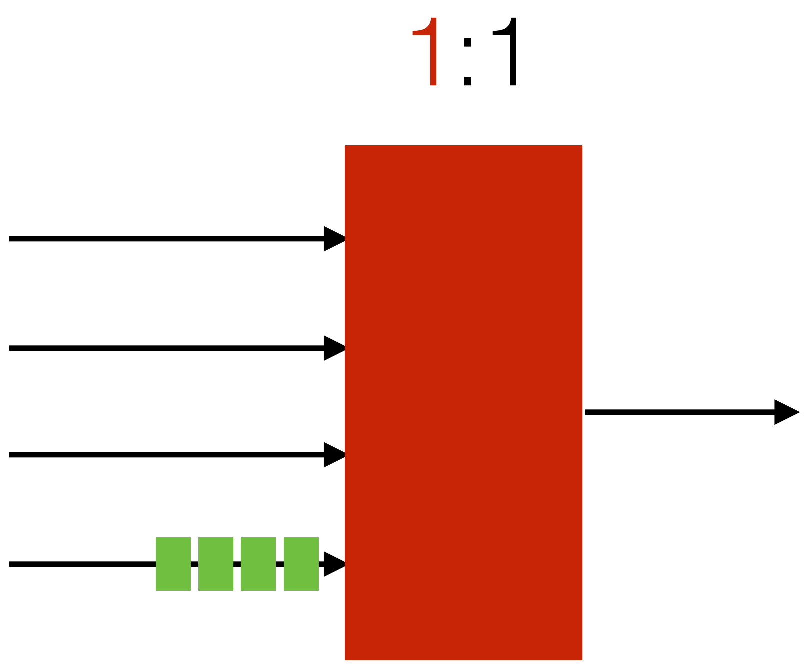


guaranteed latency

Which assumption?



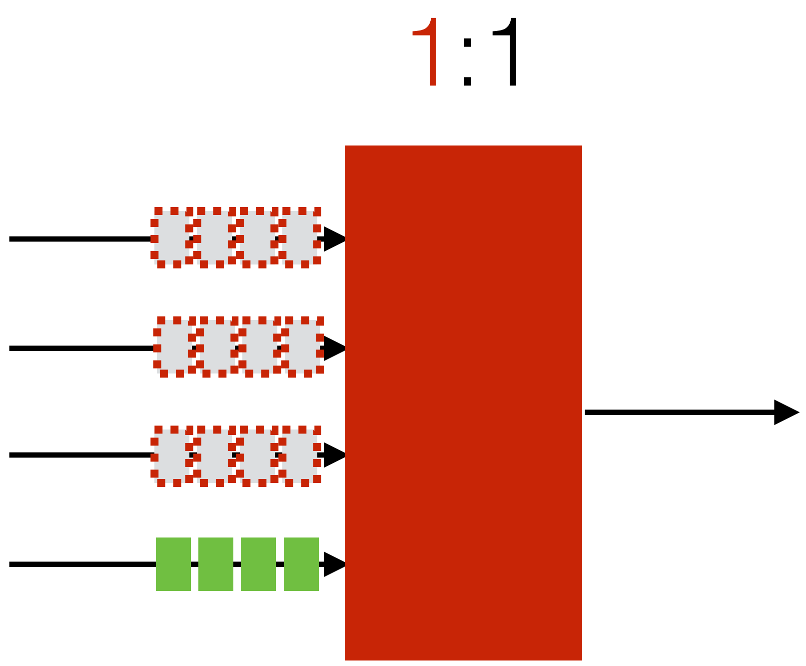
Which assumption?



Rate limit

line rate throughput

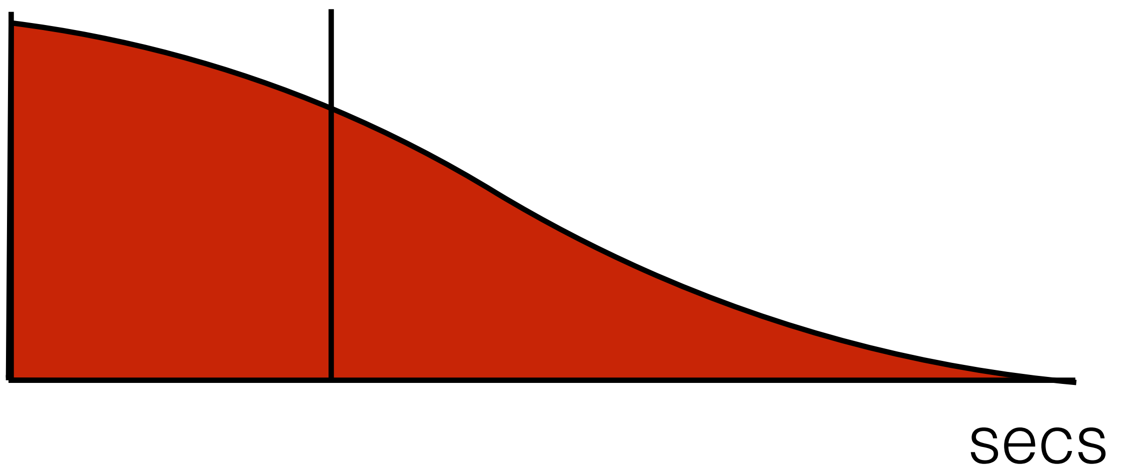
Which assumption?



Rate limit

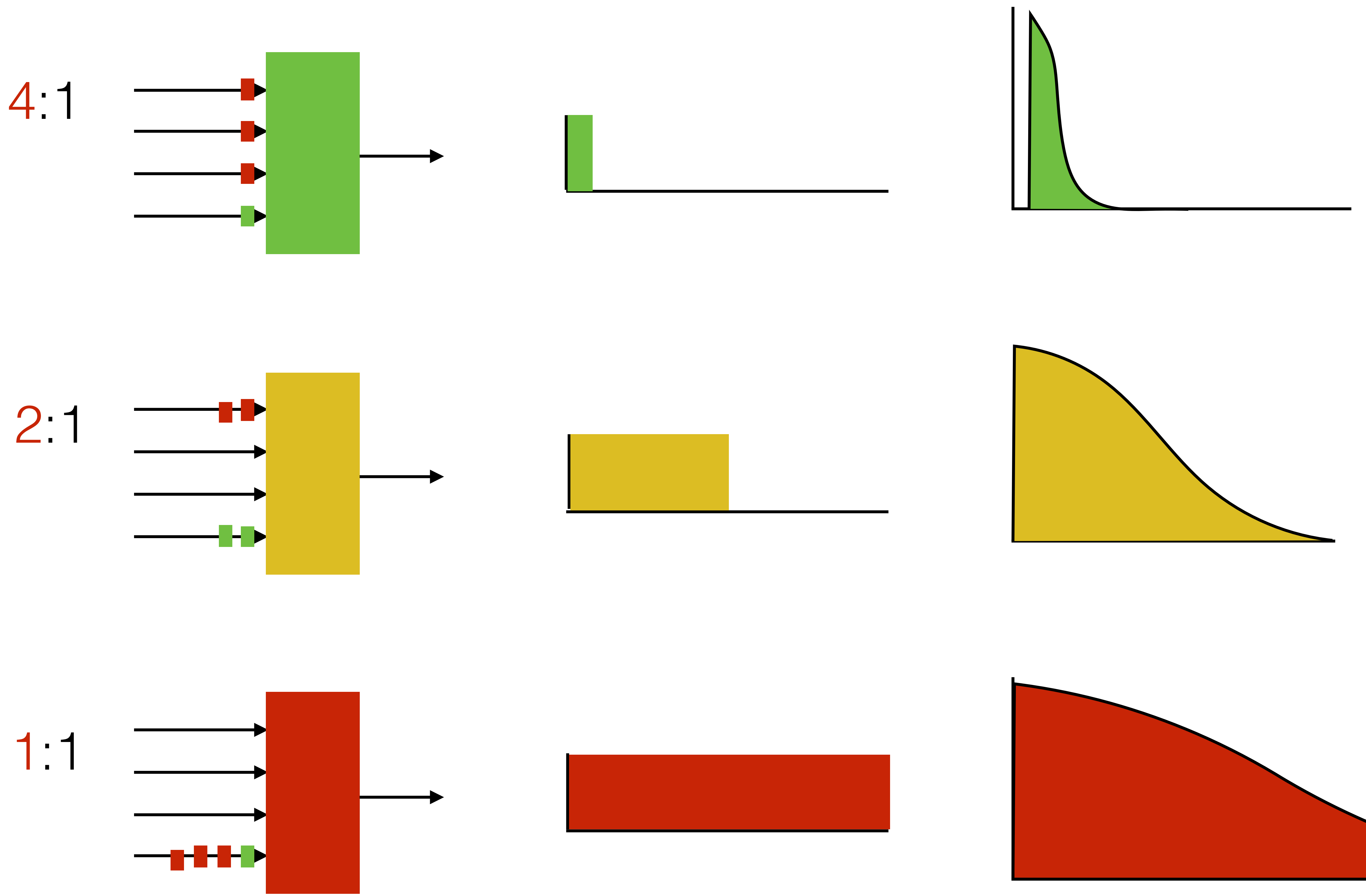
line rate throughput

Latency Distribution

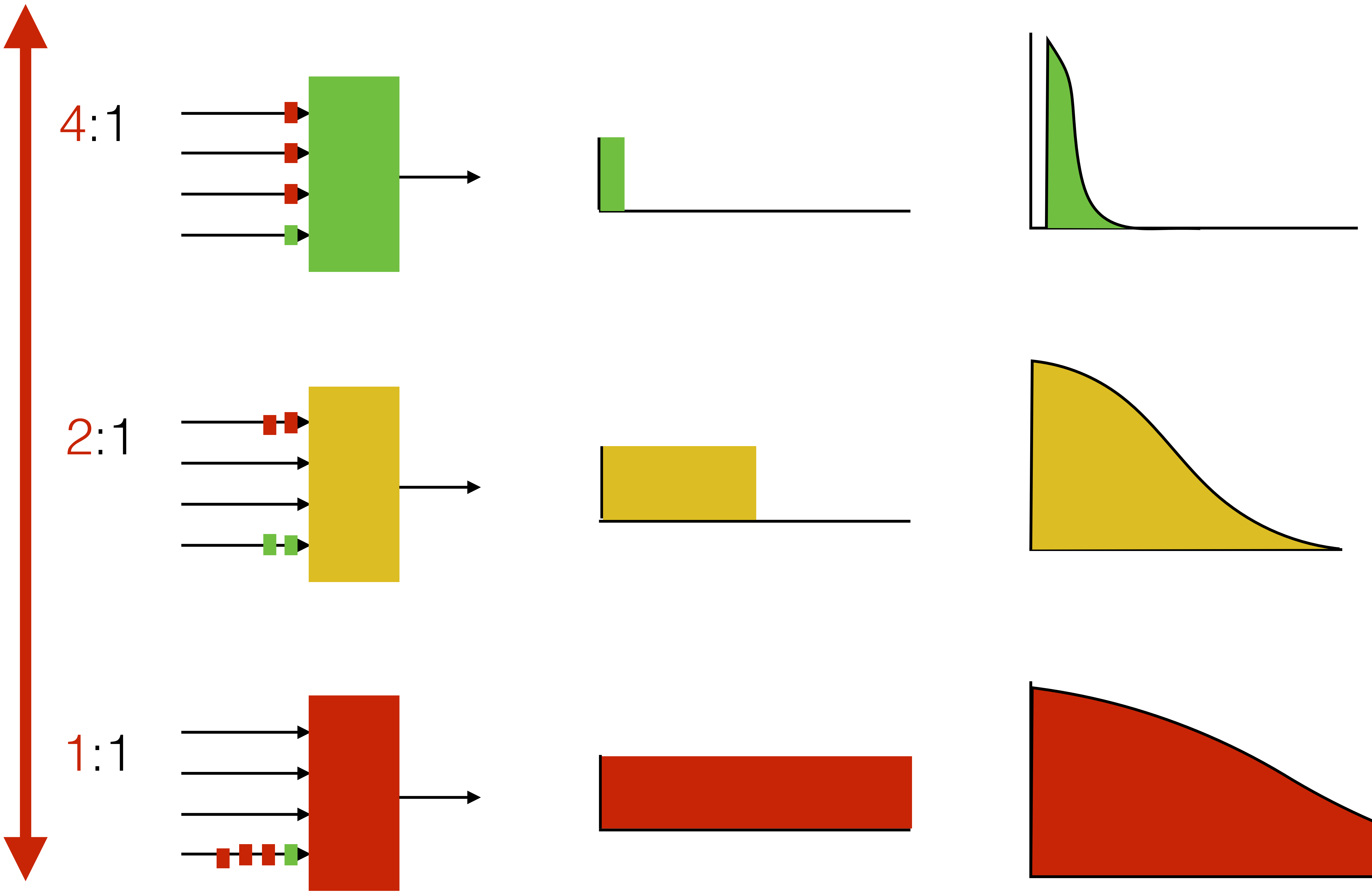


no latency guarantee

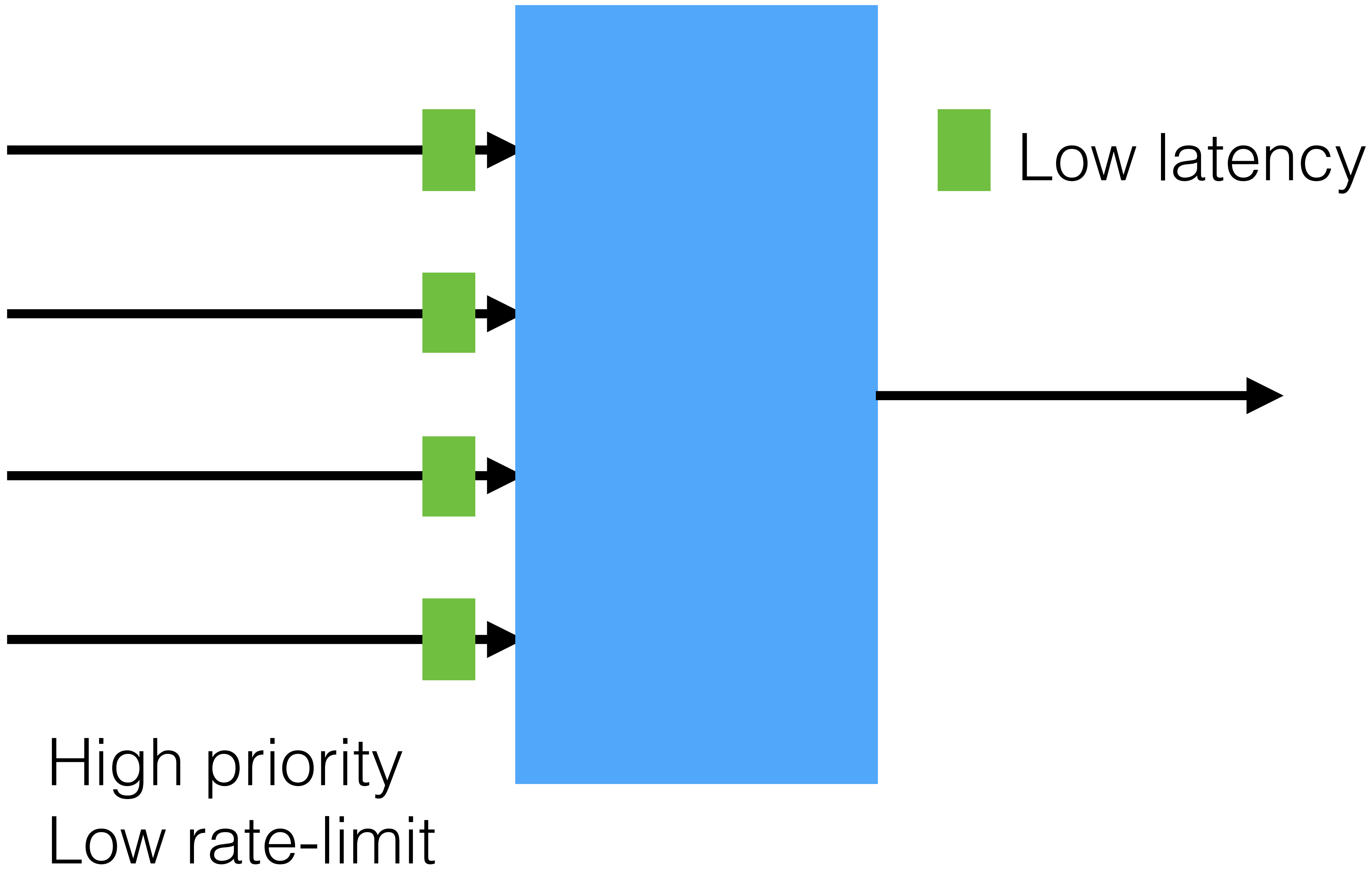
Which assumption?



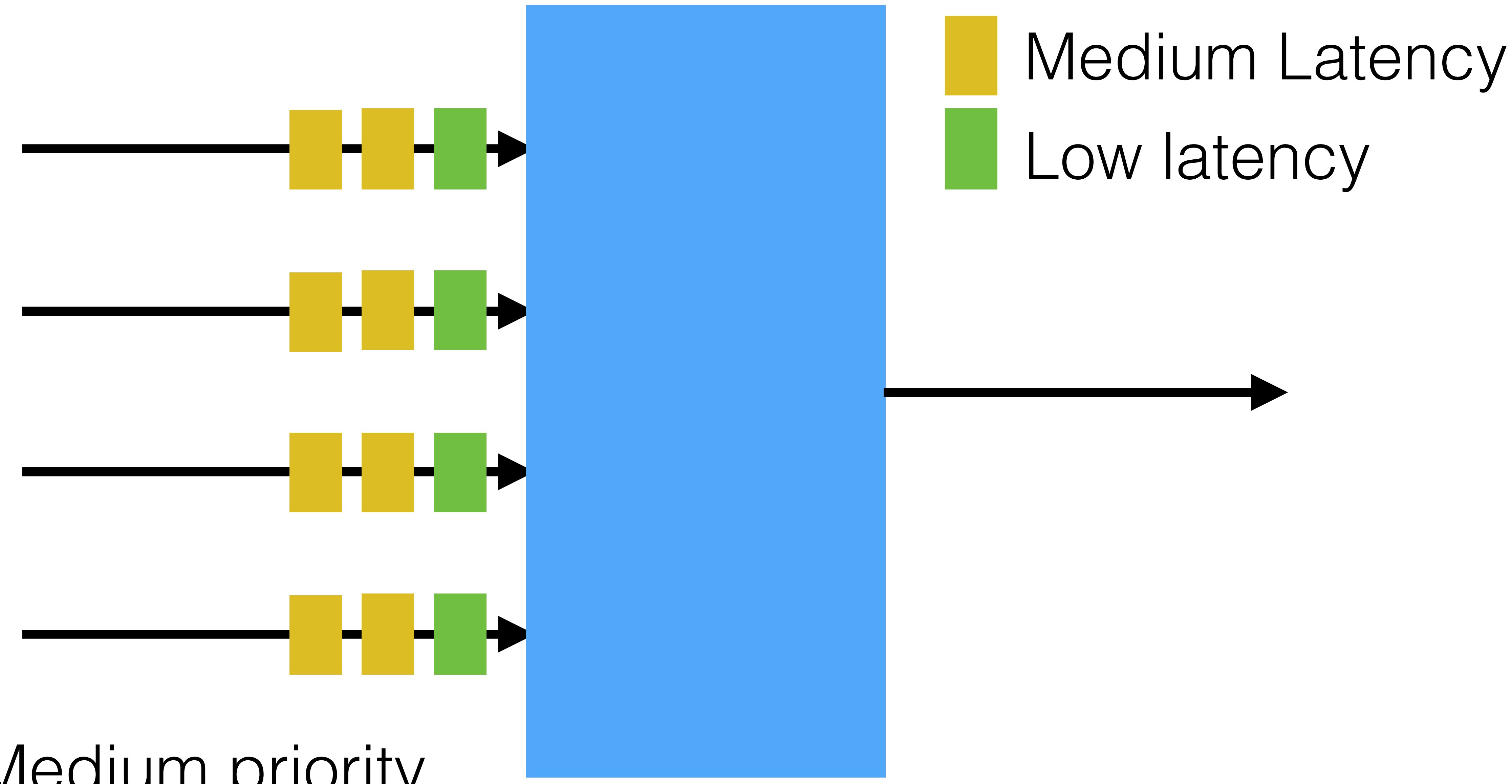
Which assumption?



QJump with priorities

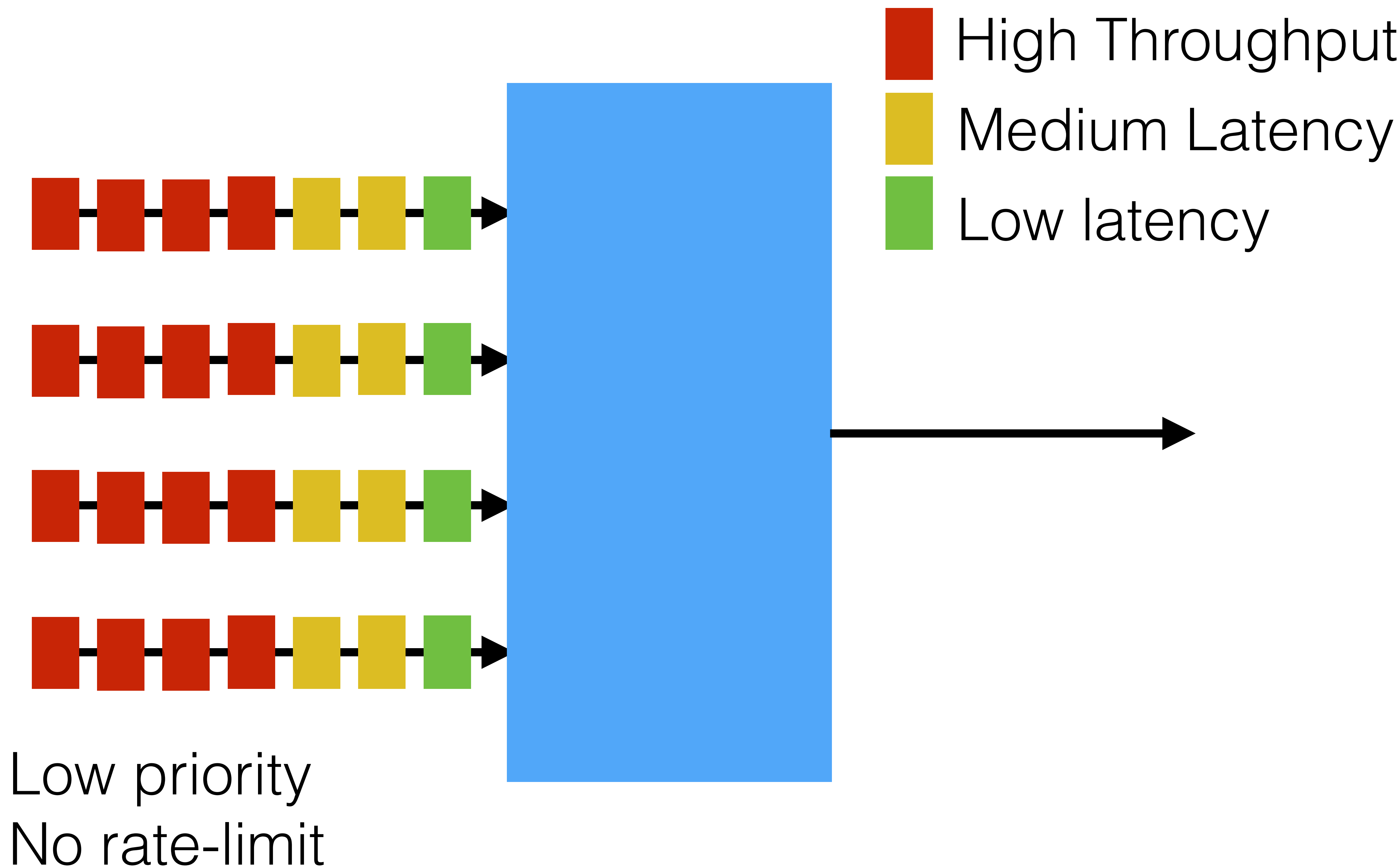


QJump with priorities

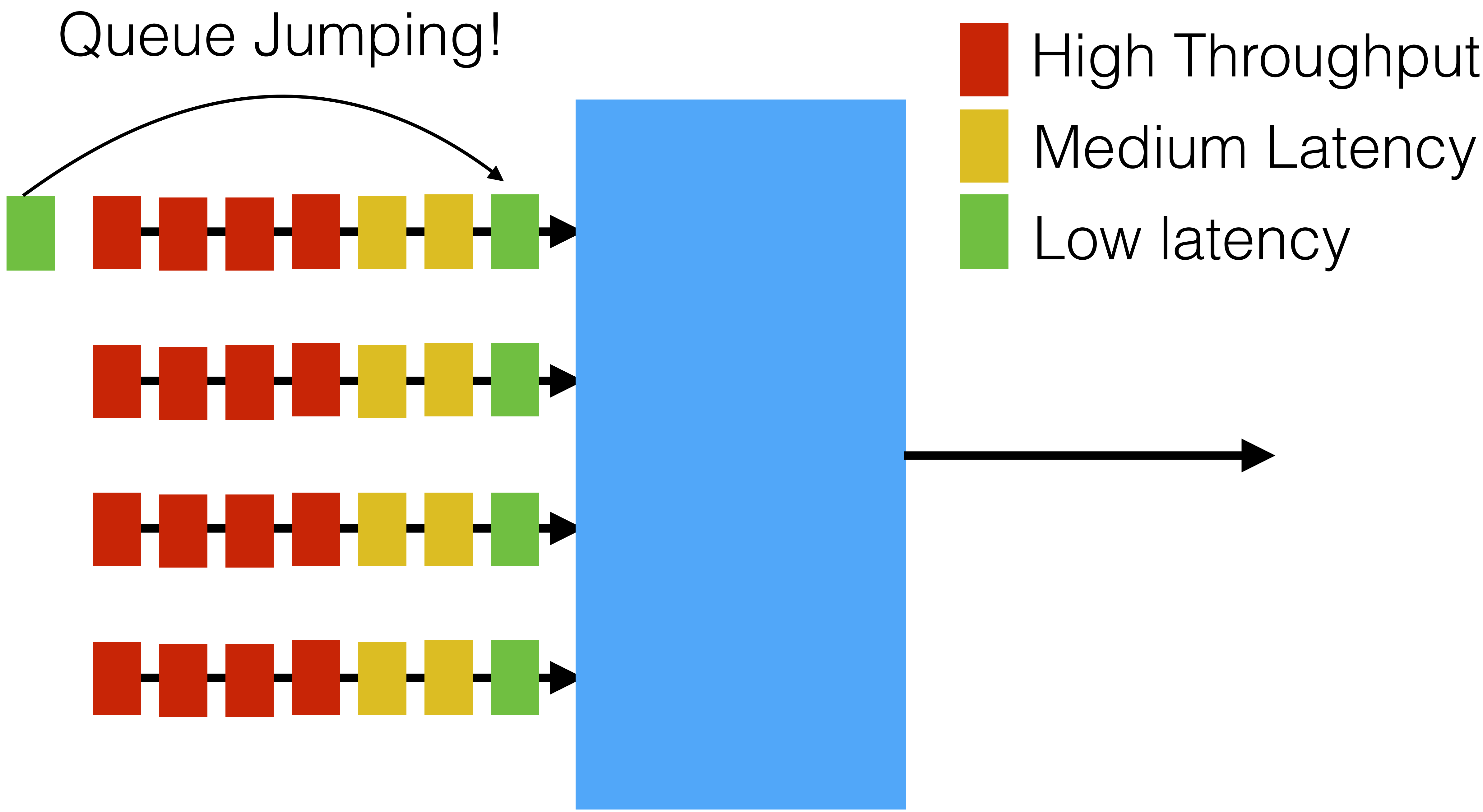


Medium priority
Medium rate-limit

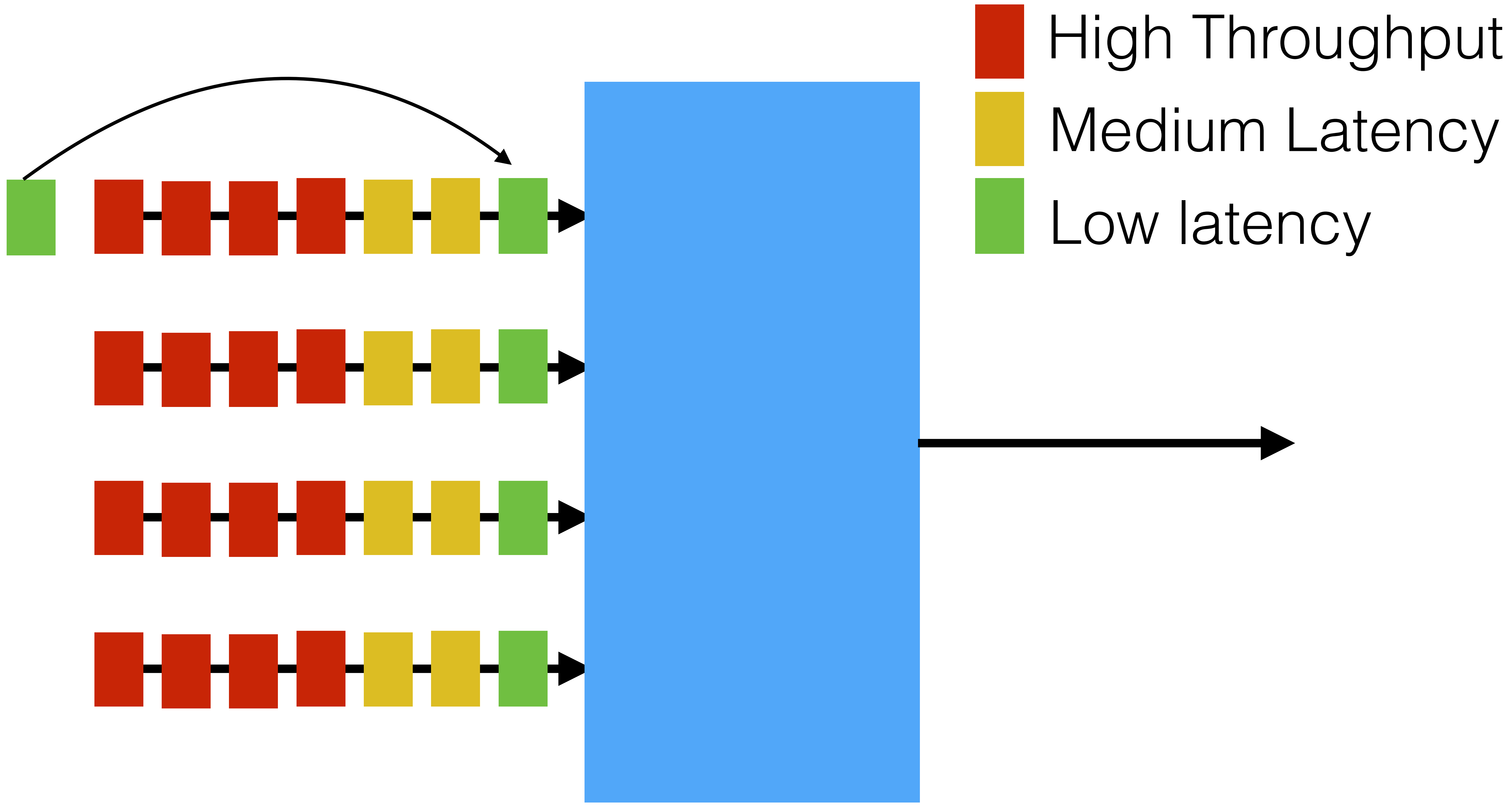
QJump with priorities



QJump with priorities



QJump with priorities



Queues don't matter when you can Jump them!

Prioritization

Use hardware priorities to run different QJump levels together, but isolated* from each other.

* from layers below

Implementation


```c
long epoch_cycles = to_cycles(network_epoch);
long timeout = start_time;
long bucket[NUM_QJUMP_LEVELS];

int qJumpRateLimiter(struct sk_buff* buffer) {
  long cycles_now = asm("rdtsc"); /* read cycle ctr */
  int level = buffer->priority;
  if (cycles_now > timeout) { /* new token alloc? */
    timeout += epoch_cycles;
    bucket[level] = tokens[level];
  }
  if (buffer->len > bucket[level]) {
    return DROP; /* tokens for epoch exhausted */
  }
  bucket[level] -= buffer->len;
  sendToHWQueue(buffer, level);
  return SENT;
}
```

Implementation

Linux TC

```
long bucket[NUM_QJUMP_LEVELS];

int qJumpRateLimiter(struct sk_buff* buffer) {
  long cycles_now = asm("rdtsc"); /* read cycle ctr */
  int level = buffer->priority;
  if (cycles_now > timeout) { /* new token alloc? */
    timeout += epoch_cycles;
    bucket[level] = tokens[level];
  }
  if (buffer->len > bucket[level]) {
    return DROP; /* tokens for epoch exhausted */
  }
  bucket[level] -= buffer->len;
  sendToHWQueue(buffer, level);
  return SENT;
}
```

Implementation

Linux TC

```
long bucket[NUM_QJUMP_LEVELS];
```

~36 cycles / packet

```
  int level = buffer->priority;
  if (cycles_now > timeout) { /* new token alloc? */
    timeout += epoch_cycles;
    bucket[level] = tokens[level];
  }
  if (buffer->len > bucket[level]) {
    return DROP; /* tokens for epoch exhausted */
  }
  bucket[level] -= buffer->len;
  sendToHWQueue(buffer, level);
  return SENT;
}
```

Implementation

Linux TC

```
long bucket[NUM_QJUMP_LEVELS];

```

~36 cycles / packet

```
  int level = buffer->priority;
  if (cycles_now > timeout) { /* new token alloc? */
```

Smart Buffer Sizing

```
  }
  if (buffer->len > bucket[level]) {
    return DROP; /* tokens for epoch exhausted */
  }
  bucket[level] -= buffer->len;
  sendToHWQueue(buffer, level);
  return SENT;
}
```

Implementation

Linux TC

```
long bucket[NUM_QJUMP_LEVELS];

```

~36 cycles / packet

```
  int level = buffer->priority;
  if (cycles_now > timeout) { /* new token alloc? */
```

Smart Buffer Sizing

```
  }
  if (buffer->len > bucket[level]) {
```

Unmodified Applications

```
  bucket[level] -= buffer->len;
  sendToHWQueue(buffer, level);
  return SENT;
}
```

Implementation

Linux TC

```
long bucket[NUM_QJUMP_LEVELS];

```

~36 cycles / packet

```
  int level = buffer->priority;
  if (cycles_now > timeout) { /* new token alloc? */
```

Smart Buffer Sizing

```
  }
  if (buffer->len > bucket[level]) {
```

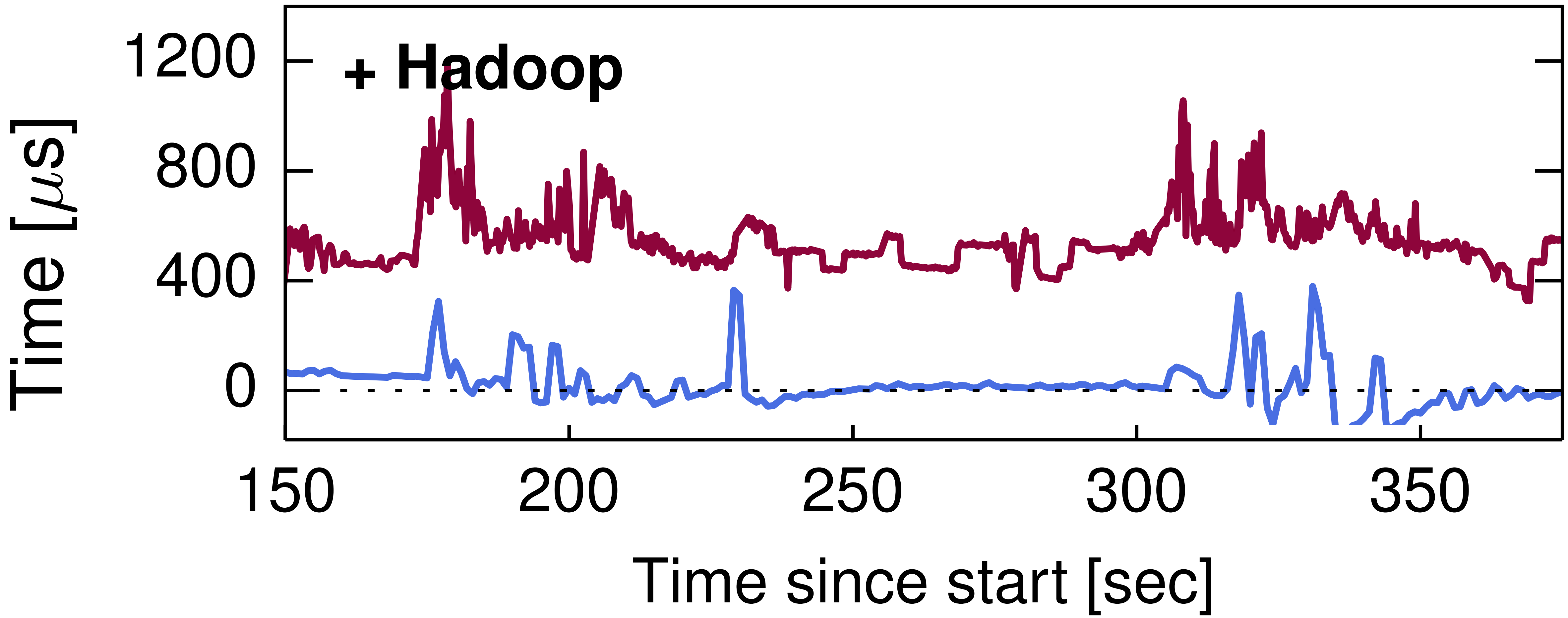
Unmodified Applications

```
  bucket[level] -= buffer->len;
```

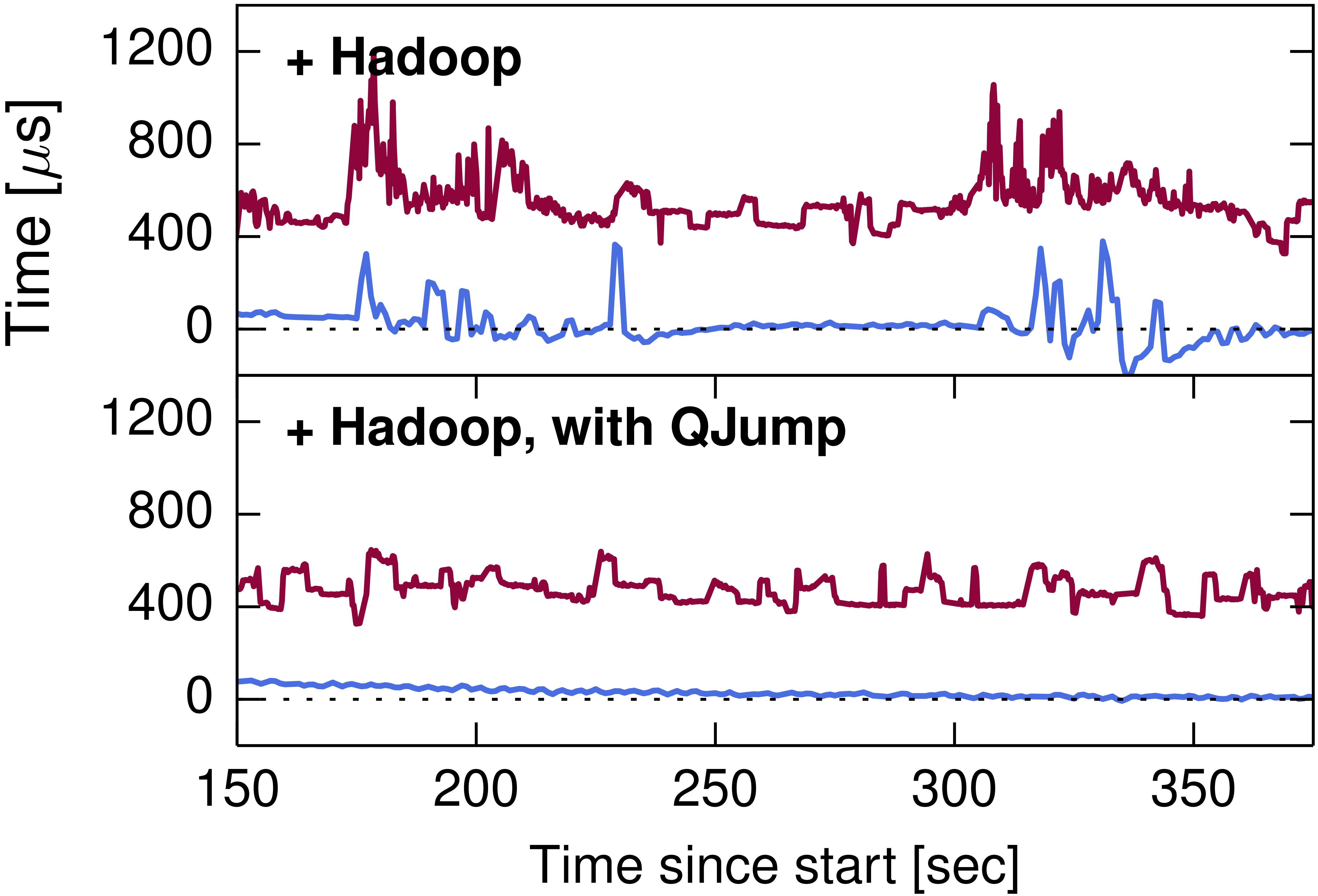
802.1 Q

```
}
```

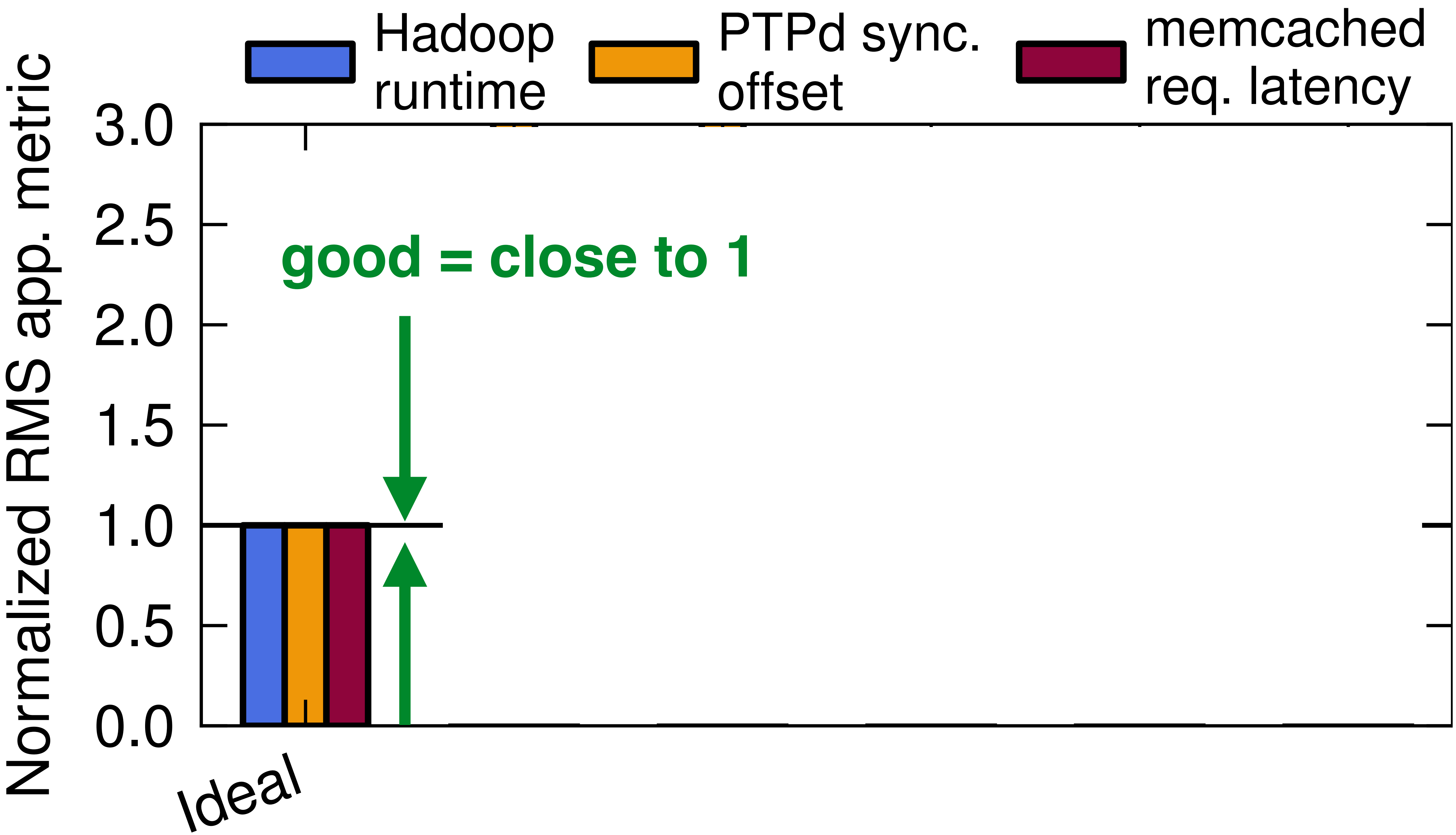
How well does it work?



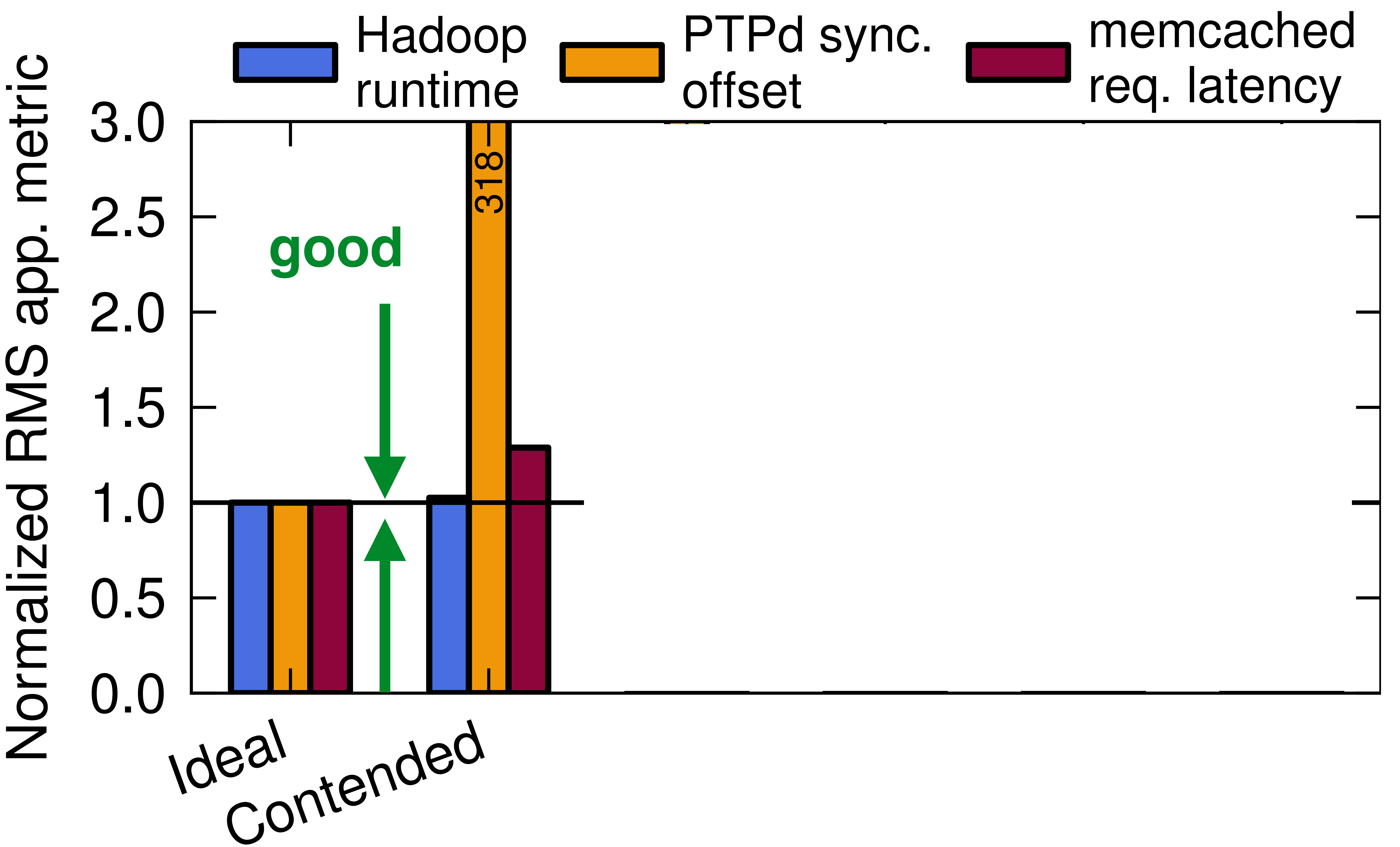
How well does it work?



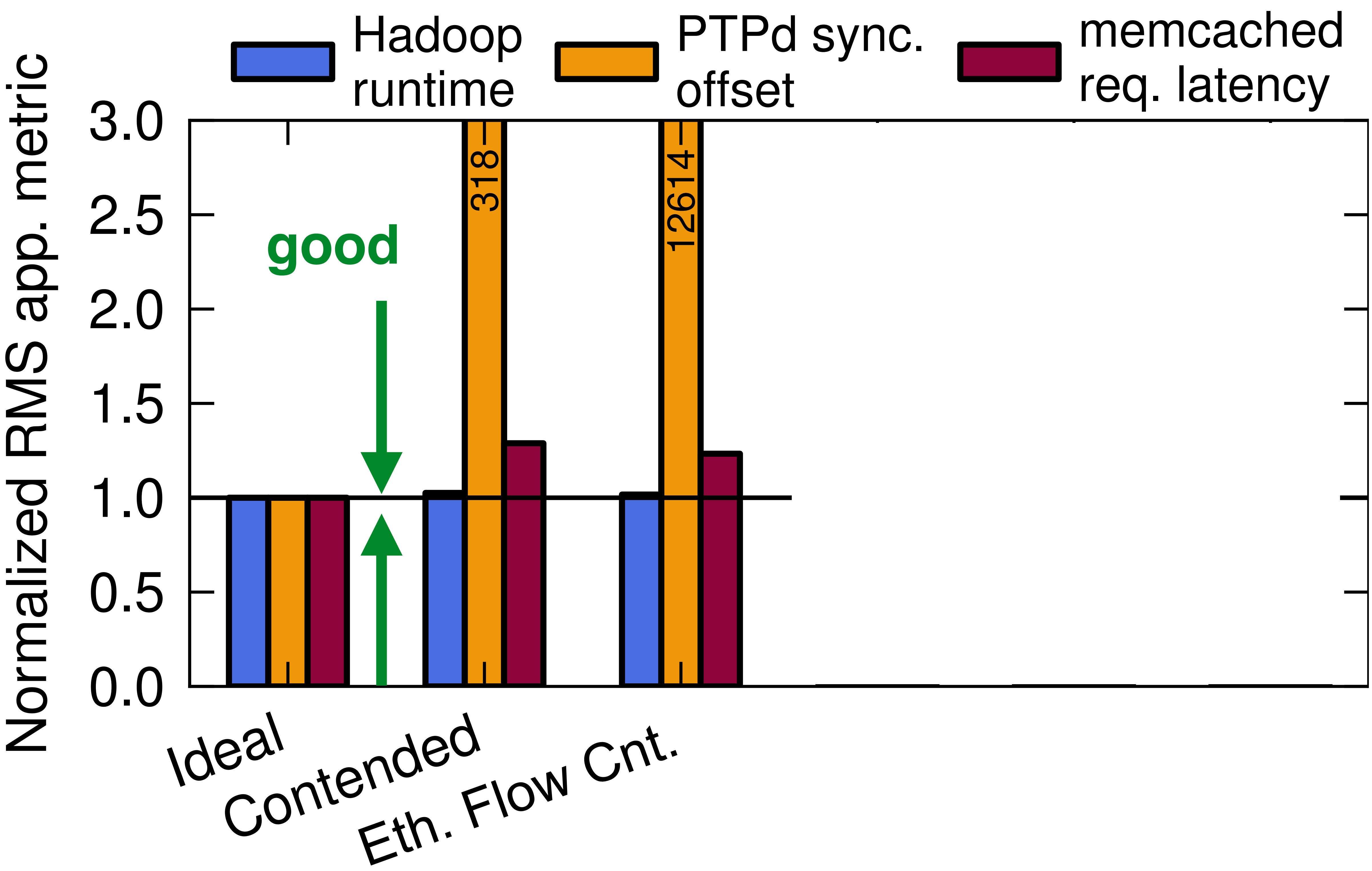
How does it compare?



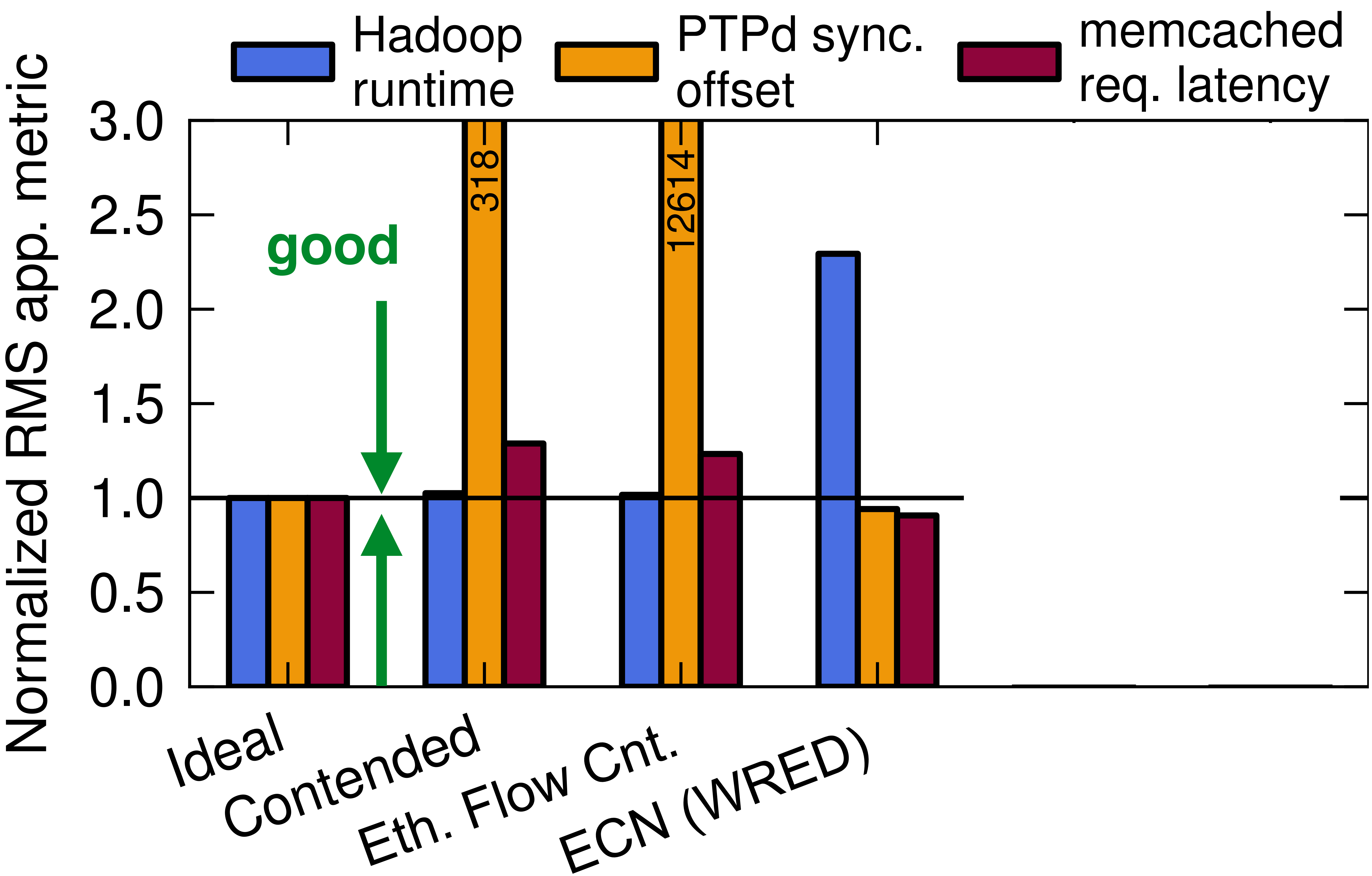
How does it compare?



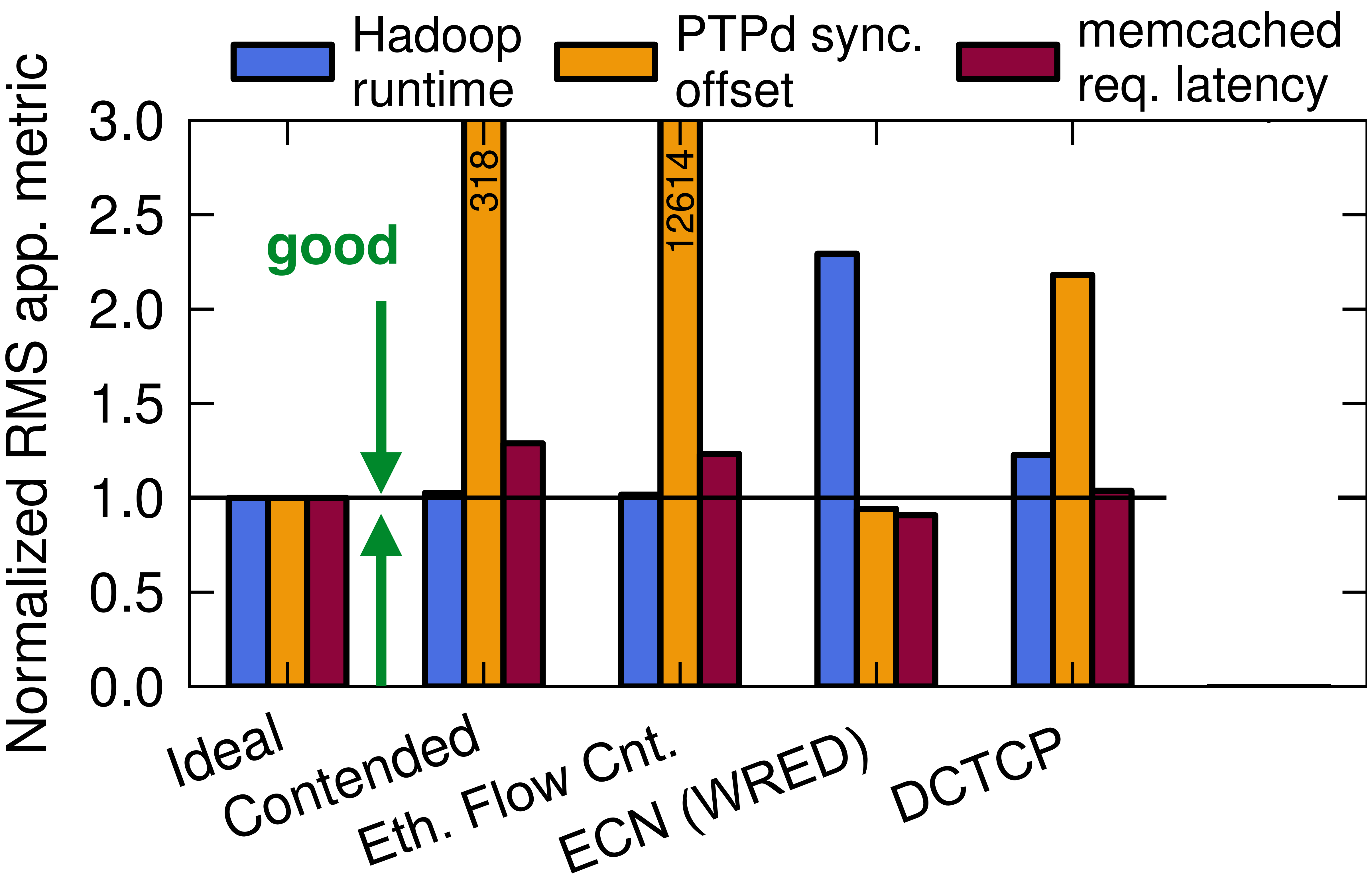
How does it compare?



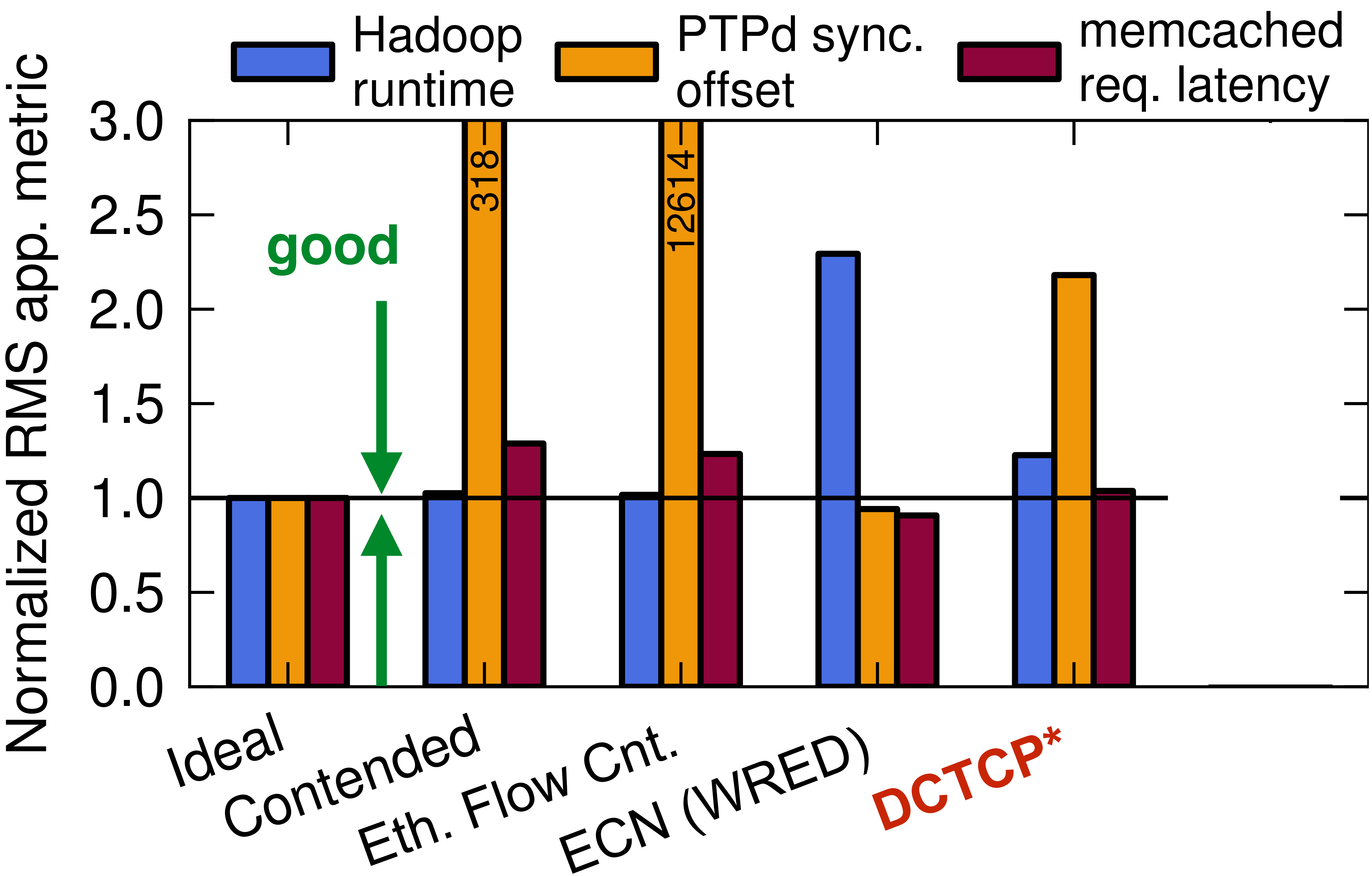
How does it compare?



How does it compare?

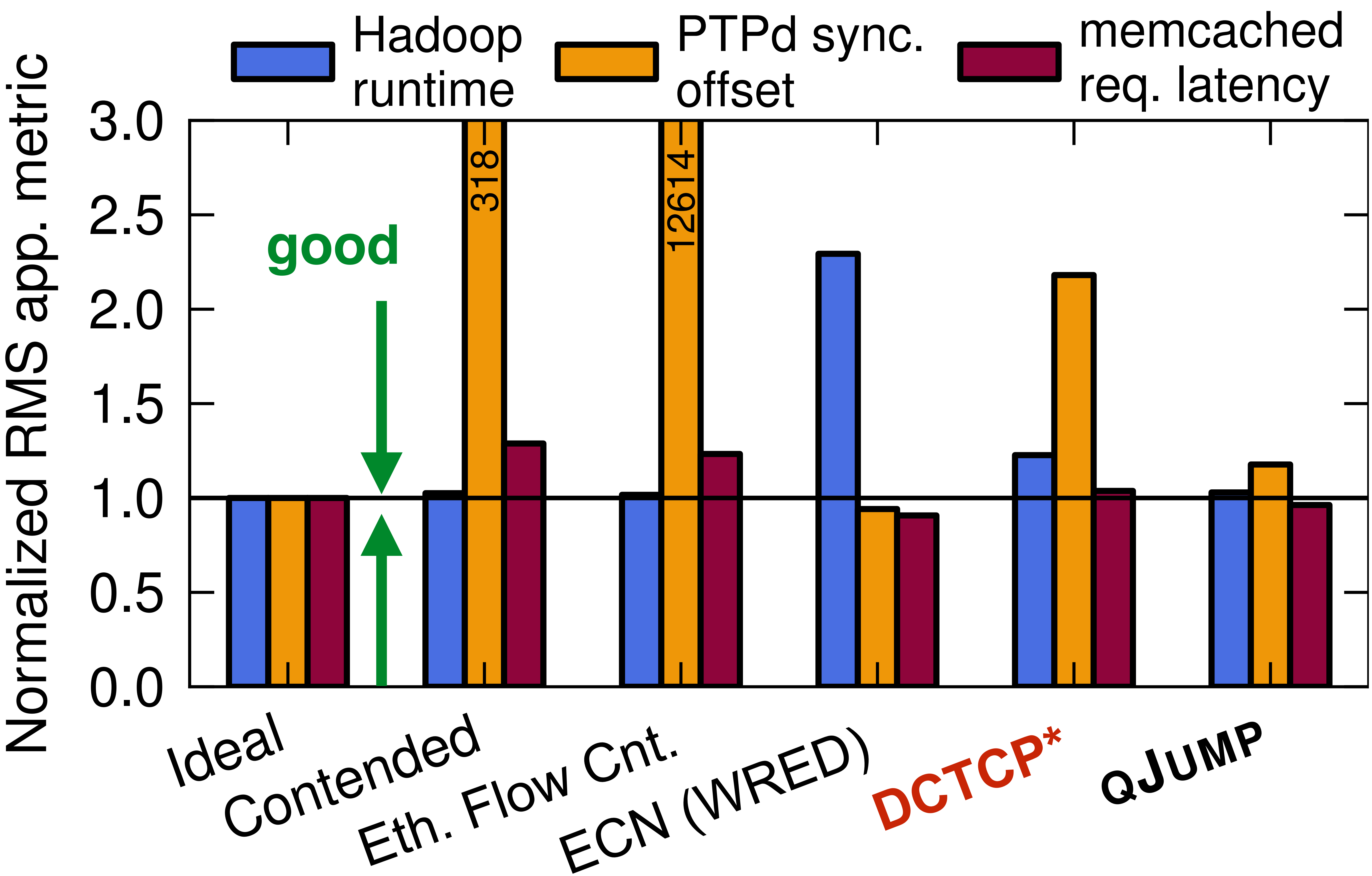


How does it compare?



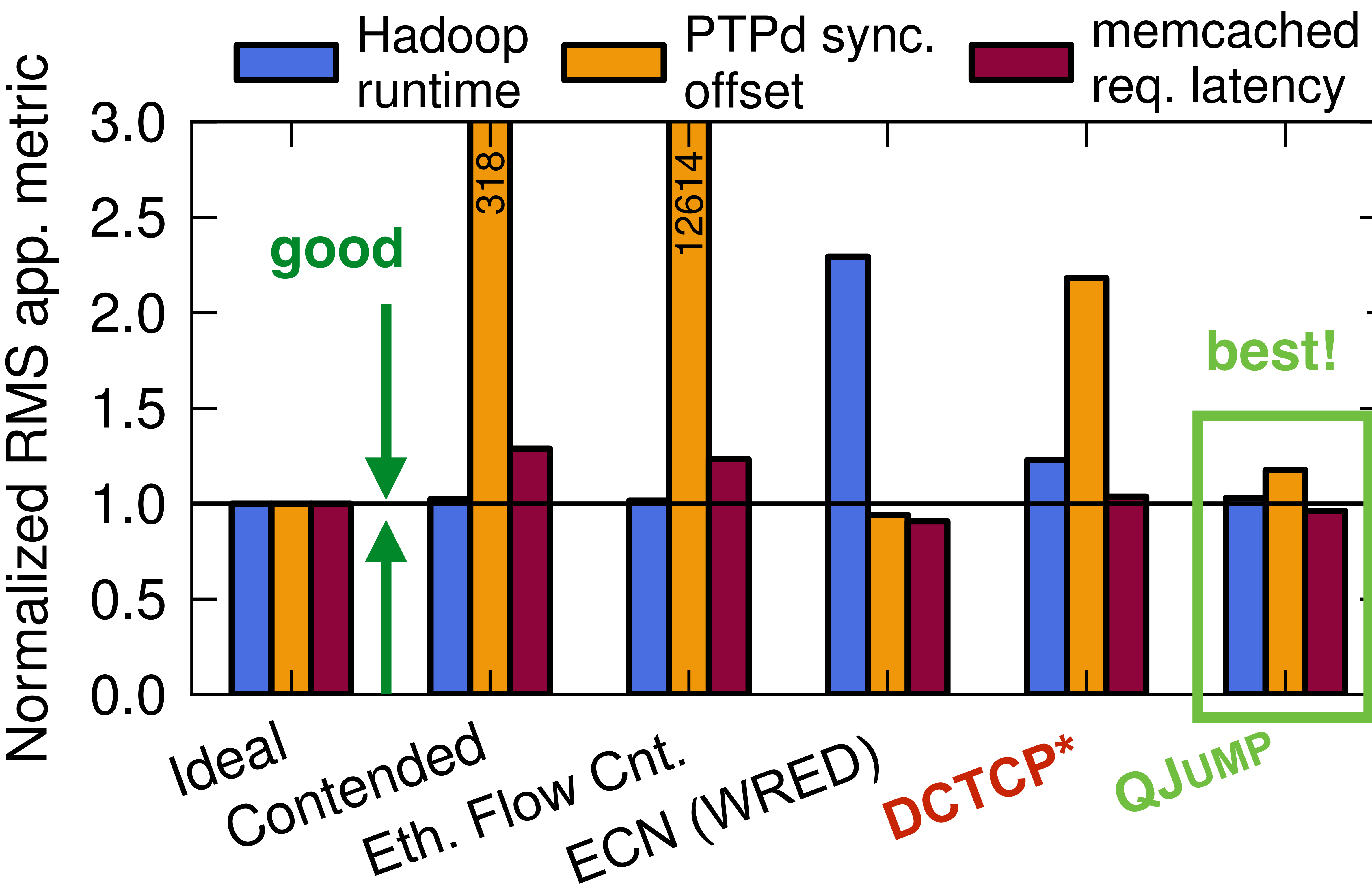
*currently requires kernel patch

How does it compare?



*currently requires kernel patch

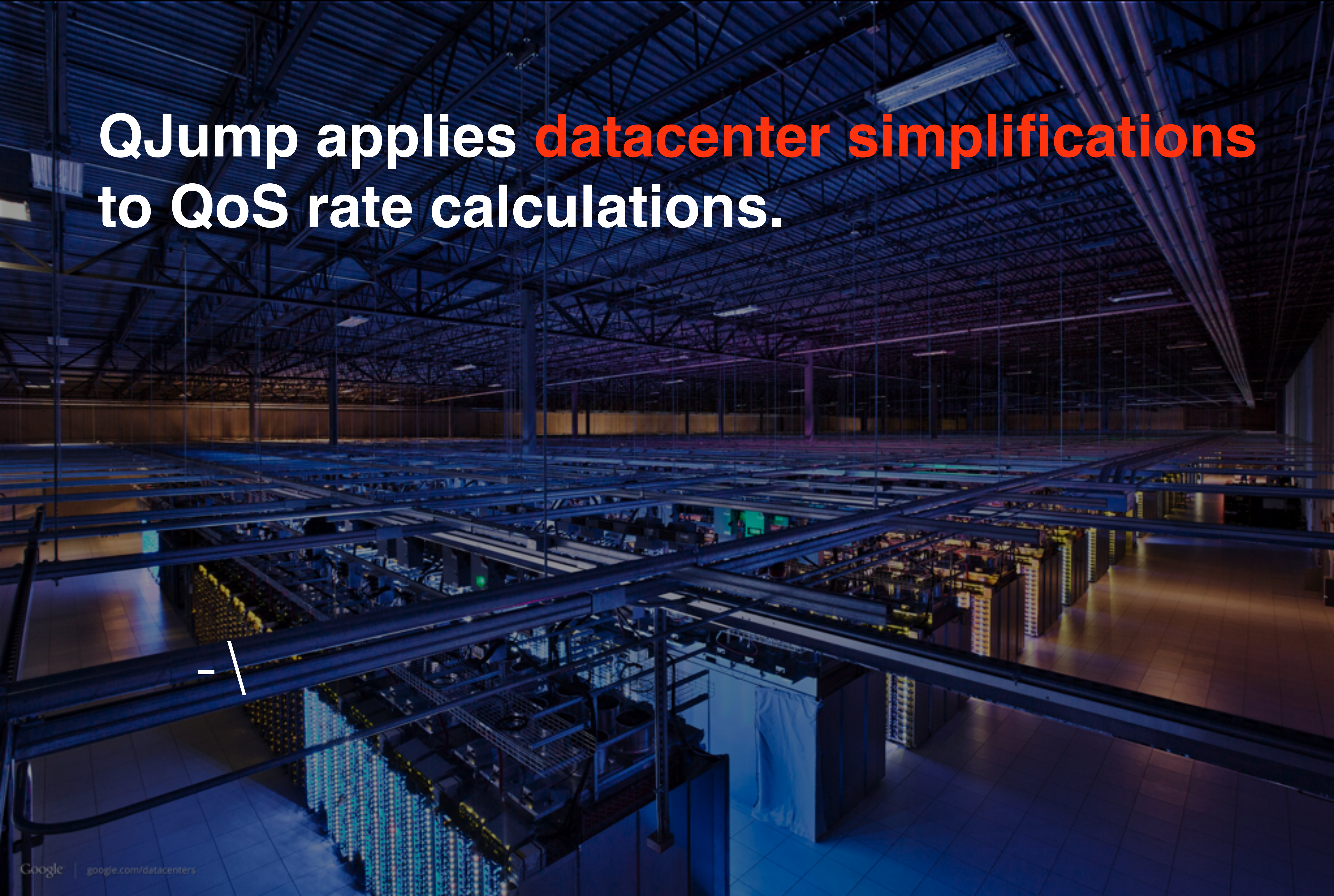
How does it compare?



*currently requires kernel patch

Conclusions

QJump applies **datacenter simplifications** to QoS rate calculations.



Conclusions

QJump applies datacenter simplifications to QoS rate calculations.

It provides service levels ranging from guaranteed latency through to line-rate throughput

- \

Conclusions

QJump applies datacenter opportunities to simplify QoS rate calculations.

It provides service levels ranging from guaranteed latency through to line-rate throughput

It can be deployed using without modifications to applications, kernel code or hardware.

Want to know more?

Queues don't matter when you can JUMP them!

Matthew P. Grosvenor Malte Schwarzkopf Ionel Gog Robert N. M. Watson
Andrew W. Moore Steven Hand[†] Jon Crowcroft

University of Cambridge Computer Laboratory

[†] now at Google, Inc.

Abstract

QJUMP is a simple and immediately deployable approach to controlling network interference in datacenter networks. Network interference occurs when congestion from throughput-intensive applications causes queueing that delays traffic from latency-sensitive applications. To mitigate network interference, QJUMP applies Internet QoS-inspired techniques to datacenter applications. Each application is assigned to a latency sensitivity level (or class). Packets from higher levels are rate-limited in the end host, but once allowed into the network can "jump-the-queue" over packets from lower levels. In settings with known node counts and link speeds, QJUMP can support service levels ranging from strictly bounded latency (but with low rate) through to line-rate throughput (but with high latency variance).

We have implemented QJUMP as a Linux Traffic Control module. We show that QJUMP achieves bounded latency and reduces in-network interference by up to 300×, outperforming Ethernet Flow Control (802.3x), ECN (WRED) and DCTCP. We also show that QJUMP improves average flow completion times, performing close to or better than DCTCP and pFabric.

1 Introduction

Many datacenter applications are sensitive to tail latencies. Even if as few as one machine in 10,000 is a straggler, up to 18% of requests can experience high latency [13]. This has a tangible impact on user engagement and thus potential revenue [8, 9].

One source of latency tails is *network interference*: congestion from throughput-intensive applications cause queueing that extends memcached request latency tails by 85 times the interference-free maximum (§2).

If memcached packets can somehow be prioritized to "jump-the-queue" over Hadoop's packets, memcached will no longer experience latency tails due to Hadoop. Of course, multiple instances of memcached may still interfere with *each other*, causing long queues or incast collapse [10]. If each memcached instance can be appropriately rate-limited at the origin, this too can be mitigated.

These observations are not new: QoS technologies like DiffServ [7] demonstrated that coarse-grained classification and rate-limiting can be used to control network latencies. Such schemes struggled for widespread deployment, and hence provided limited benefit [12]. However, unlike the Internet, datacenters have well-known network structures (i.e. host counts and link rates), and the bulk of the network is under the control of a single authority. In this environment, we can enforce system-wide policies, and calculate specific rate-limits which take into account worst-case behavior, ultimately allowing us to provide a guaranteed bound on network latency.

QJUMP implements these concepts in a minimal rate-limiting Linux kernel module and application utility. QJUMP has four key features. It:

1. resolves network interference for latency-sensitive applications **without sacrificing utilization** for throughput-intensive applications;
2. offers **bounded latency** to applications requiring low-rate, latency-sensitive messaging (e.g. timing, consensus and network control systems);
3. is simple and **immediately deployable**, requiring no changes to hardware or application code; and
4. **performs close to or better** than competing sys-

Want to know more?

Setup	50th%	99th%
one host, idle network	85	126µs
two hosts, shared switch	110	130µs
shared source host, shared egress port	228	268µs
shared dest. host, shared ingress port	125	278µs
shared host, shared ingress and egress	221	229µs
two hosts, shared switch queue	**1,920**	**2,100µs**

u can JUMP them!

Ionel Gog Robert N. M. Watson
† Jon Crowcroft
uter Laboratory

Abstract

QJUMP is a simple and immediately deployable approach to controlling network interference in datacenter networks. Network interference occurs when congestion from throughput-intensive applications causes queueing that delays traffic from latency-sensitive applications. To mitigate network interference, QJUMP applies Internet QoS-inspired techniques to datacenter applications. Each application is assigned to a latency sensitivity level (or class). Packets from higher levels are rate-limited in the end host, but once allowed into the network can "jump-the-queue" over packets from lower levels. In settings with known node counts and link speeds, QJUMP can support service levels ranging from strictly bounded latency (but with low rate) through to line-rate throughput (but with high latency variance).

We have implemented QJUMP as a Linux Traffic Control module. We show that QJUMP achieves bounded latency and reduces in-network interference by up to 300×, outperforming Ethernet Flow Control (802.3x), ECN (WRED) and DCTCP. We also show that QJUMP improves average flow completion times, performing close to or better than DCTCP and pFabric.

1 Introduction

Many datacenter applications are sensitive to tail latencies. Even if as few as one machine in 10,000 is a straggler, up to 18% of requests can experience high latency [13]. This has a tangible impact on user engagement and thus potential revenue [8, 9].

One source of latency tails is *network interference*: congestion from throughput-intensive applications

cause queueing that extends memcached request latency tails by 85 times the interference-free maximum (§2).

If memcached packets can somehow be prioritized to "jump-the-queue" over Hadoop's packets, memcached will no longer experience latency tails due to Hadoop. Of course, multiple instances of memcached may still interfere with *each other*, causing long queues or incast collapse [10]. If each memcached instance can be appropriately rate-limited at the origin, this too can be mitigated.

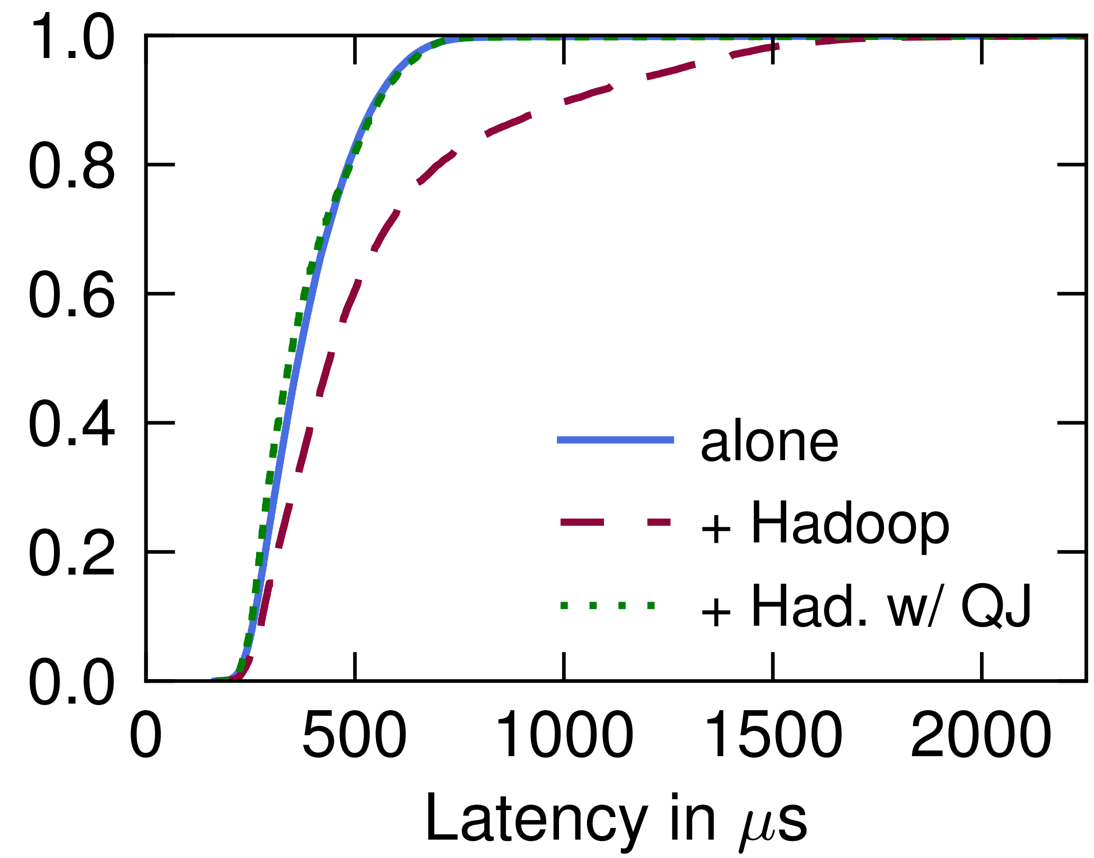
These observations are not new: QoS technologies like DiffServ [7] demonstrated that coarse-grained classification and rate-limiting can be used to control network latencies. Such schemes struggled for widespread deployment, and hence provided limited benefit [12]. However, unlike the Internet, datacenters have well-known network structures (i.e. host counts and link rates), and the bulk of the network is under the control of a single authority. In this environment, we can enforce system-wide policies, and calculate specific rate-limits which take into account worst-case behavior, ultimately allowing us to provide a guaranteed bound on network latency.

QJUMP implements these concepts in a minimal rate-limiting Linux kernel module and application utility. QJUMP has four key features. It:

1. resolves network interference for latency-sensitive applications **without sacrificing utilization** for throughput-intensive applications;
2. offers **bounded latency** to applications requiring low-rate, latency-sensitive messaging (e.g. timing, consensus and network control systems);
3. is simple and **immediately deployable**, requiring no changes to hardware or application code; and
4. **performs close to or better** than competing sys-

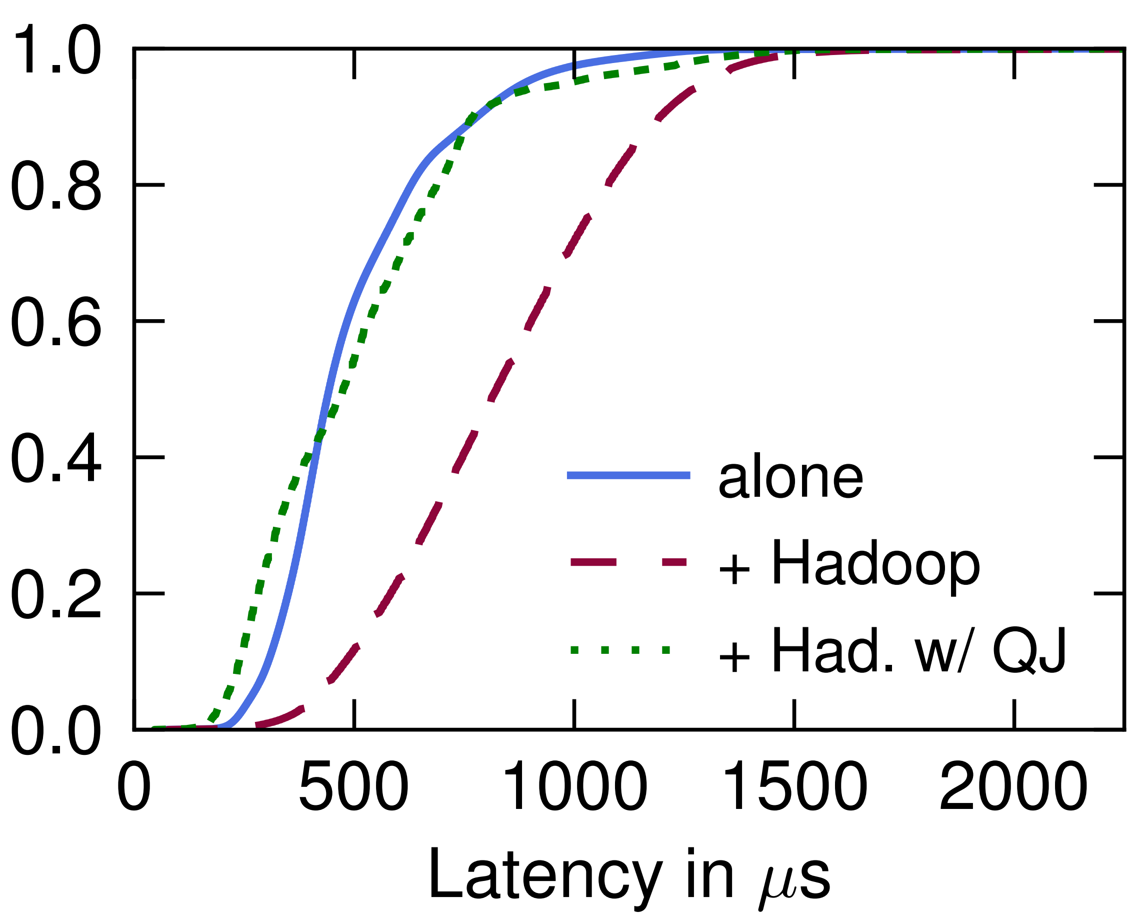
Want to know more?

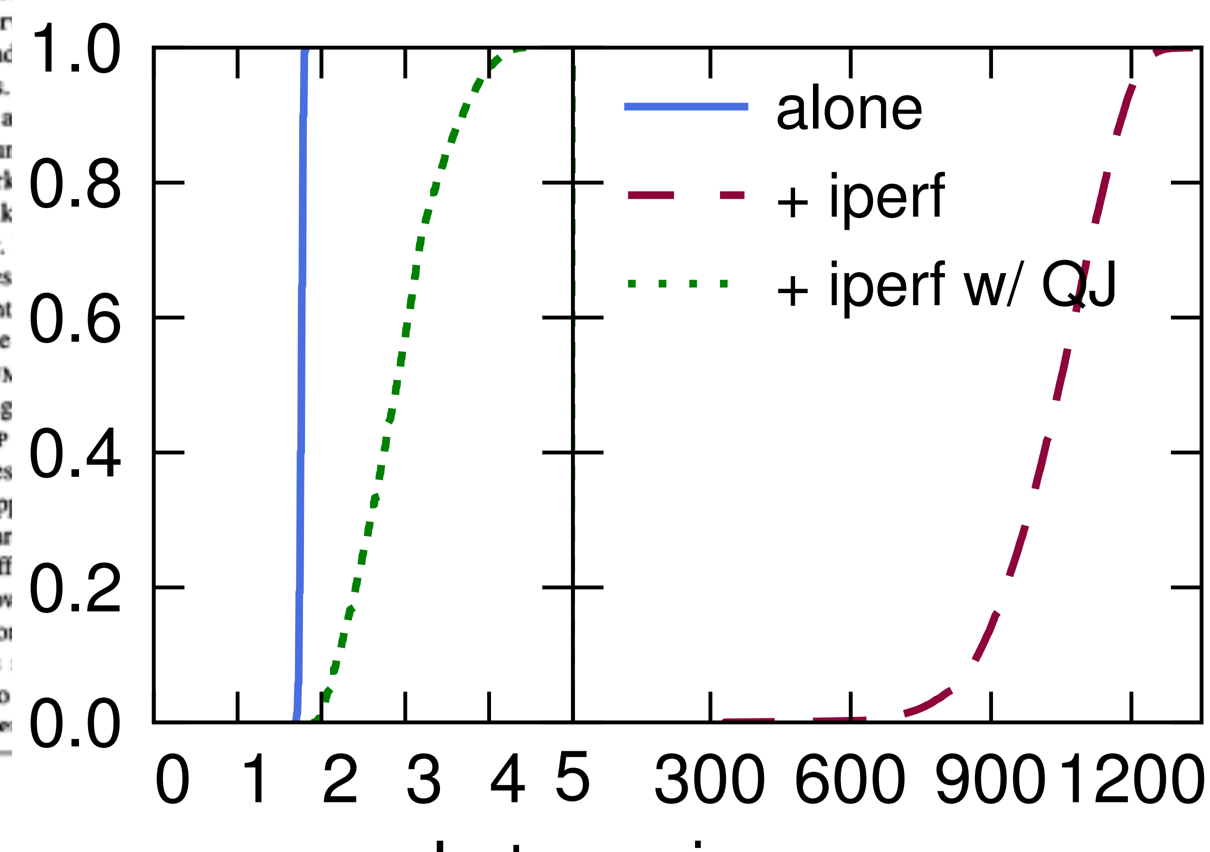
Setup	$50^{th}\%$	$99^{th}\%$
one host, idle network	85	126µs
two hosts, shared switch	110	130µs
shared source host, shared egress port	228	268µs
shared dest. host, shared ingress port	125	278µs
shared host, shared ingress and egress	221	229µs
two hosts, shared switch queue	**1,920**	**2,100µs**



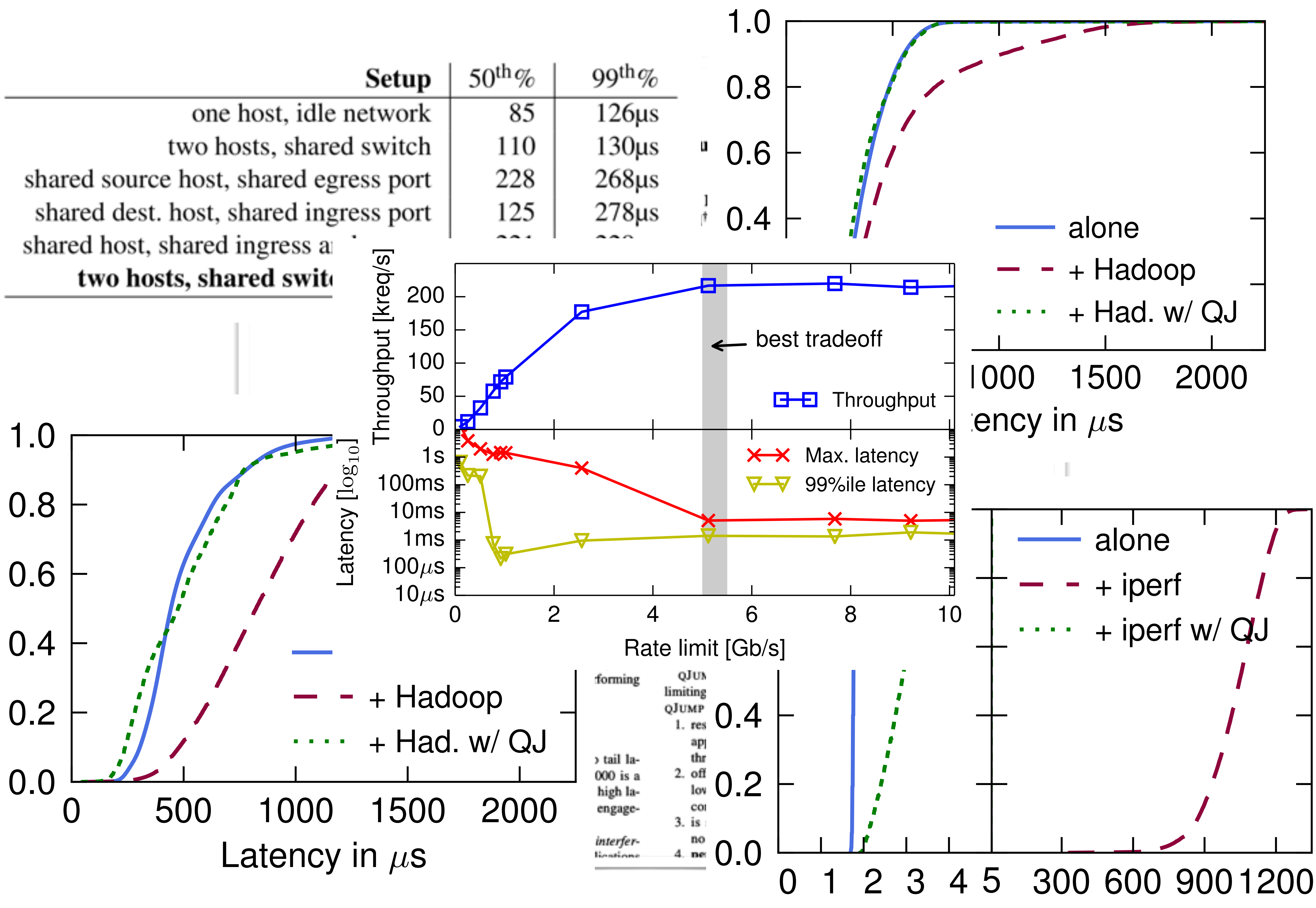
Abstract

QJUMP is a simple and immediately deployable approach to controlling network interference in datacenter networks. Network interference occurs when congestion

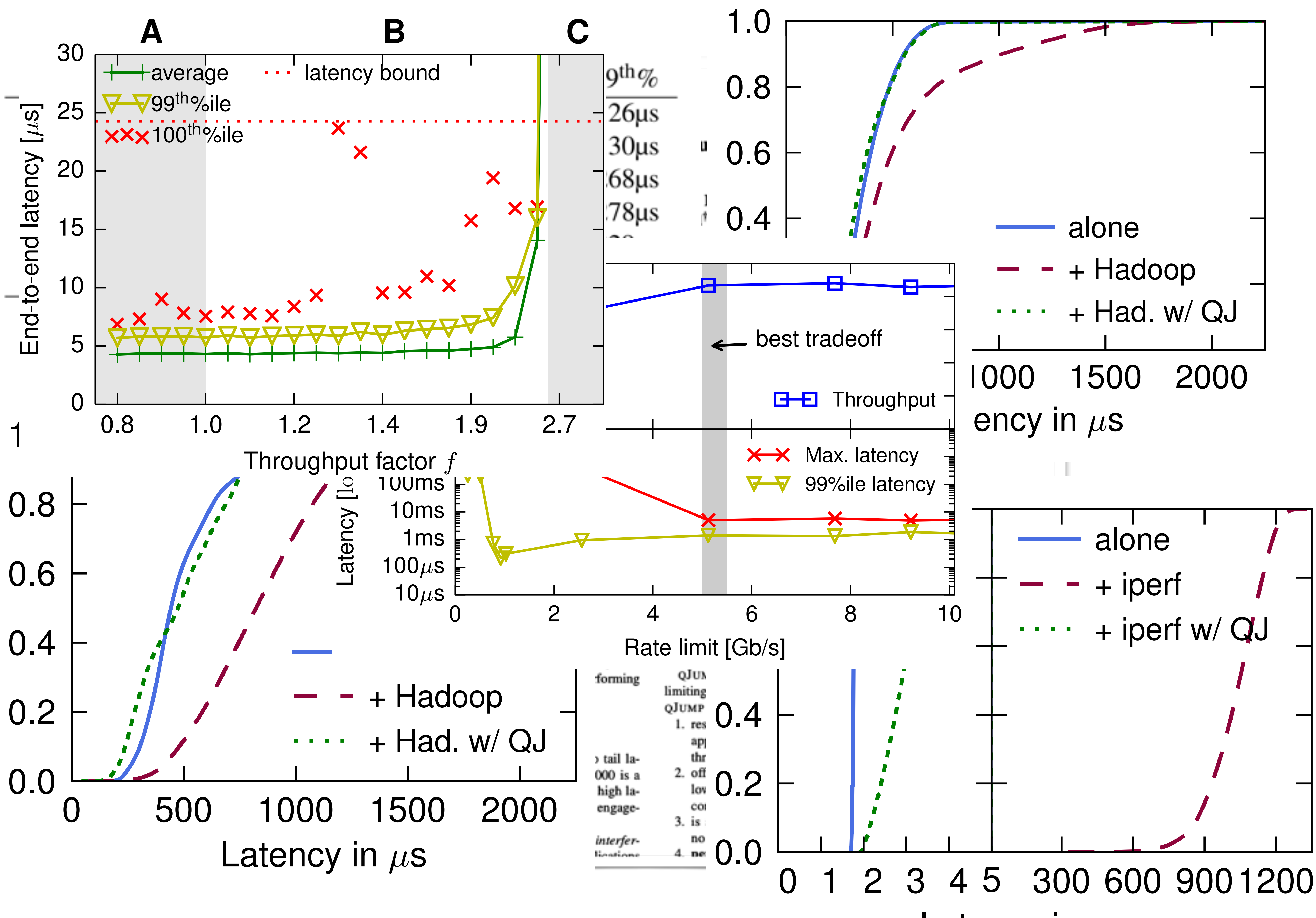




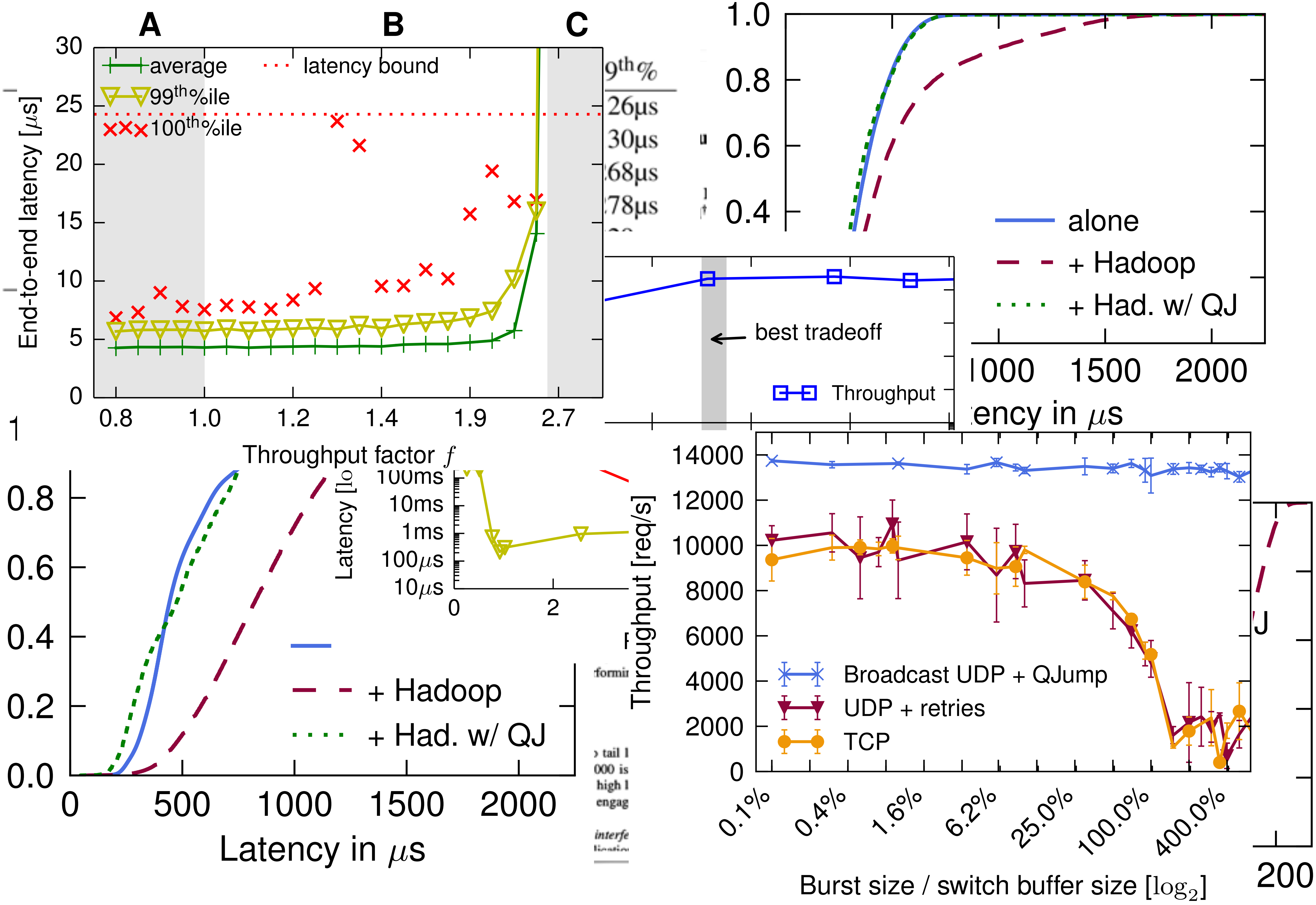
Want to know more?



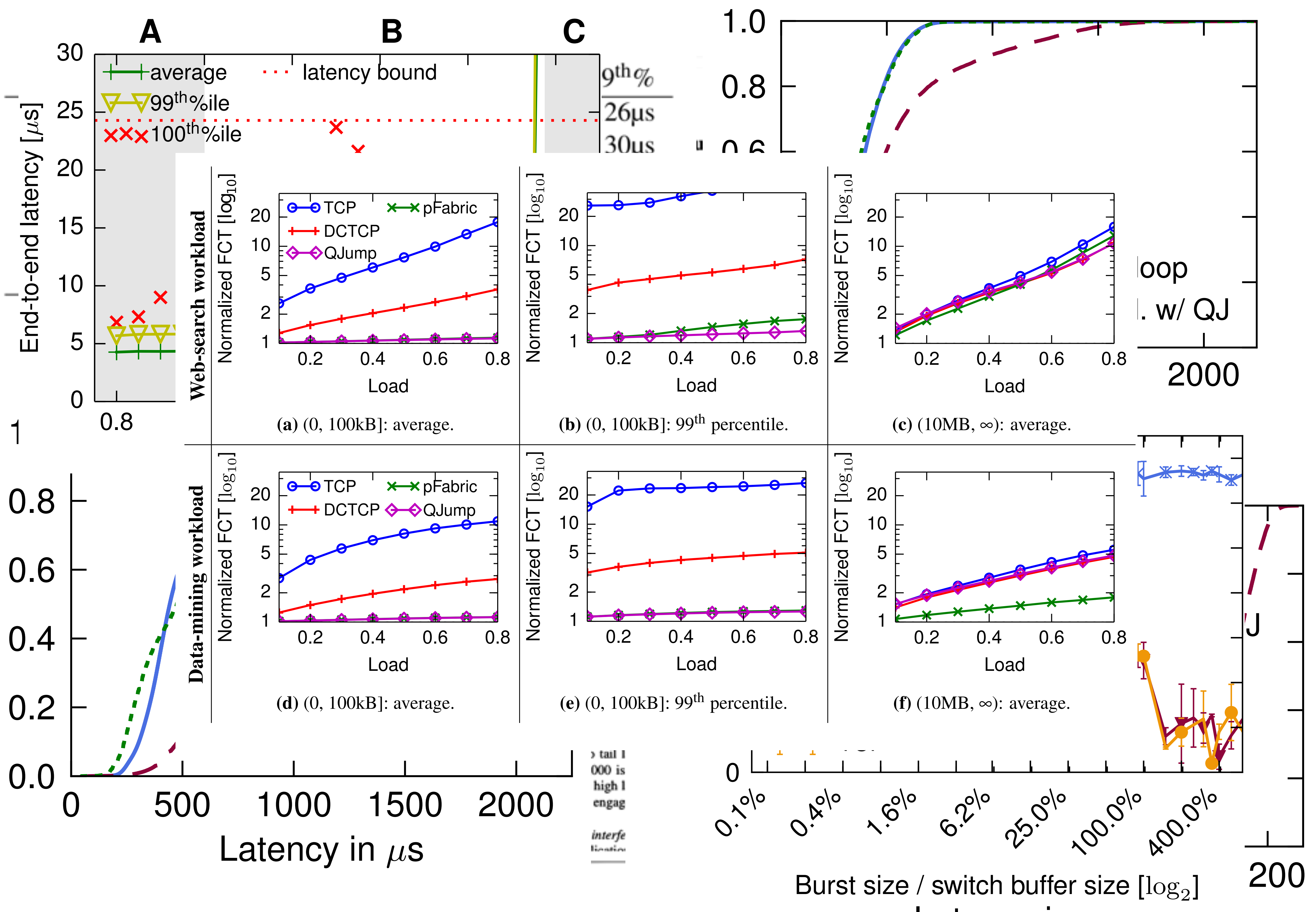
Want to know more?



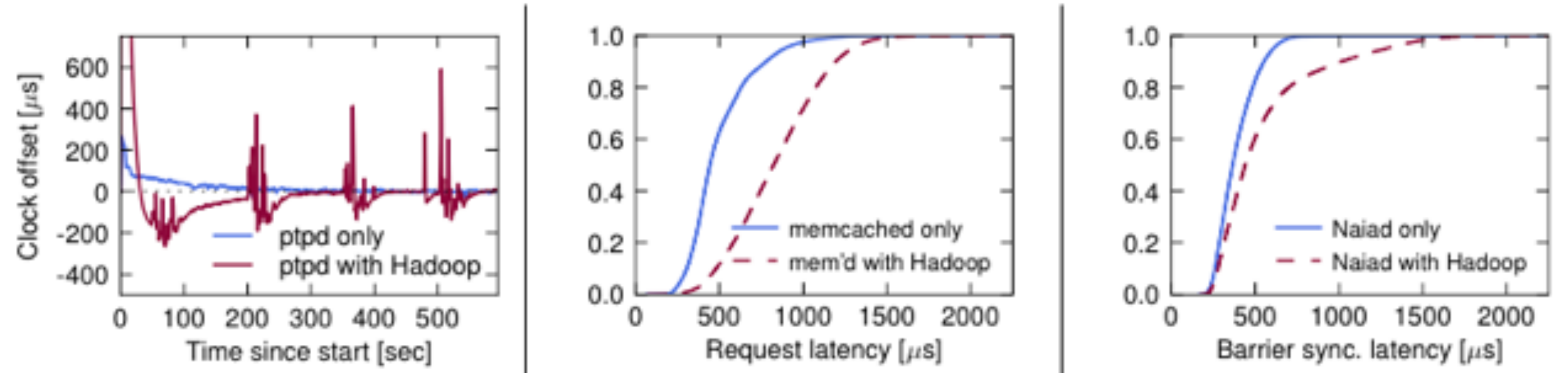
Want to know more?



Want to know more?



Just one more thing…



(a) Timeline of PTP synchronization offset. **(b)** CDF of memcached request latencies. **(c)** CDF of Naiad barrier sync. latencies.

Figure 1: Motivating experiments: Hadoop traffic interferes with *(a)* PTPd, *(b)* memcached and *(c)* Naiad traffic.

Setup	50th%	99th%
one host, idle network	85	126µs
two hosts, shared switch	110	130µs
shared source host, shared egress port	228	268µs
shared dest. host, shared ingress port	125	278µs
shared host, shared ingress and egress	221	229µs
two hosts, shared switch queue	**1,920**	**2,100µs**

Table 1: Median and 99th percentile latencies observed as `ping` and `iperf` share various parts of the network.

2 Motivation

We begin by showing that shared switch queues are the primary source of network interference. We then quantify the extent to which network interference impacts application-observable metrics of performance.

2.1 Where does the latency come from?

Network interference may occur at many places on the

in §6) and measure the effects.

1. Clock Synchronization Precise clock synchronization is important to distributed systems such as Google's Spanner [11]. PTPd offers microsecond-granularity time synchronization from a time server to machines on a local network. In Figure 1a, we show a timeline of PTPd synchronizing a host clock on both an idle network and when sharing the network with Hadoop. In the shared case, Hadoop's shuffle phases causes queueing, which delays PTPd's synchronization packets. This causes PTPd to temporarily fall 200–500µs out of synchronization, 50× worse than on an idle network.

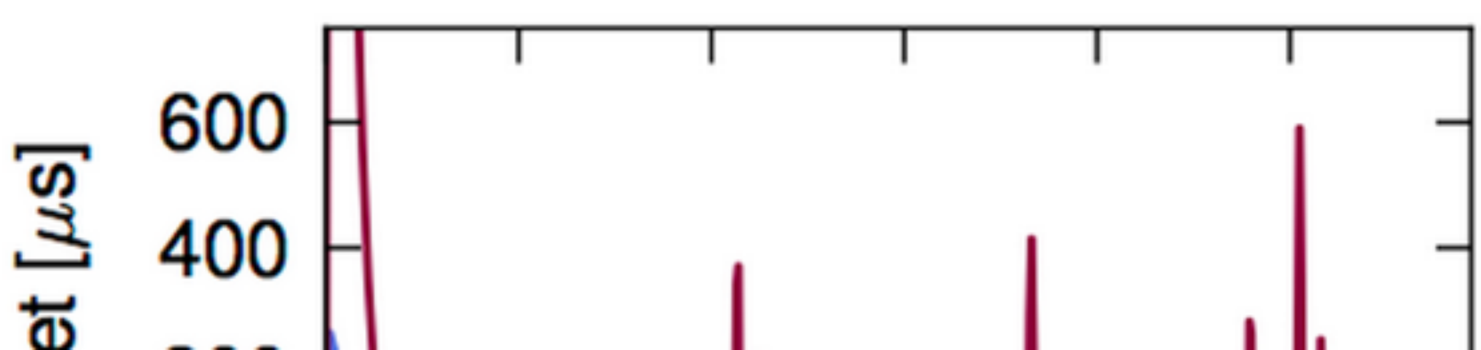
2. Key-value Stores Memcached is a popular in-memory key-value store used by Facebook and others to store small objects for quick retrieval [25]. We benchmark memcached using the `memaslap` load generator[2] and measure the request latency. Figure 1b shows the distribution of request latencies on an idle network and a

Just one more thing…

Figure 1a / 5

Figure 1a (page 2) is used as a motivational experiment to show that Hadoop MapReduce is capable of interfering with the behaviour of precision time protocol. This figure is repeated in Figure 5 (page 8) in a slightly different form, combined with results from memcached combined. In this case, the figure shows that QJump is capable of resolving interference in PTPd as well as memchaced.

Figure 1a

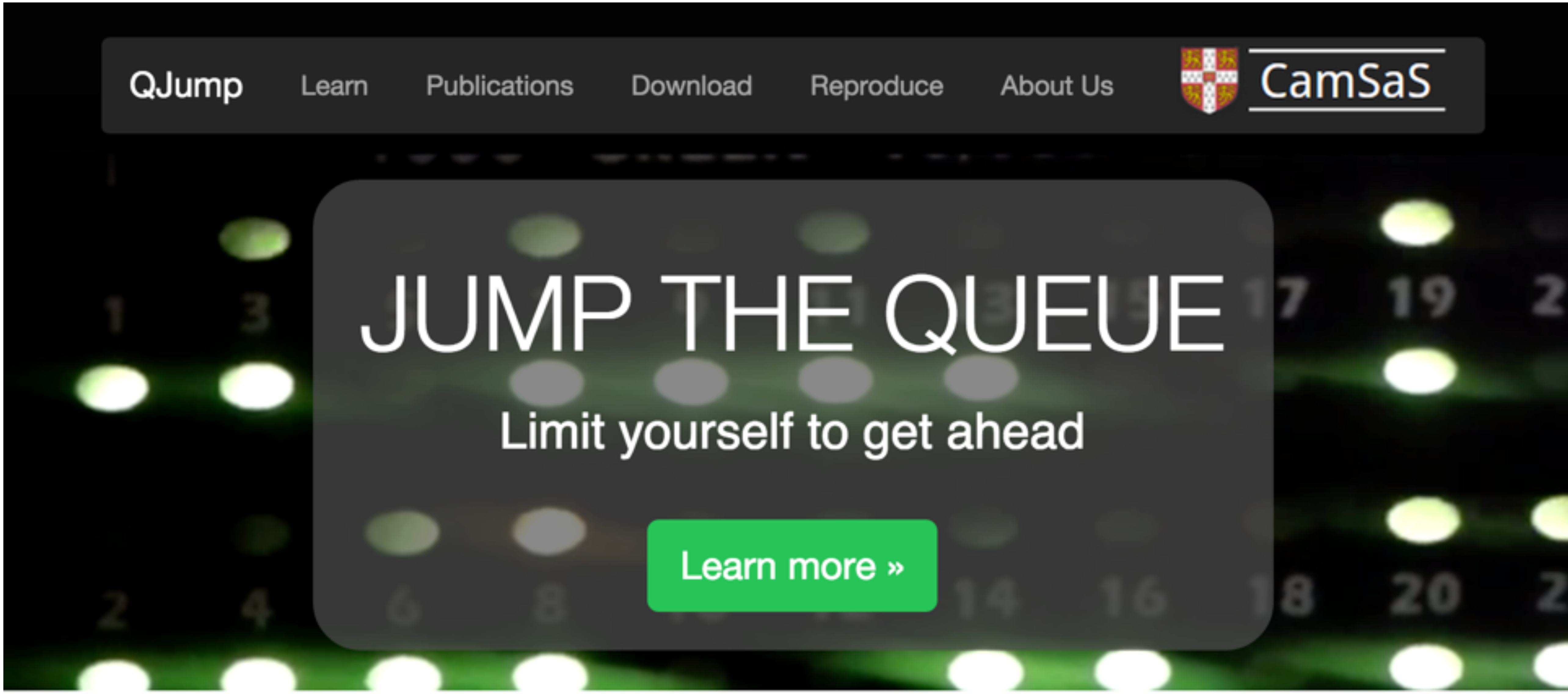


Just one more thing...

NSDI 2015 - Queues don't matter when you can Jump them!

Figure	Description
Fig. 1a	PTPd synchronization offset with and without sharing the network with Hadoop Map-Reduce
Fig. 1b	Memcached request latencies with and without sharing the network with Hadoop Map-Reduce
Fig. 1c	Naiad barrier synchronization latencies with and without sharing the network with Hadoop Map-Reduce
Tbl. 1	Latencies observed as ping and iperf share various parts of the network
Fig. 3a	Ping packet latency across a switch with and without QJump enabled
Fig. 3b	QJump reducing memcached request latency in the presence of Hadoop Map-Reduce traffic
Fig. 3c	QJump fixes Naiad barrier synchronization latency in the presence of Hadoop Map-Reduce traffic
Fig. 5	PTPd, memcached and Hadoop sharing a cluster, with and without QJump enabled
Fig. 6	QJump offers constant two phase commit throughput even at high levels of network interference
Fig. 7	QJump comes closest to ideal performance when compared with Ethernet Flow Control, ECN and DCTCP
Fig. 9	Normalized flow completion times in a 144-host simulation. QJump outperforms stand-alone TCP, DCTCP and pFabric for small flows
Fig. 10	Memcached throughput and latency as a function of the QJump rate limits
Fig. 11	Latency bound validation of QJump with 60 host generating full rate, fan in traffic

Just one more thing…



Guaranteed latency in datacenter networks

QJump offers a range of network service levels, from guaranteed latency for low-rate, latency-sensitive network coordination services to line-rate throughput

Conclusions

QJump applies datacenter opportunities to simplify QoS rate calculations.

It provides levels of service from guaranteed latency through to line-rate throughput

It can be deployed using without modifications to applications, kernel code or hardware.

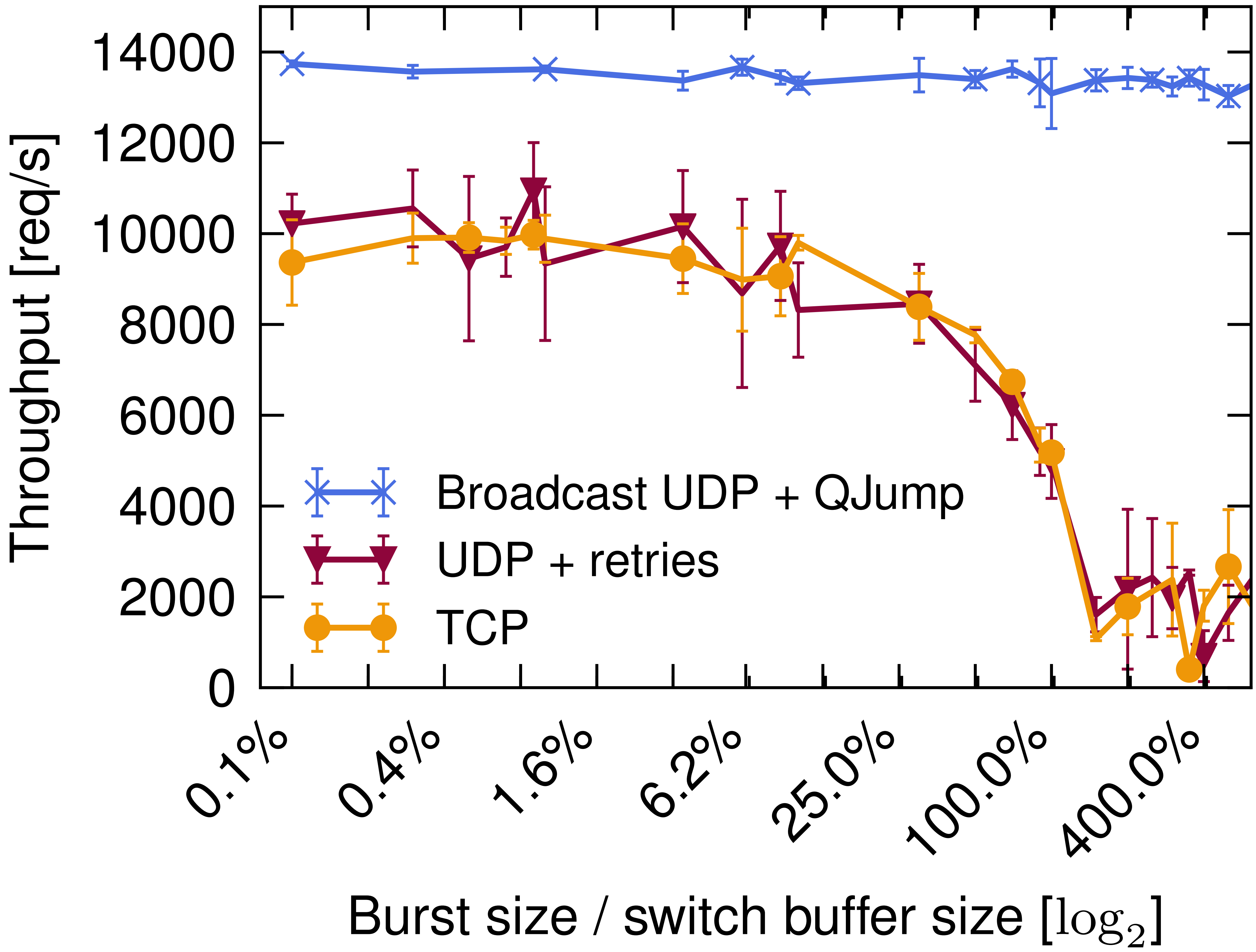
All source data, patches and source code at

http://camsas.org/qjump

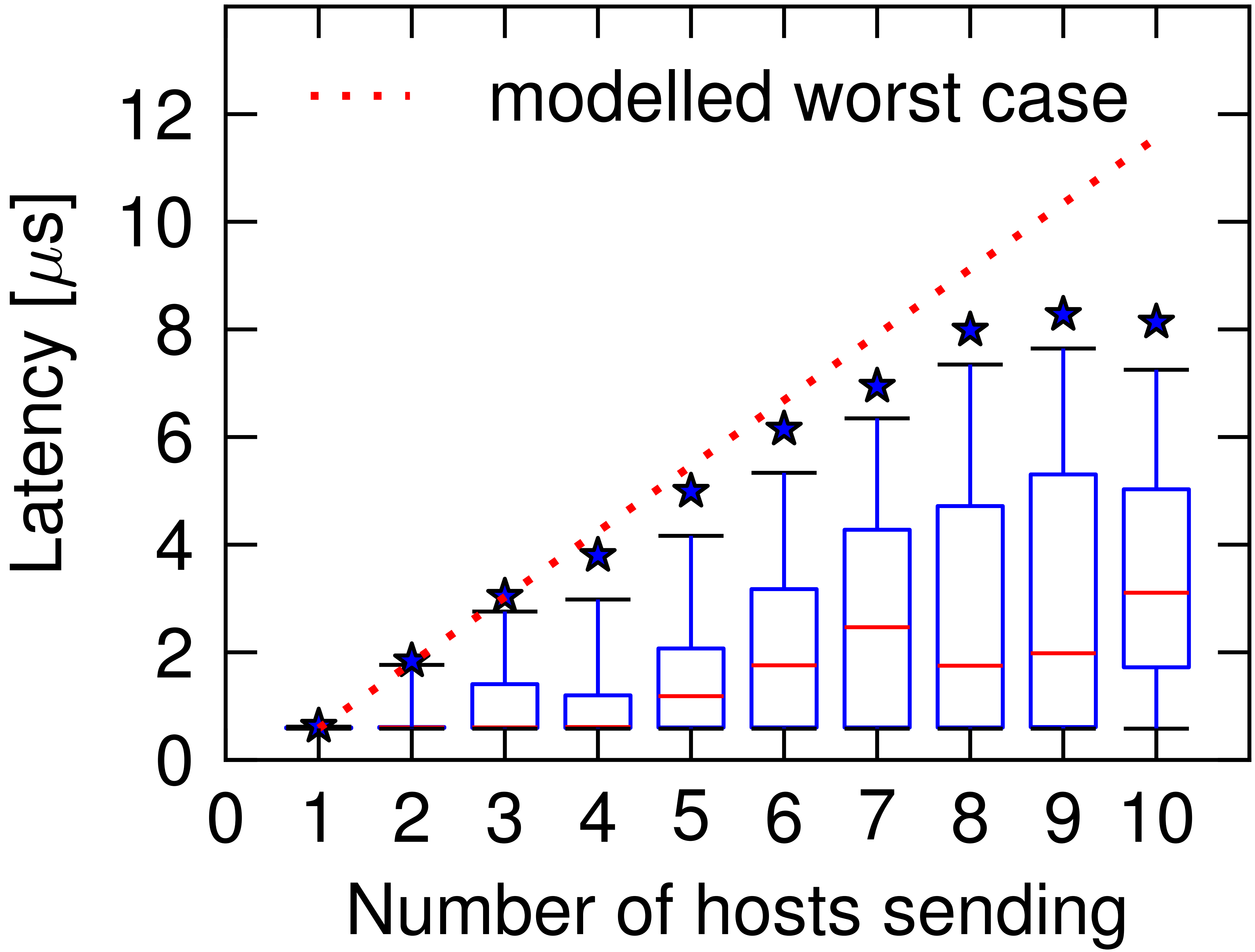
This work was jointly supported by the EPSRC INTERNET Project EP/H040536/1 and the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8750-11-C-0249. The views, opinions, and/or findings contained in this article/presentation are those of the author/presenter and should not be interpreted as representing the official views or policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the Department of Defense.

Backup Slides

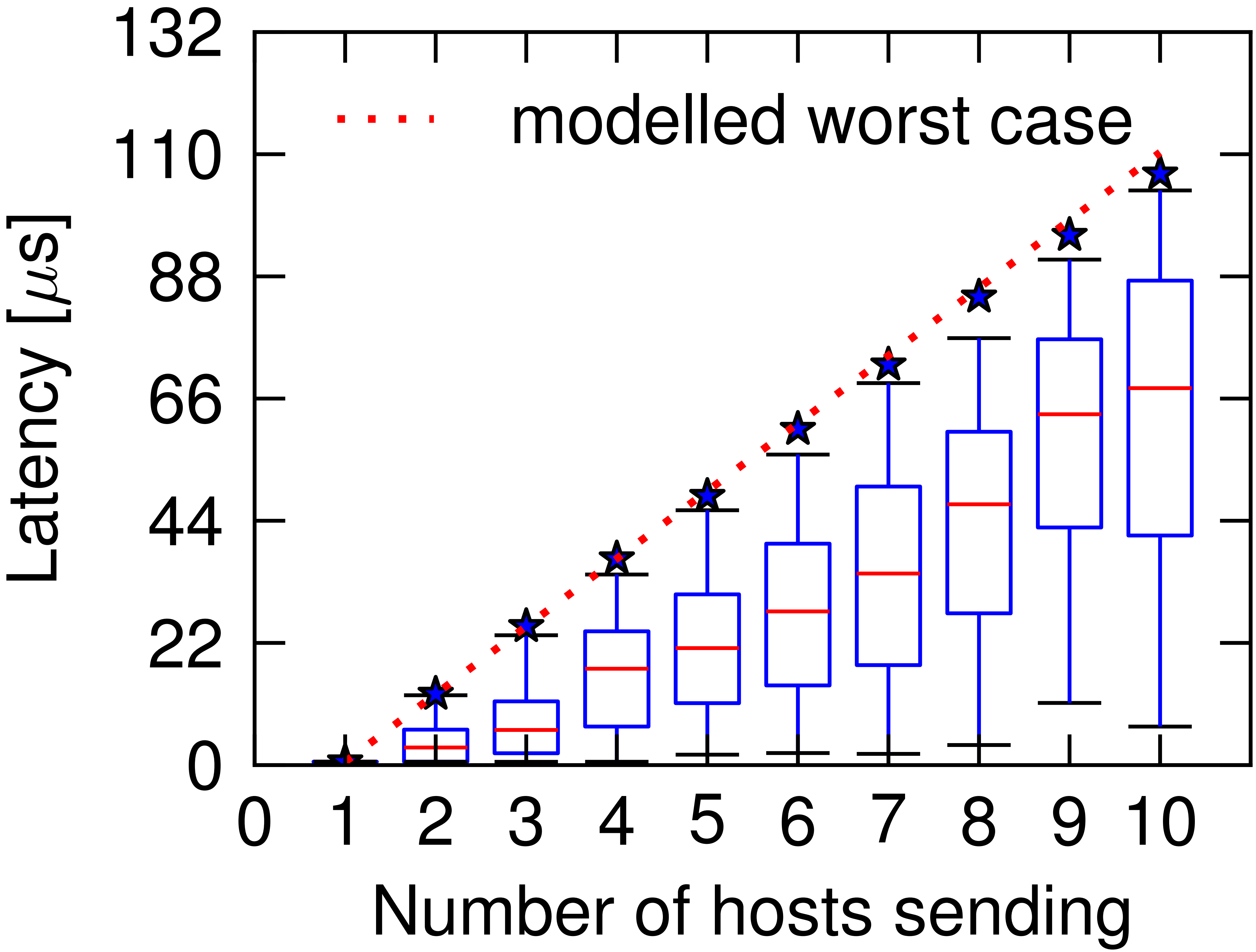
What is it good for?



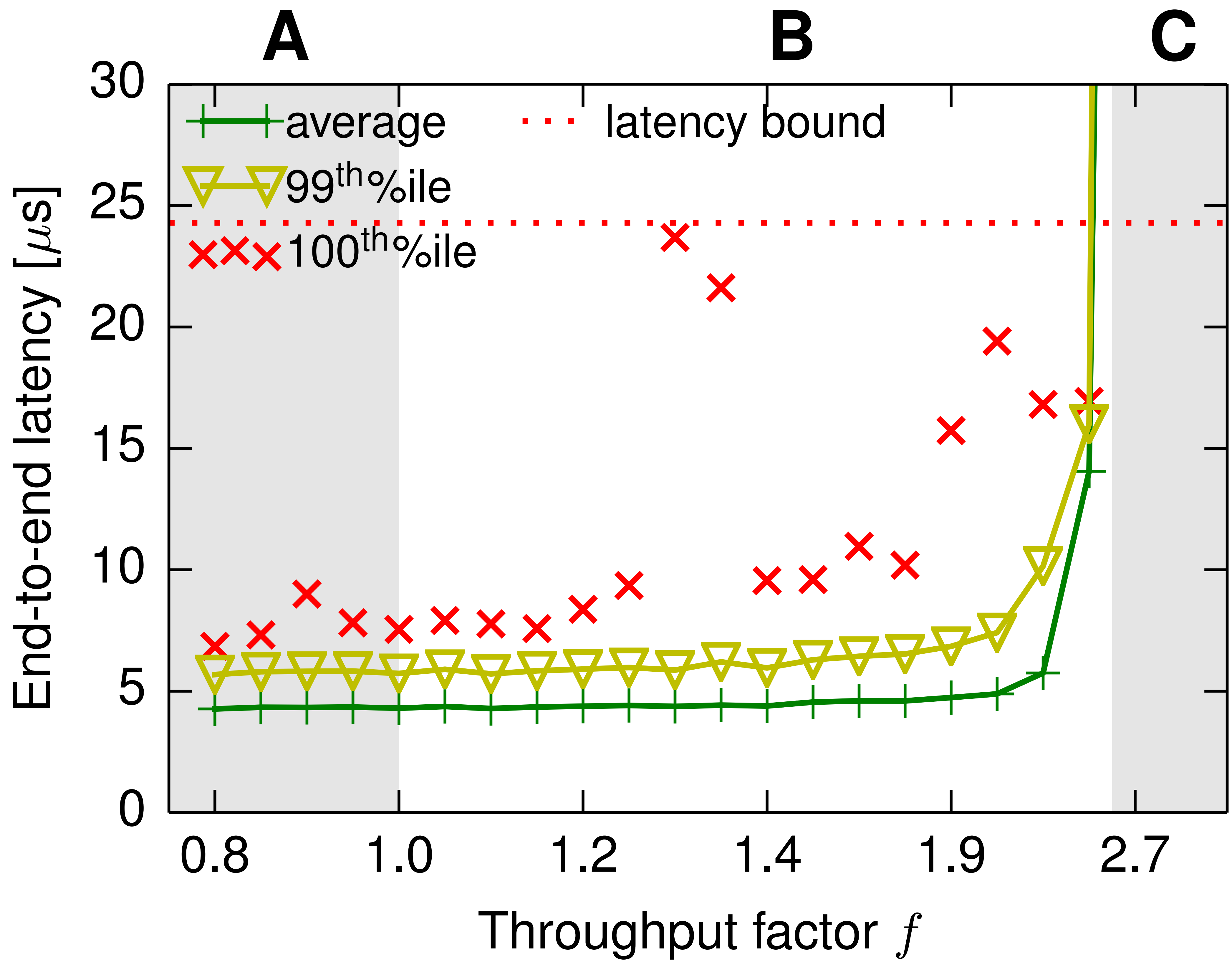
Accuracy of Switch Model



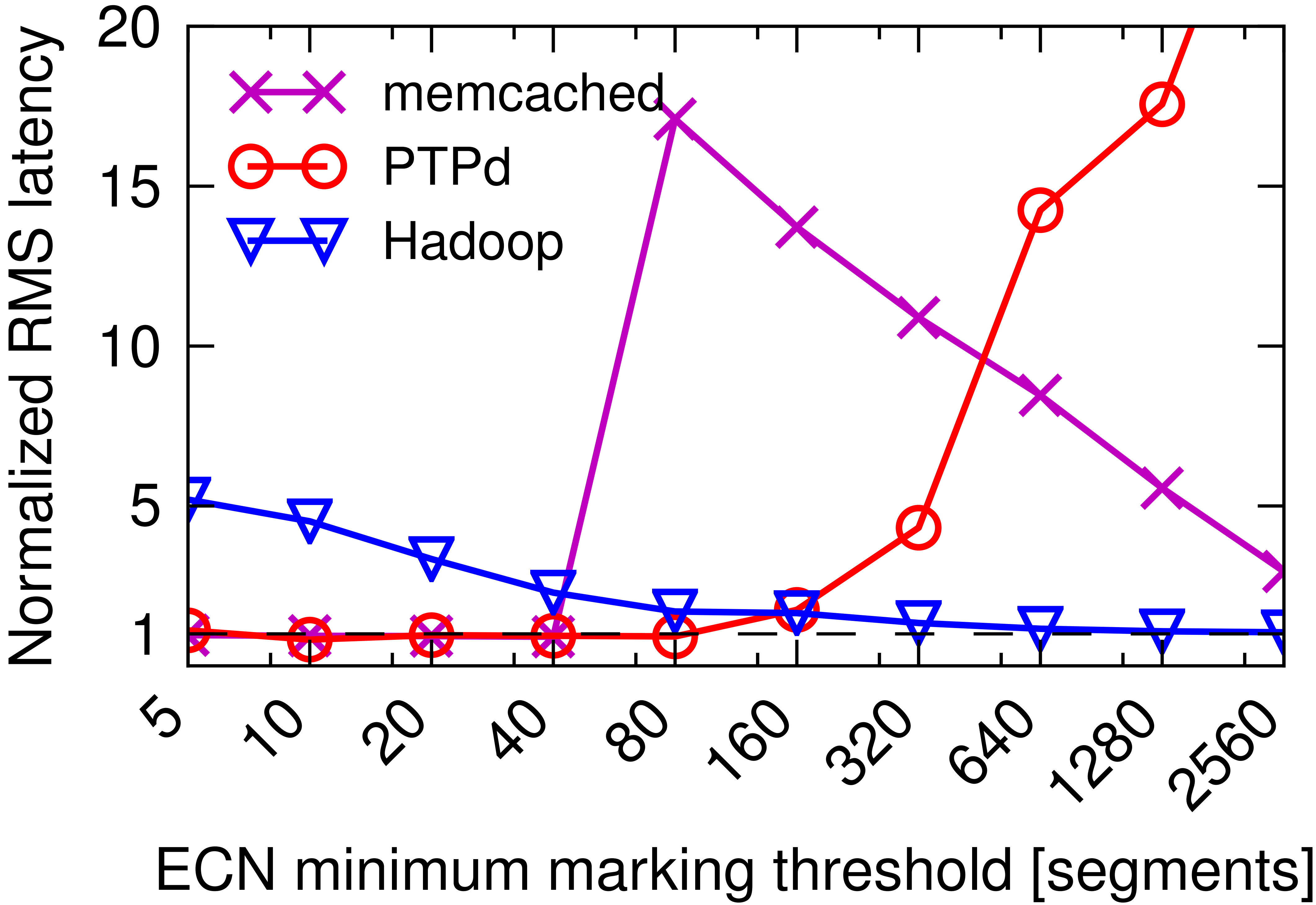
Accuracy of Switch Model



Sensitivity to *f*



ECN WRED Config.

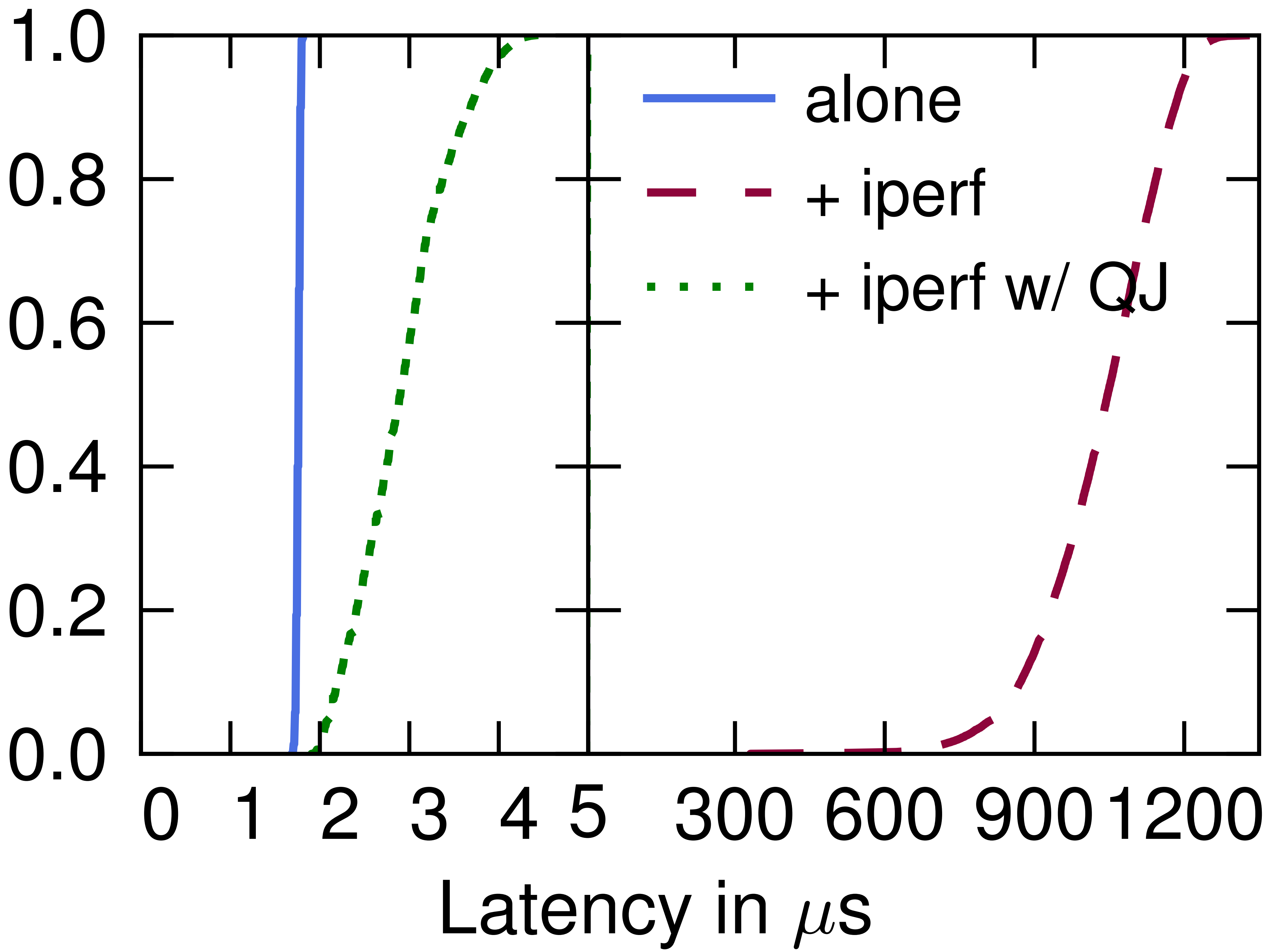


Host based interference?

Setup	$50^{th}\%$	$99^{th}\%$
one host, idle network	85	126µs
two hosts, shared switch	110	130µs
shared source host, shared egress port	228	268µs
shared dest. host, shared ingress port	125	278µs
shared host, shared ingress and egress	221	229µs
two hosts, shared switch queue	**1,920**	**2,100µs**

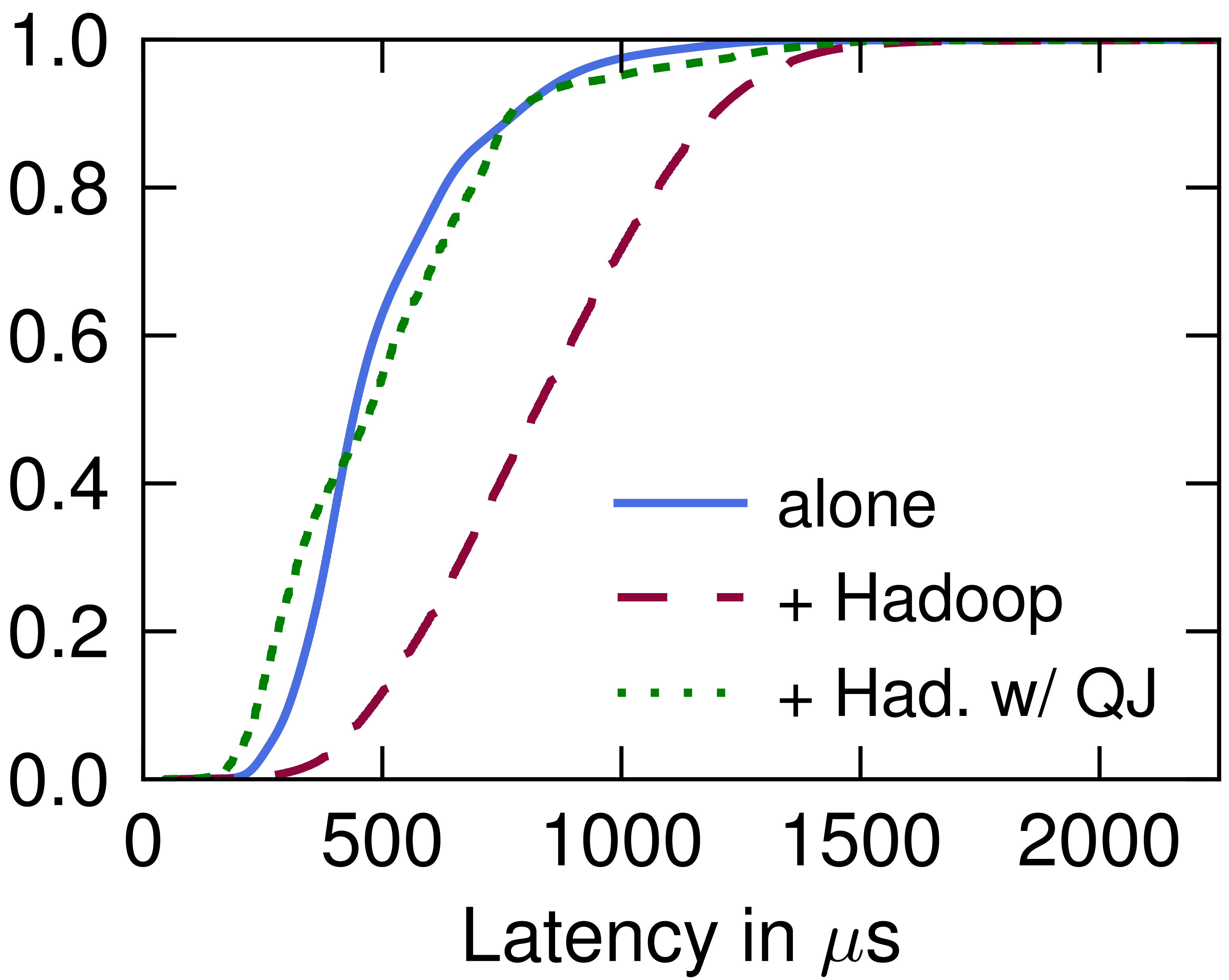
Switch Queue Interference

Ping (rpc) vs Iperf (bulk transfer)



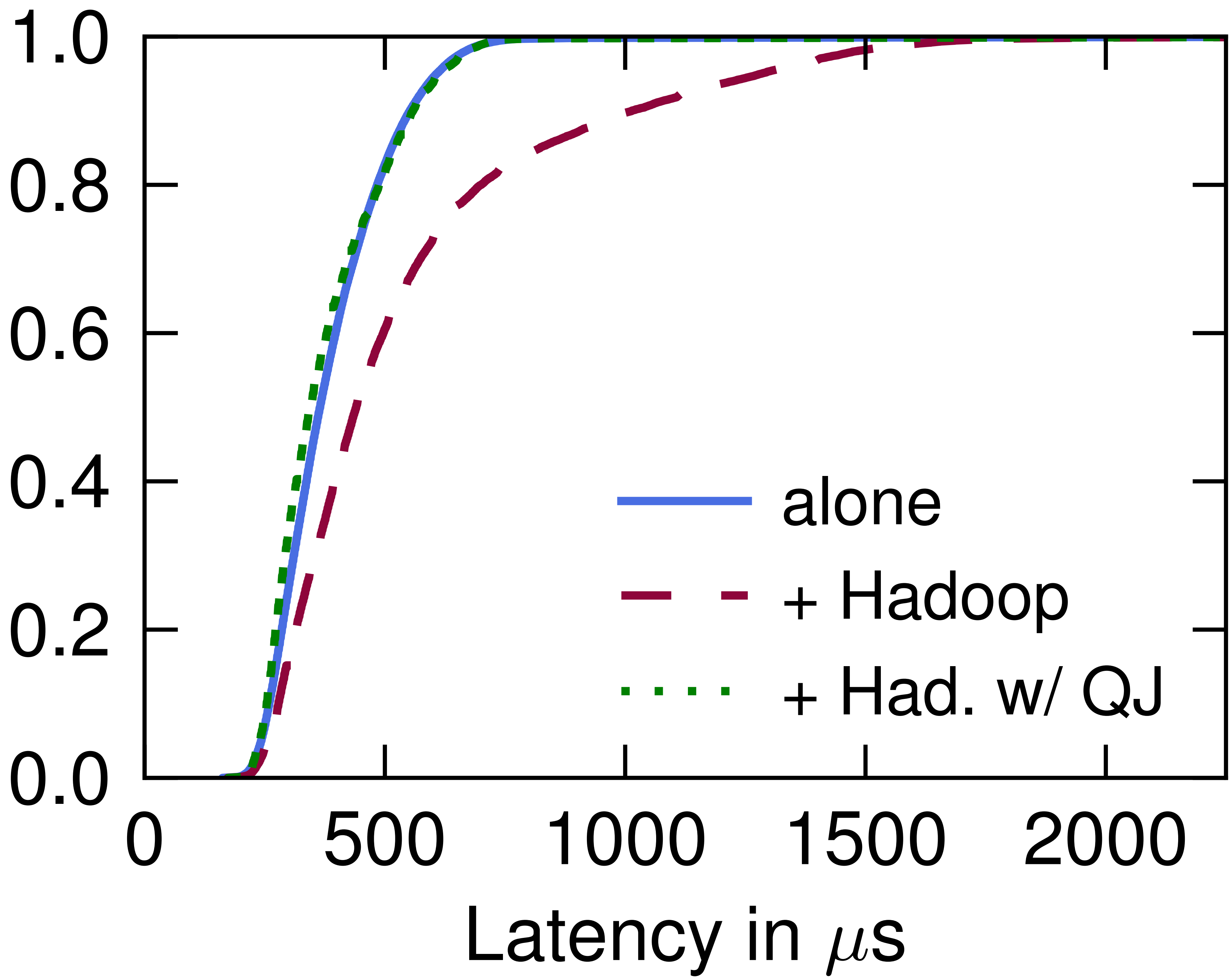
How well does it work?

memcached key-value store vs Hadoop

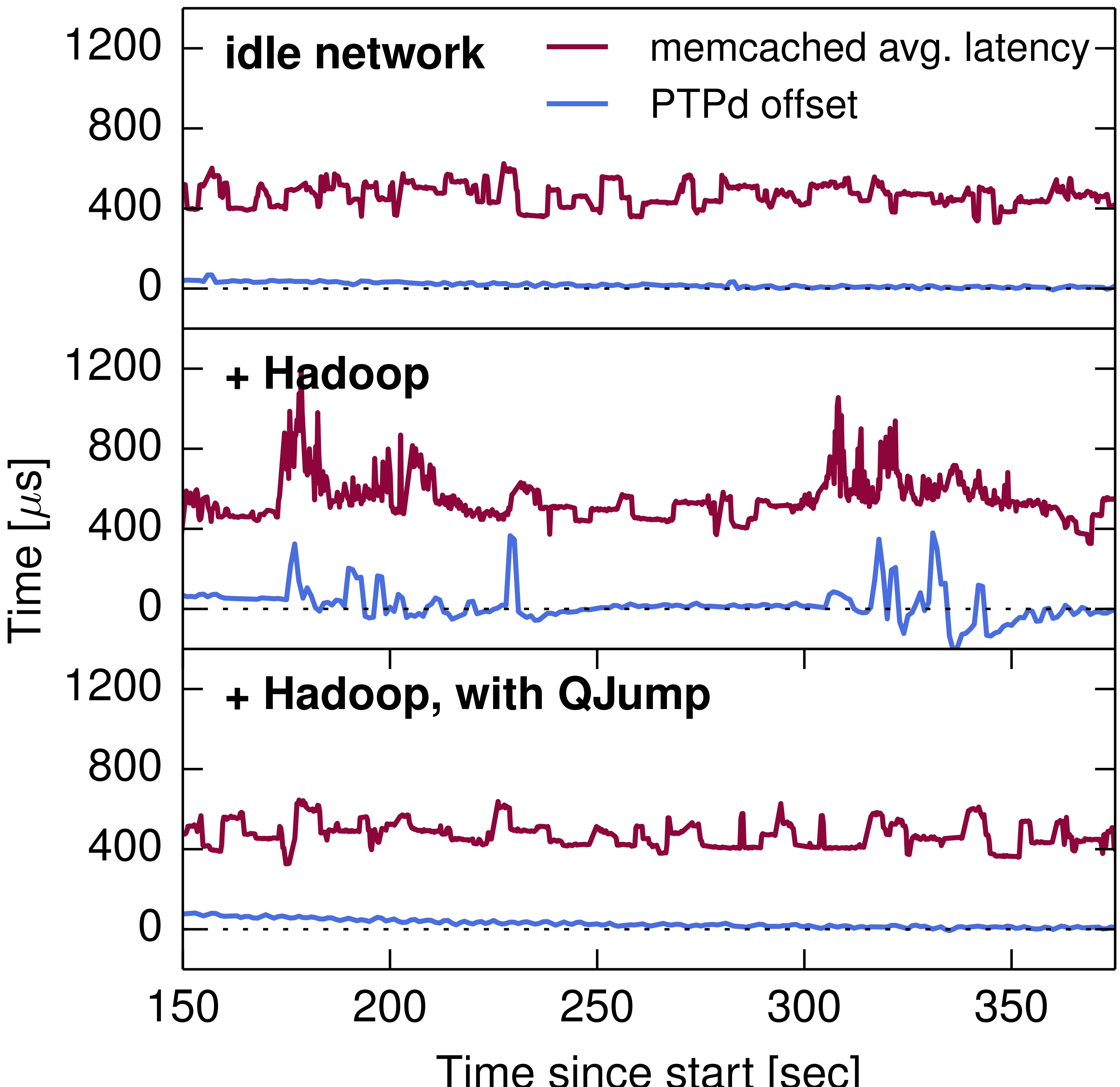


How well does it work?

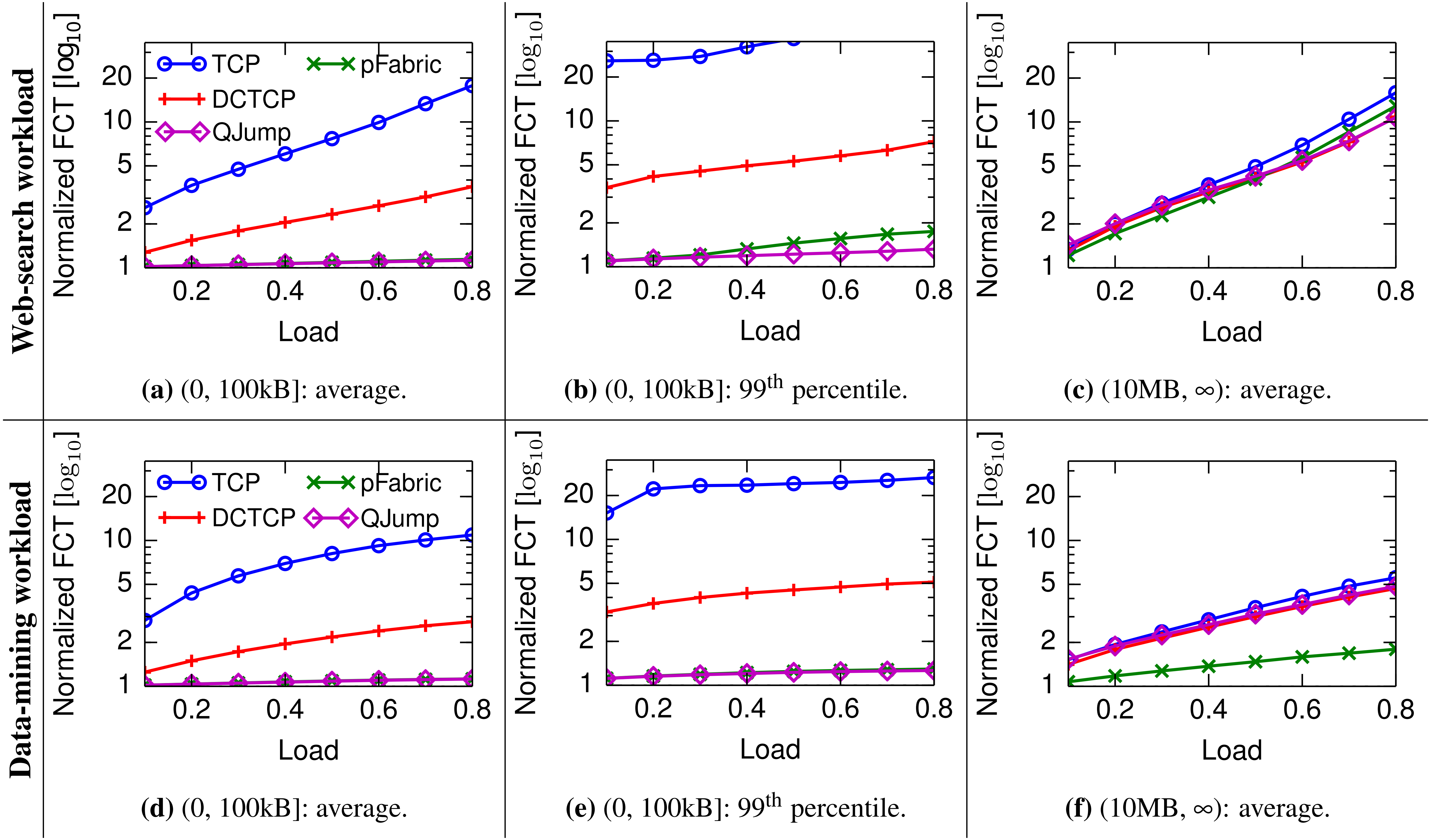
Naiad data processing framework vs Hadoop



How well does it work?



Flow Completion Times



(a) (0, 100kB]: average.
(b) (0, 100kB]: 99th percentile.
(c) (10MB, ∞): average.
(d) (0, 100kB]: average.
(e) (0, 100kB]: 99th percentile.
(f) (10MB, ∞): average.

How to calculate *f*