A Distributed Algorithm to Compute Max-Min Fair Rates Without Per-Flow State

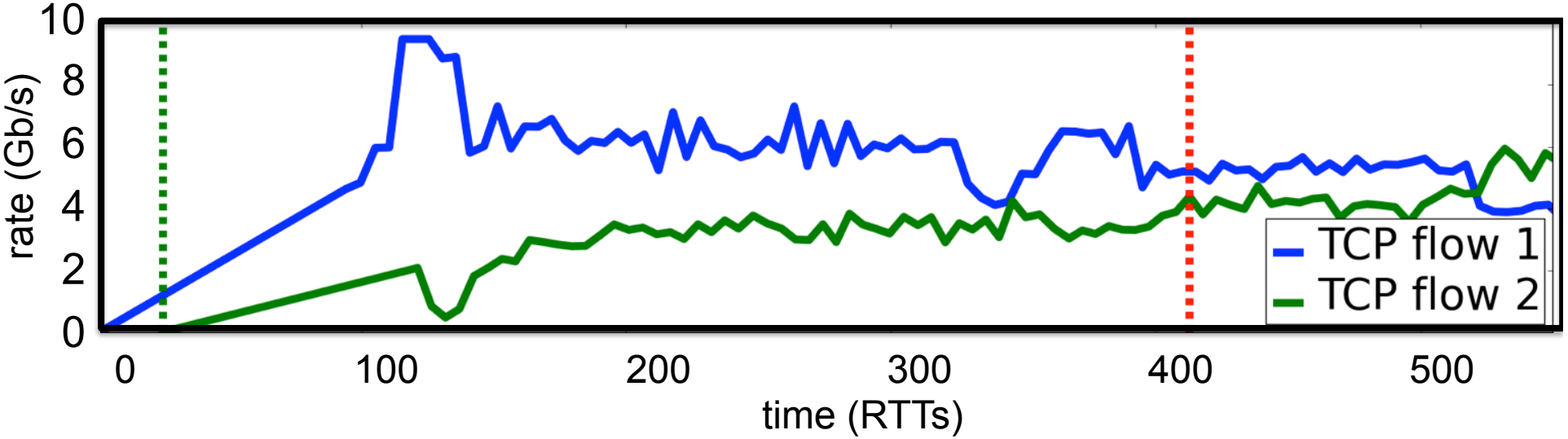
Lavanya Jose, Stephen Ibanez, Nick McKeown
Stanford University

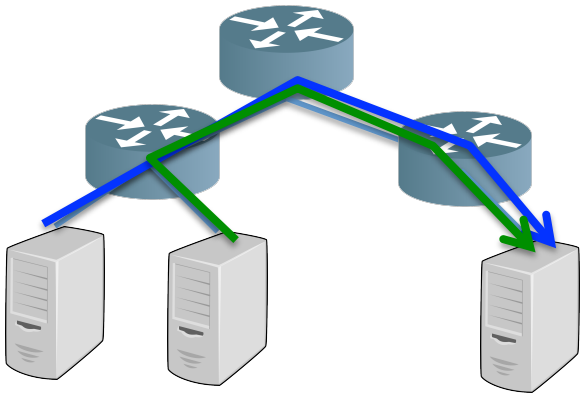
Mohammad Alizadeh
MIT CSAIL

Imagine you're a cloud provider..

- You want
  - Fair + High Throughput Rates for Long Flows
  - Low Latency for Short Flows
- Many attempts to achieve this, but they all have the same *problem*—they *react* to congestion instead of preventing it.

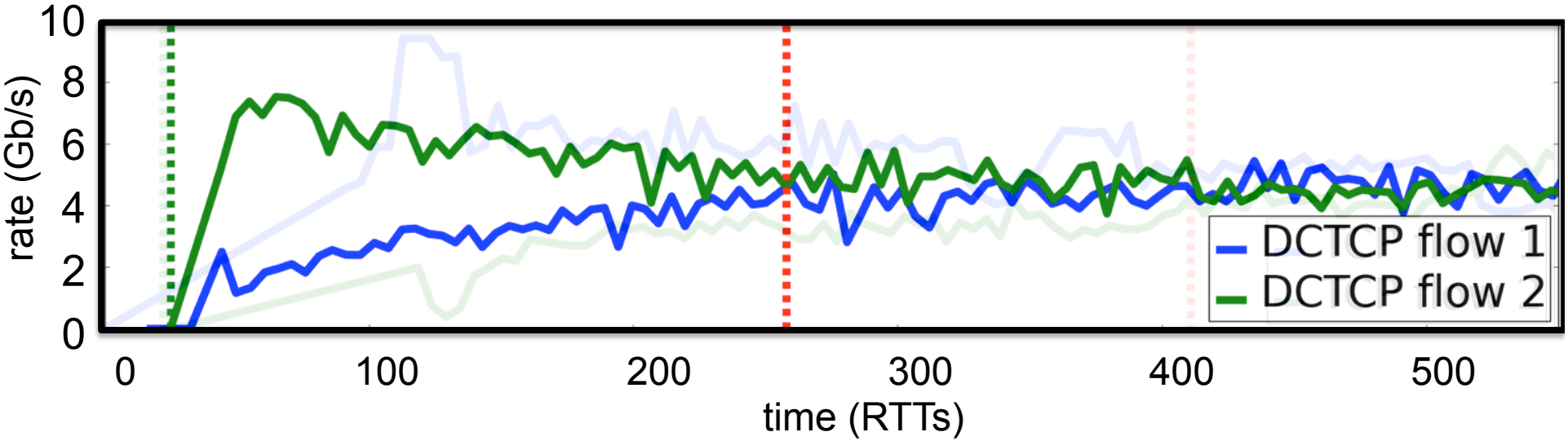
TCP takes 400 round trips for 2 flows!

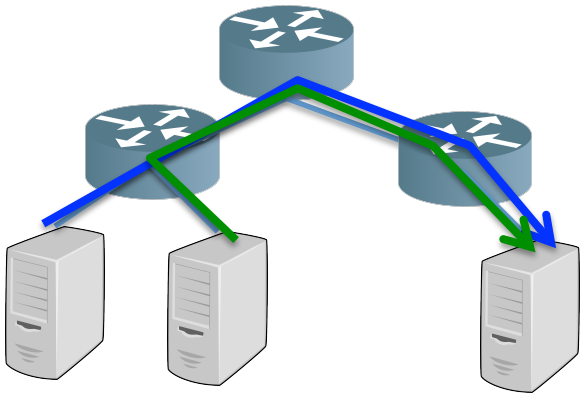




10 Gb/s links
0.2 ms RTT

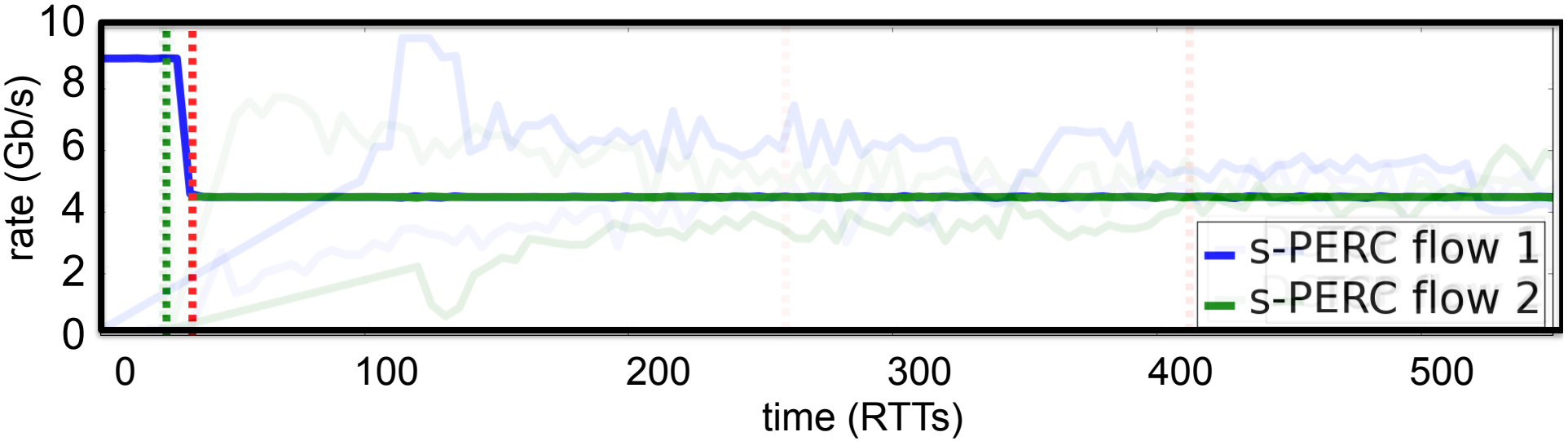
DCTCP takes 250 round trips for 2 flows!

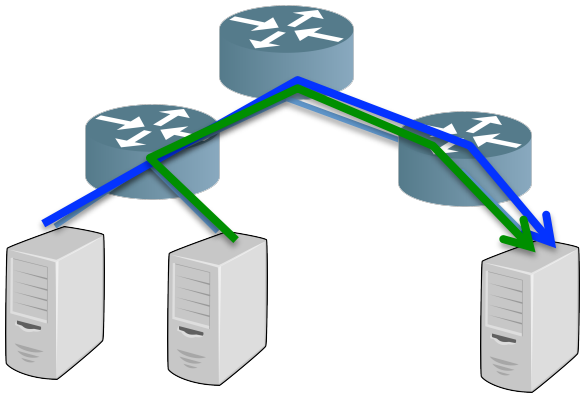




10 Gb/s links
0.2 ms RTT

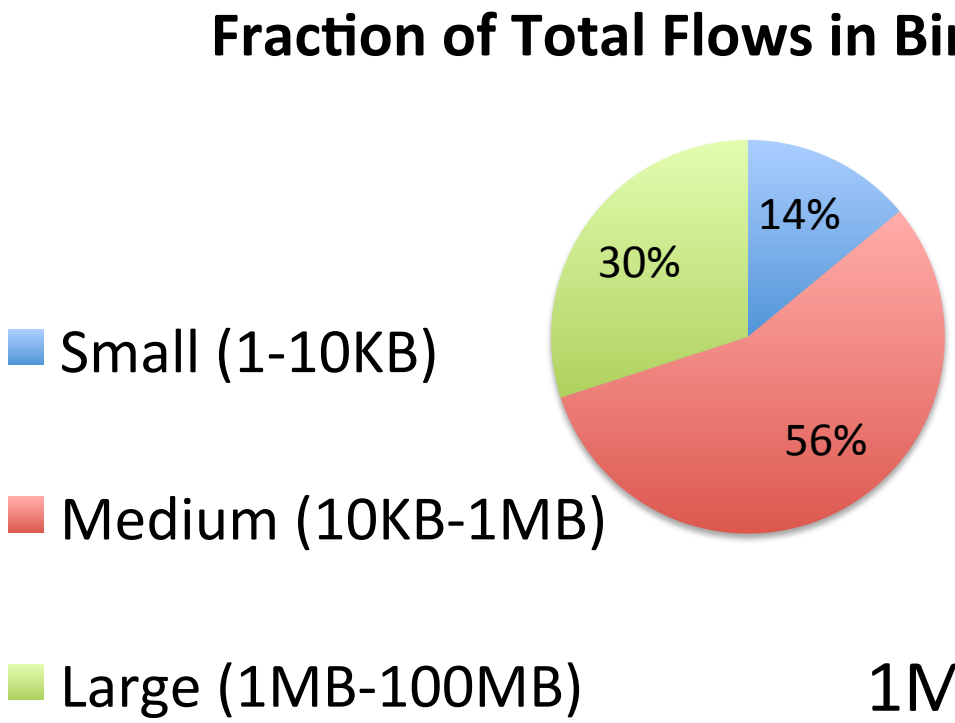
Why not ~**1 round trip** to find rates?





10 Gb/s links
0.2 ms RTT

Flows can finish much faster

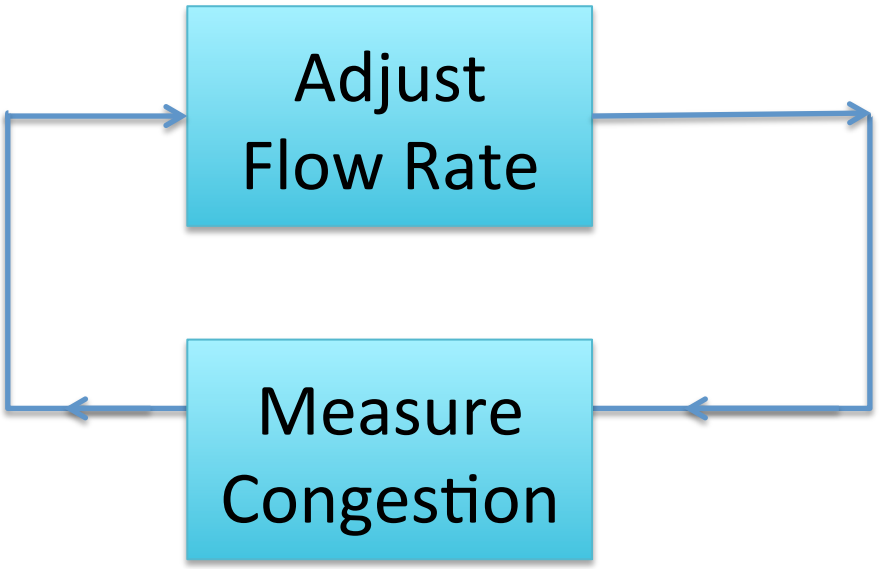


1MB / 100 Gb/s = 80 µs
= 7—8 RTTs

Most flows can finish long before congestion control finds good rates.

Why "Reactive" Algorithms Are Slow

1. No info. about routing matrix, link speeds.
2. Measure congestion and adjust flow rates.
3. Timid steps to remain stable.



What if instead of reacting to congestion, we prevent it in the first place?

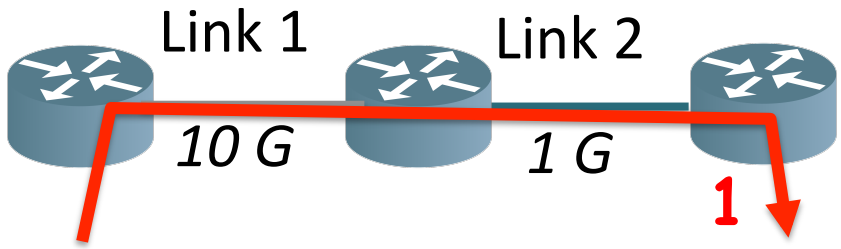
Proactive Algorithms

- Use knowledge of the routing matrix and link speeds to calculate flow rates directly.
- Centralized: can't do this at scale
- Distributed?

PERC Algorithms

Red Flow's Per-Link State	Link	
	1	2
alloc. (a)	10	1

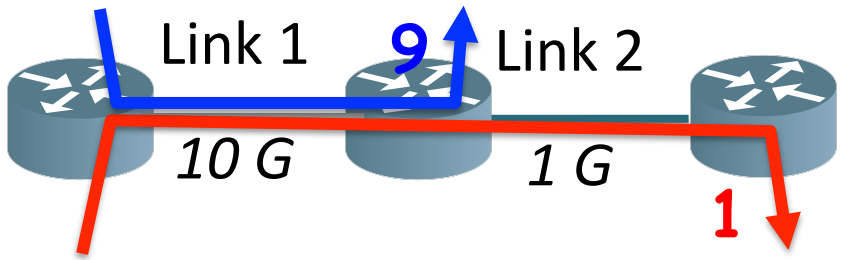
"Proactive Explicit Rate Control (PERC)"



PERC algorithms for the Cloud

- Practical algorithms should
  - update the control packet in a few nanoseconds
  - using very little switch memory, constant state.
- Relevant today: cloud providers own entire infrastructure

Max-Min Fairness



"Fair Share Rate"

C / N
= 10 G / 2
= 5 G

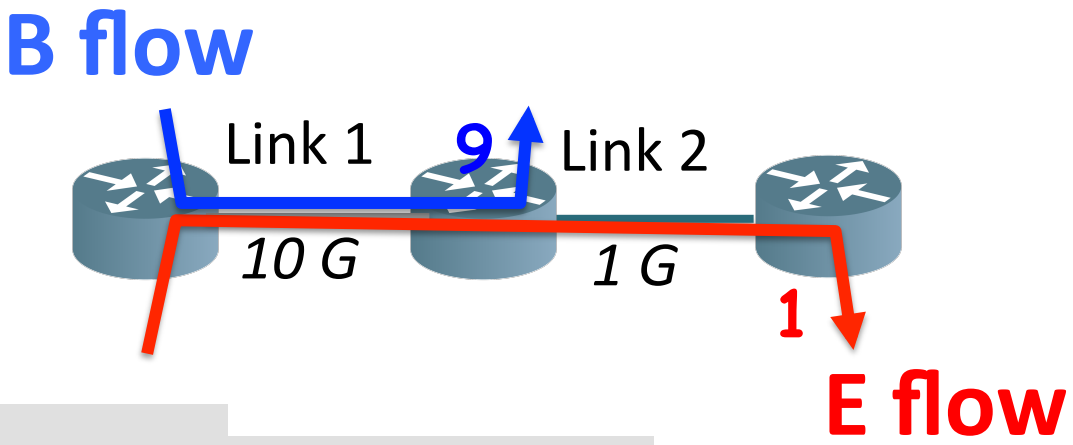
C / N
= 1 G / 1
= 1 G

Max-Min Rate

Outline

- Simple practical “naïve” PERC (n-PERC)
  - Can take arbitrarily long to converge
- Modify n-PERC using constant state to get “stateless” PERC (s-PERC)
  - Converges in known bounded time
  - Without per-flow state at switches

Two Kinds of Flows At a Link



$$\frac{(C - \text{SumE})}{\text{NumB}} = \text{Max-Min Rate of B flow}$$

The n-PERC Algorithm

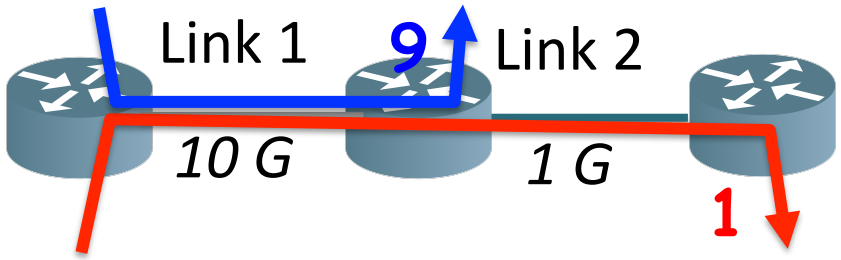
- Calculate *b*ottleneck rate assuming flow will be bottlenecked here, $b = (C - SumE)/NumB$
- Calculate limit rate, *e*, lowest bottleneck rate *e*lsewhere
- Allocate lower of the two $a = min(b,e)$

The n-PERC Algorithm

- Calculate *b*ottleneck rate, *b* …
- Calculate limit rate, *e* …
- Allocate lower of the two ..

b = (1 − 0)/1 = 1 Gb/s
e = 10 Gb/s
a = min(b, e) = 1 Gb/s

Red Flow's Per-Link State	Link	
	1	2
bn. rate (b)	10	∞



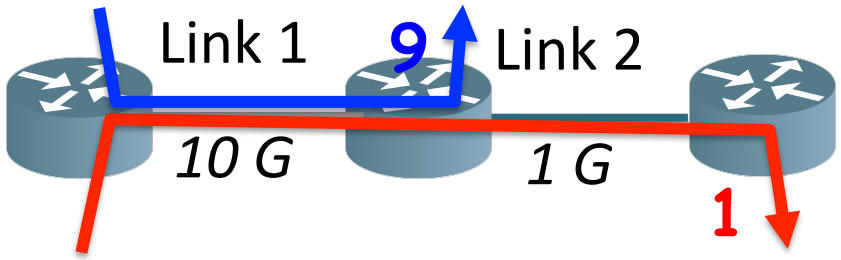
Link 2's State	
NumB	1
SumE	0

The n-PERC Algorithm

- Calculate *b*ottleneck rate, *b* …
- Calculate limit rate, *e* …
- Allocate lower of the two ..
- Update packet and link

b = (1 − 0)/1 = 1 Gb/s
e = 10 Gb/s
a = min(b, e) = 1 Gb/s
b < e: bottlenecked here

Red Flow's Per-Link State	Link	
	1	2
bn. rate (b)	10	**1**
state @ alloc. (a)	B @ 10	**B @ 1**

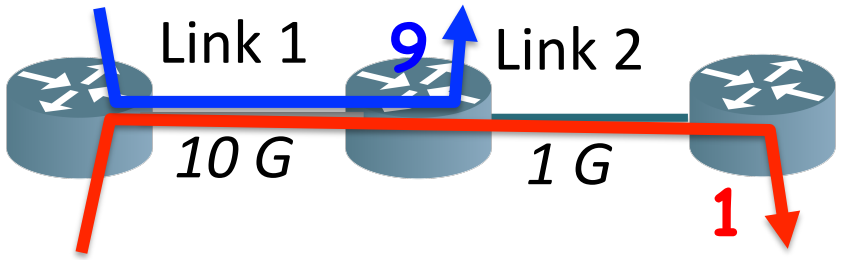


Link 2's State	
NumB	1
SumE	0

The n-PERC Algorithm

Red Flow's Per-Link State	Link	
	1	2
bn. rate (b)	10	**1**
state @ alloc. (a)	**E @ 1**	B @ 1

Flow's nk State	Link	
	1	2
b)	10	1
alloc. (a)	B @ 10	**B @ 1**

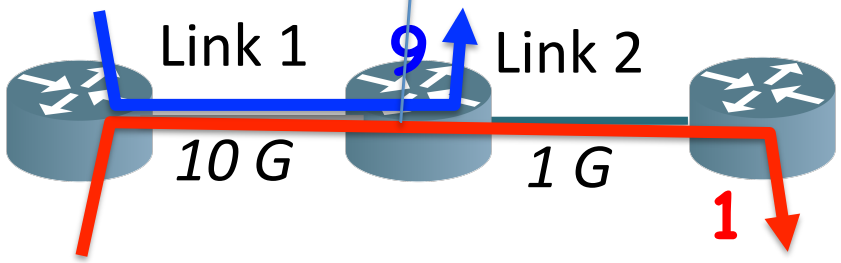


Link 2's State	
NumB	0
SumE	0

The n-PERC Algorithm

b = 1 Gb/s
e = 10 Gb/s
a = min(b, e) = 1 Gb/s
b < e: bottlenecked here.

Red Flow's Per-Link State	Link	
	1	2
bn. rate (b)	**10**	
state @ alloc. (a)	B @ 10	

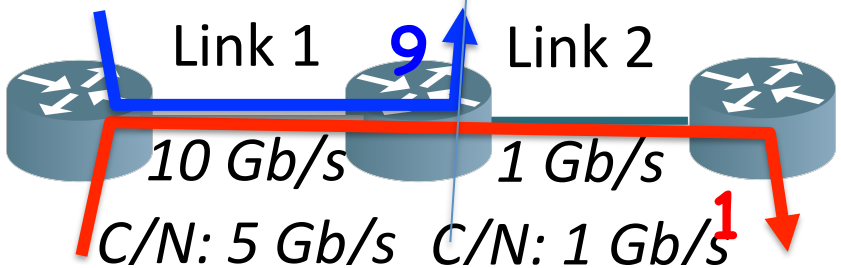


Link 2's State	
NumB	1
SumE	0

Propagating a Bad Bottleneck Rate

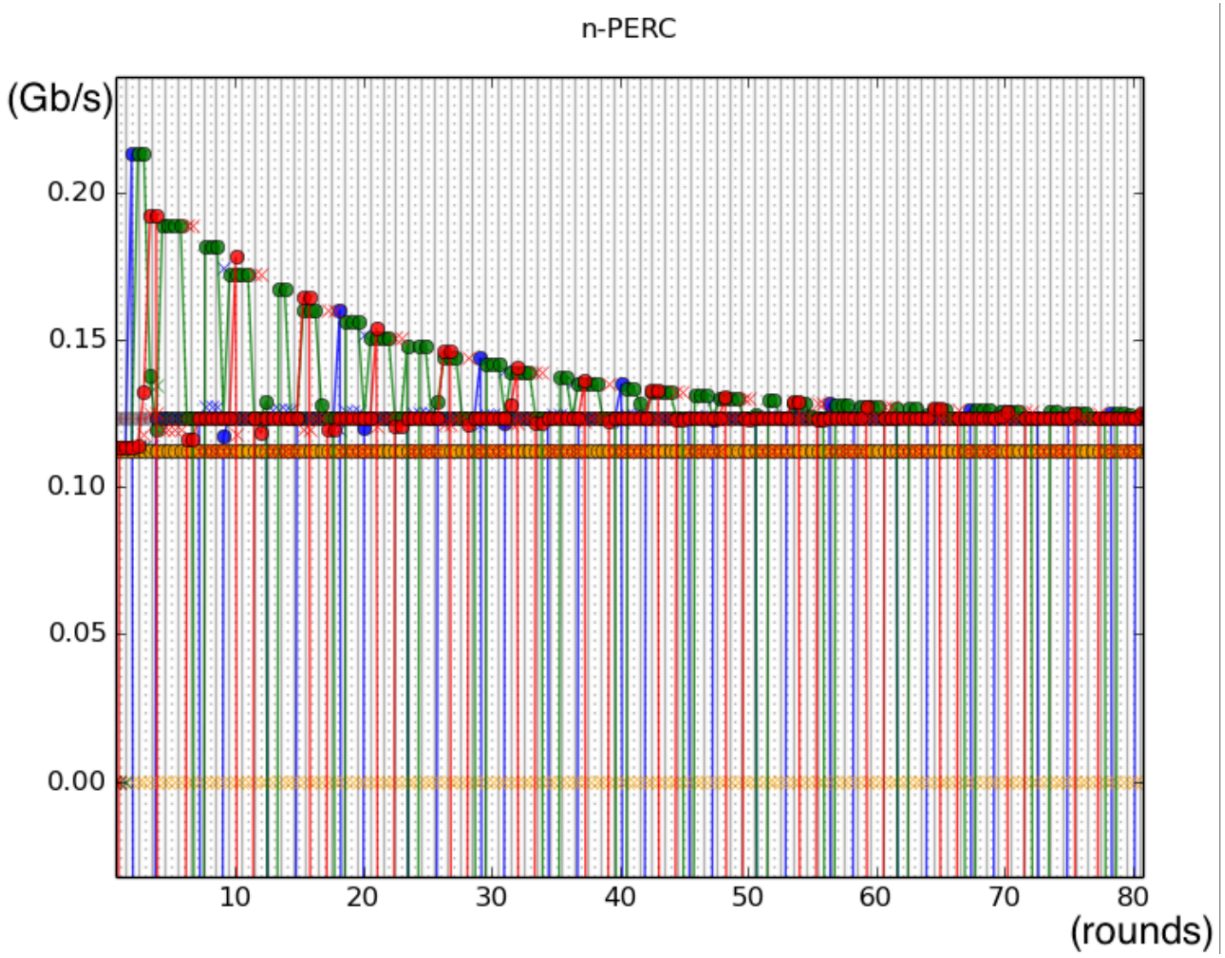
b = 1 Gb/s
e = 0.5 Gb/s
a = min(b, e) = 0.5 Gb/s
e < b: bottlenecked elsewhere at 0.5 Gb/s!?

Red Flow's Per-Link State	Link	
	1	2
bn. rate (b)	0.5	



Link 2's State	
NumB	1
SumE	0

Bad bottleneck rates delay convergence



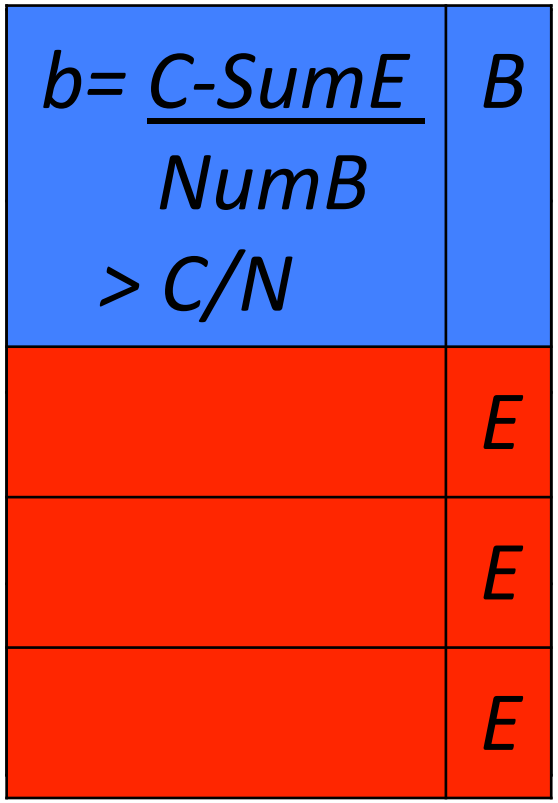
Key Insight

- Recognize when the bottleneck rate is safe to propagate, using constant state.
- *MaxE*, maximum rate allocated to E flow in the last two RTTs.
- Propagate *b* only when *b >= MaxE.*
- Propagated rate at least fair share rate.
- Link with lowest fair share recognizes *B* flows.

The **s**-PERC Algorithm

- Calculate *b*ottleneck rate assuming flow will be bottlenecked here, $b = (C - SumE)/NumB$
- Calculate limit rate, *e*, lowest **propagated** bottleneck rate elsewhere
- Allocate lower of the two *a = min(b,e)*
- Update packet and link
- Decide whether to propagate b (b >= MaxE?)

Link propagates at least its Fair Share Rate.



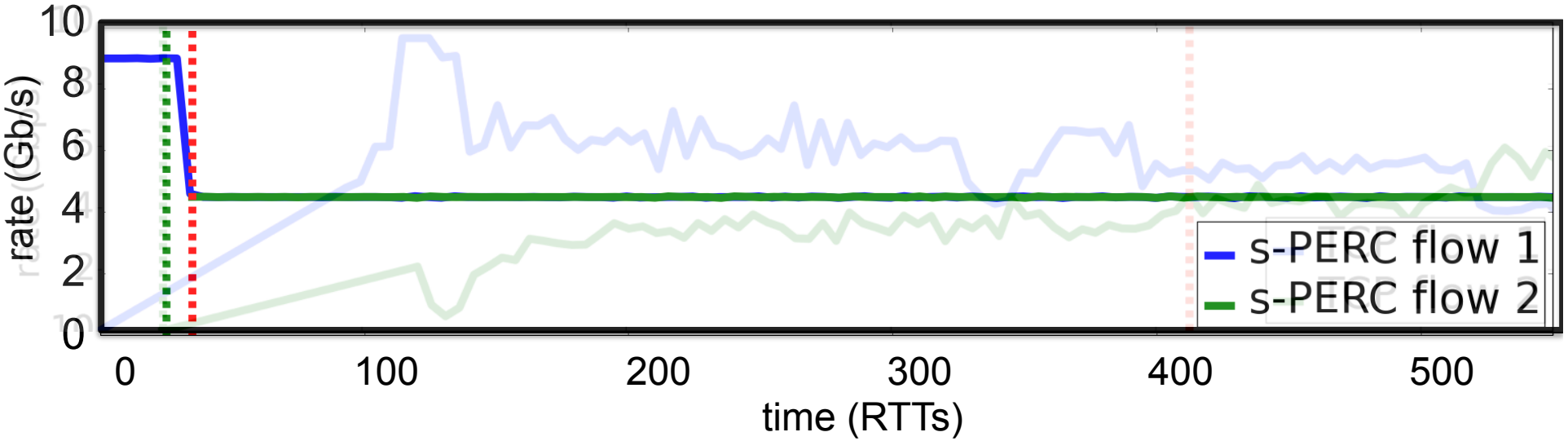
propagate
b > MaxE.

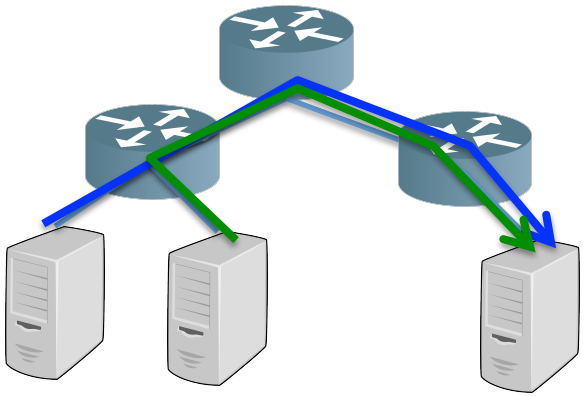
MaxE < C/N

Convergence of s-PERC

s-PERC converges in 6N RTTs, where N is the number of steps that k=2 Waterfilling takes for the same routing matrix and link speeds.

s-PERC is practical
4x10 Gb/s NetFPGA SUME prototype





10 Gb/s links
0.2 ms RTT

Fairness + Throughput for Long Flows, Low Latency for Short Flows

Convergence Times (Fairness)

- 10x faster than RCP in DC networks
- 1.3-6x faster than RCP in WAN networks

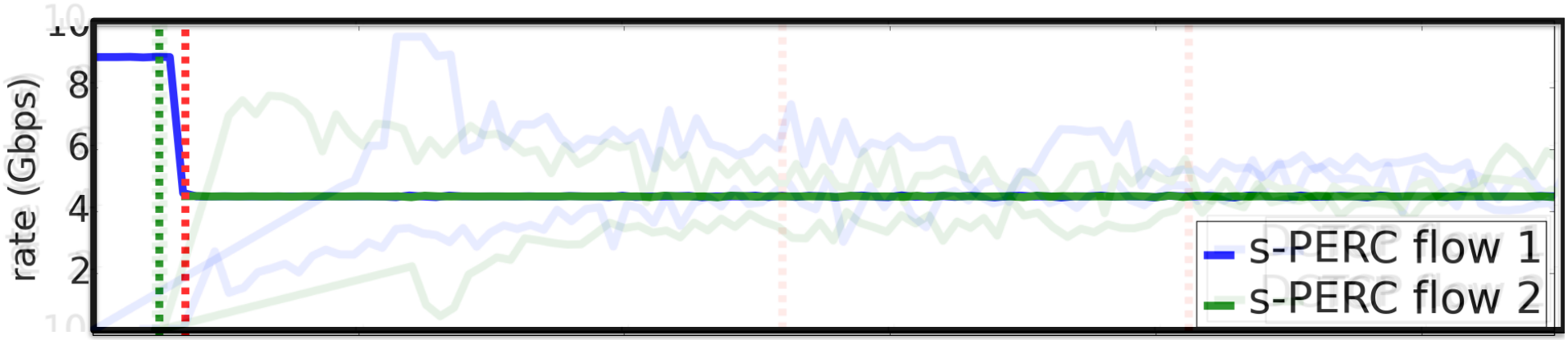
Flow Completion Times (Throughput for Long Flows, Latency for Short Flows)

- Close to Ideal Max-Min For Long Flows
- Close to Minimum For Short Flows

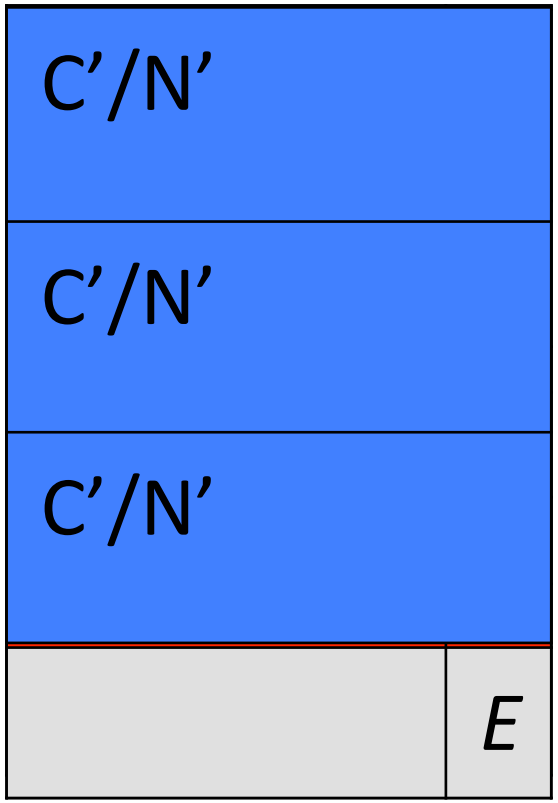
Conclusion

- Reactive algorithms converge slowly relative to flow durations in fast networks.
- Proactive PERC algorithms, use information about the network and flows to compute rates directly, they are much faster.
- PERC algorithms are promising and relevant today and s-PERC, a distributed PERC algorithm for max-min fair rates without per-flow state, is only a first step.

Thank you!



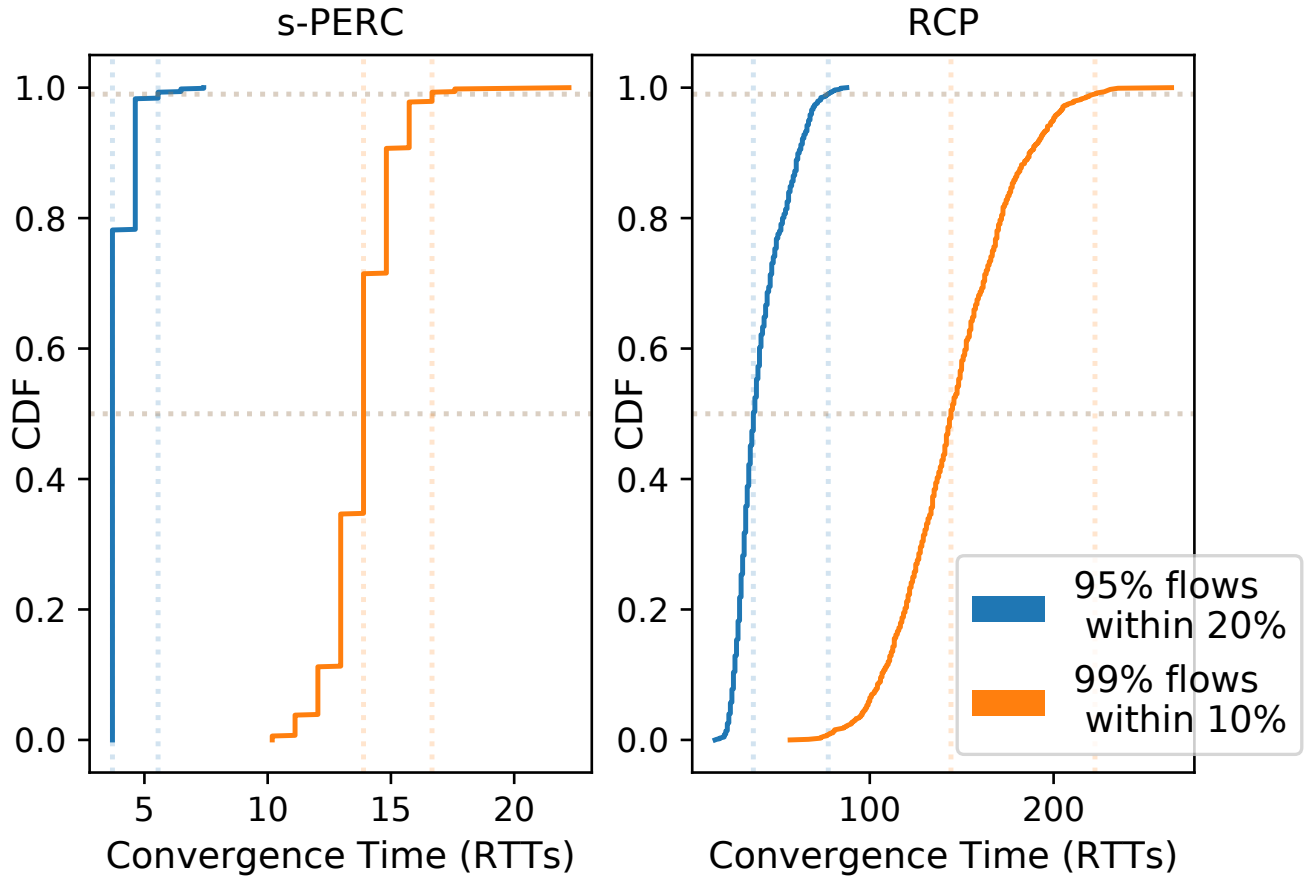
The bottleneck rate propagated by a link is at least its Fair Share Rate.



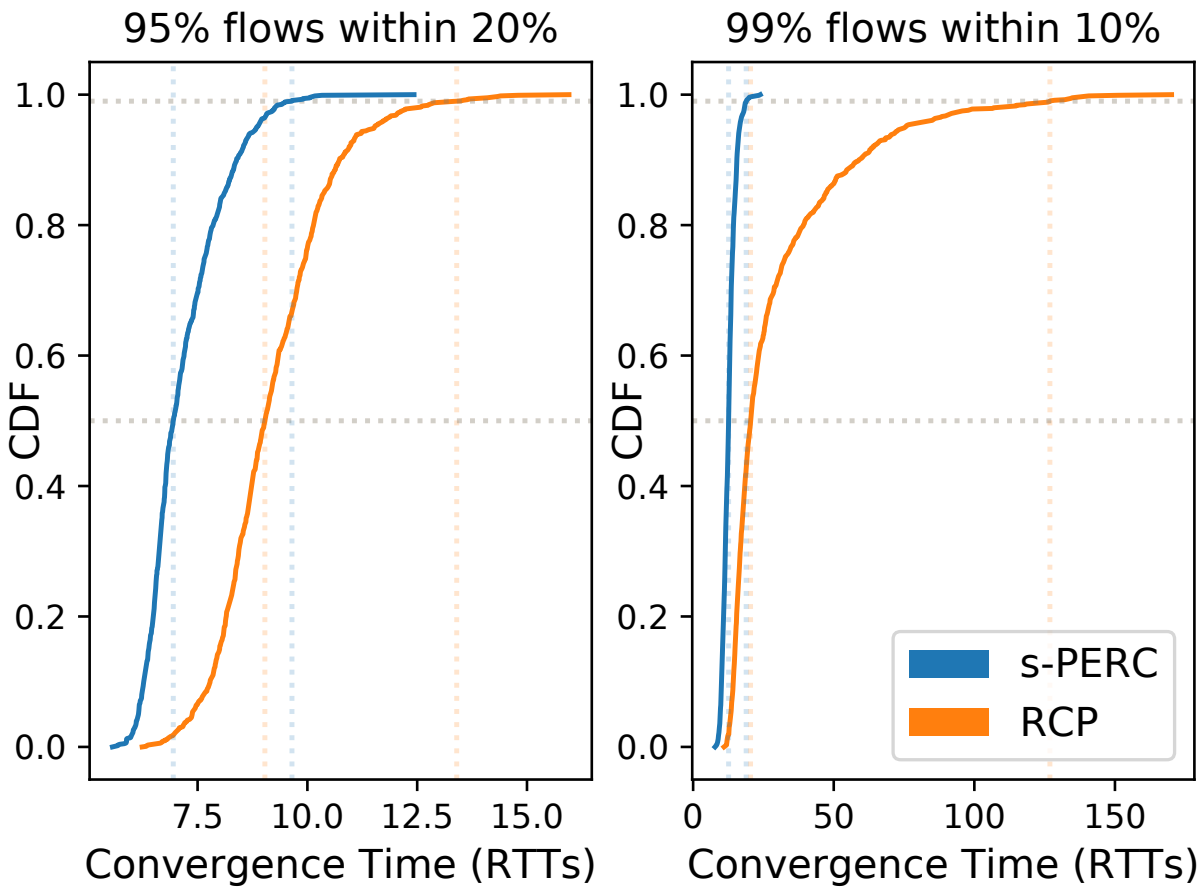
propagate
b > MaxE.

MaxE < C/N

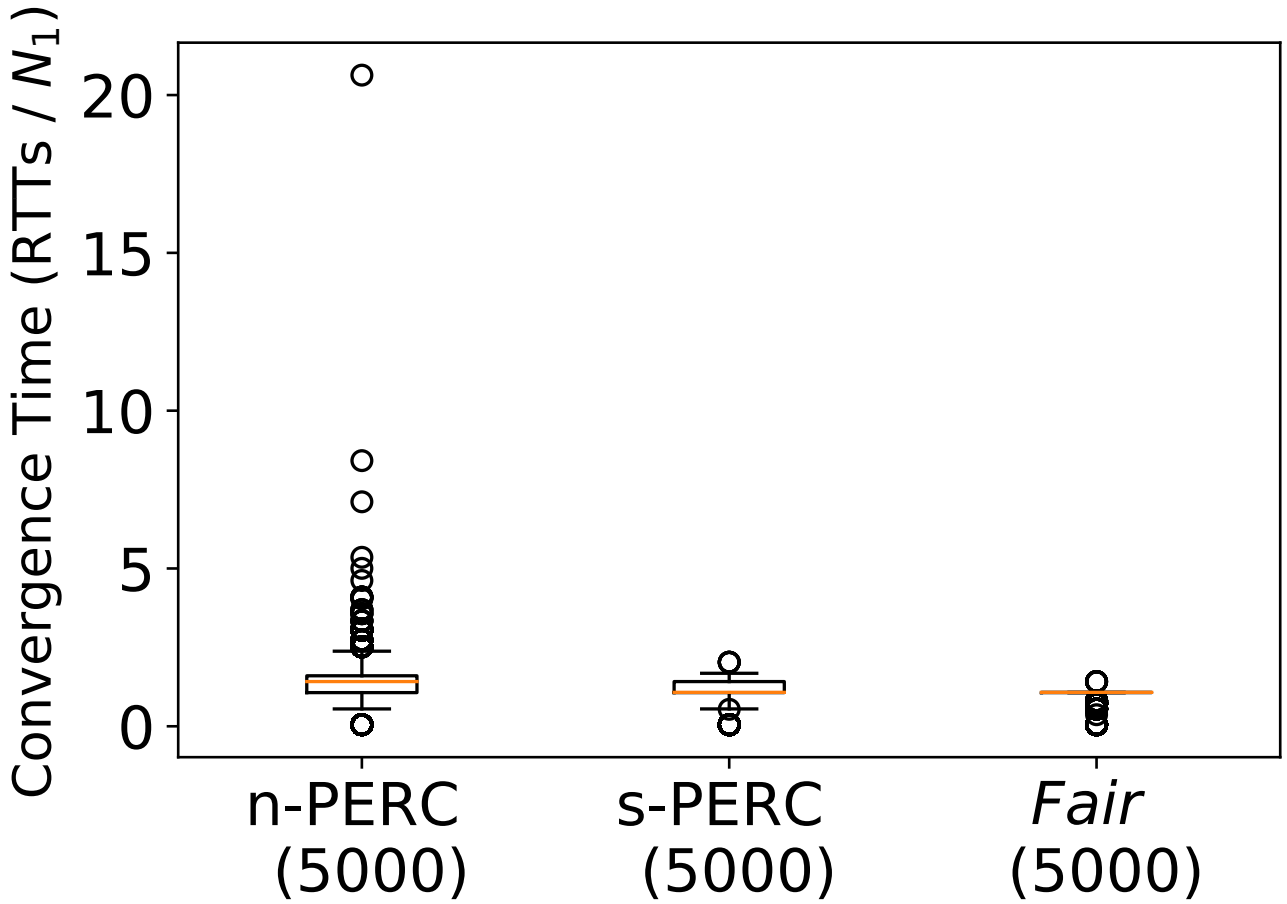
Convergence Time (DC, 100G)



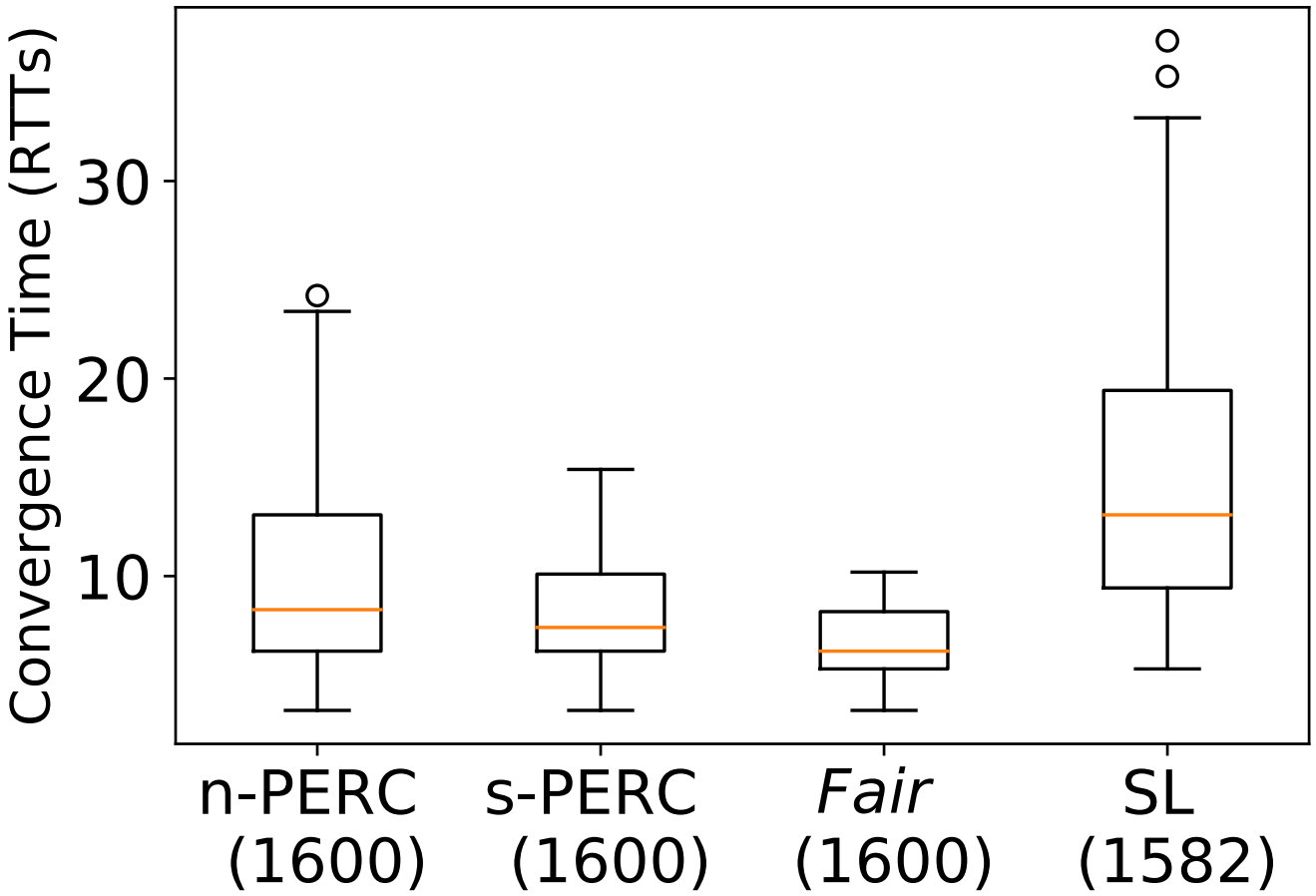
Convergence Times (WAN, 10G)



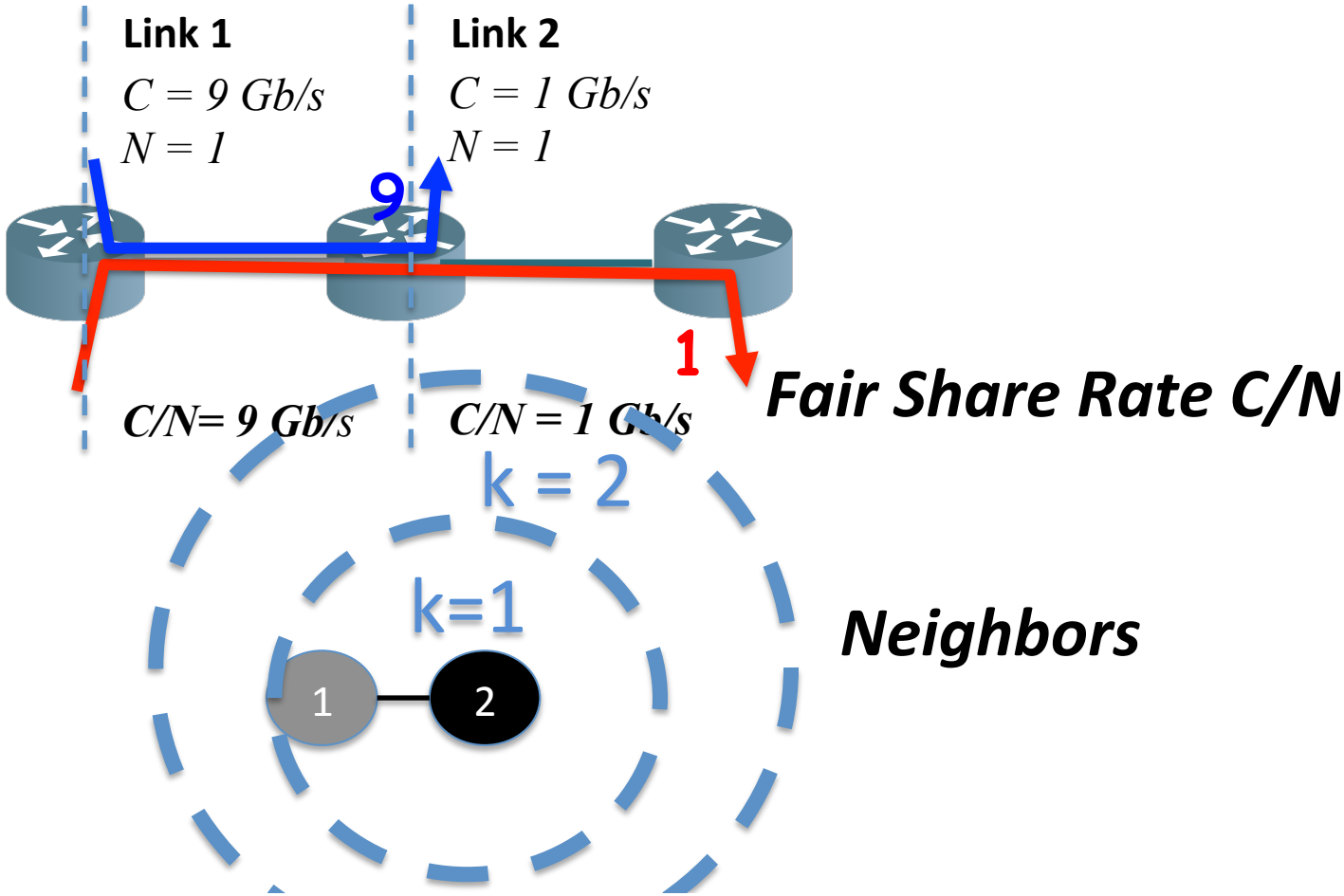
Convergence Times (PERC, worst-case)



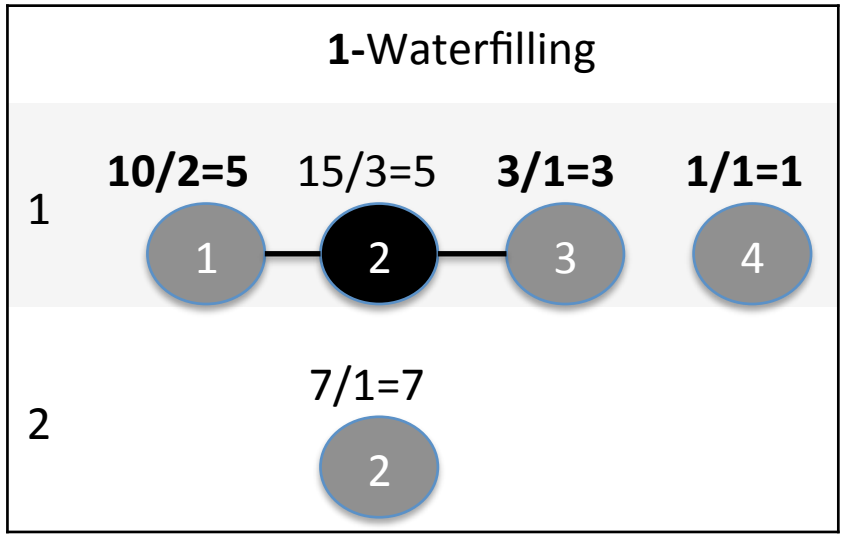
Convergence Times (PERC, common)

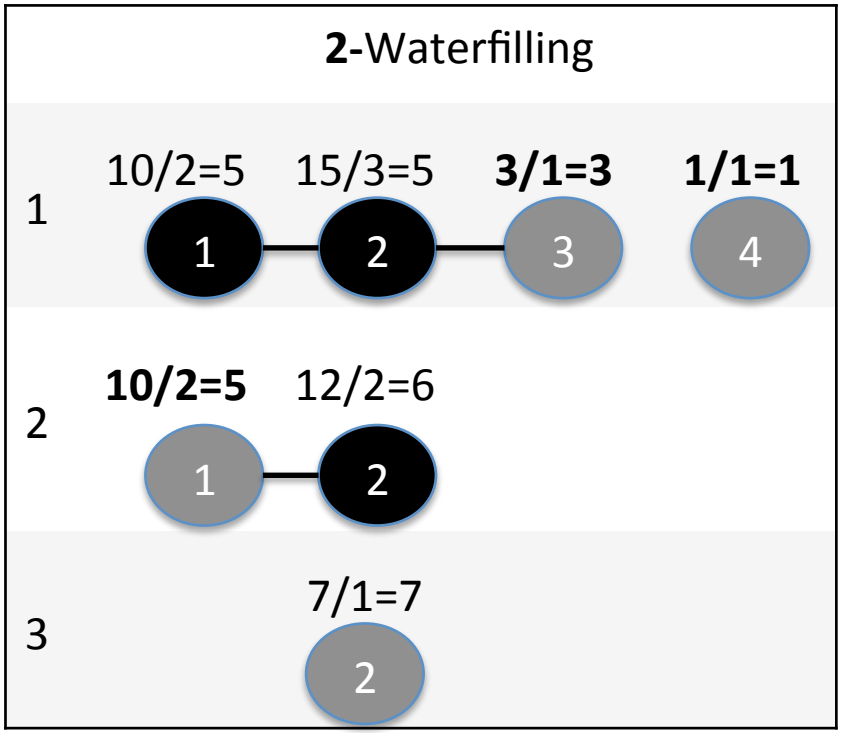


k-Waterfilling: Recursive Algorithms To Compute Max-Min Rates

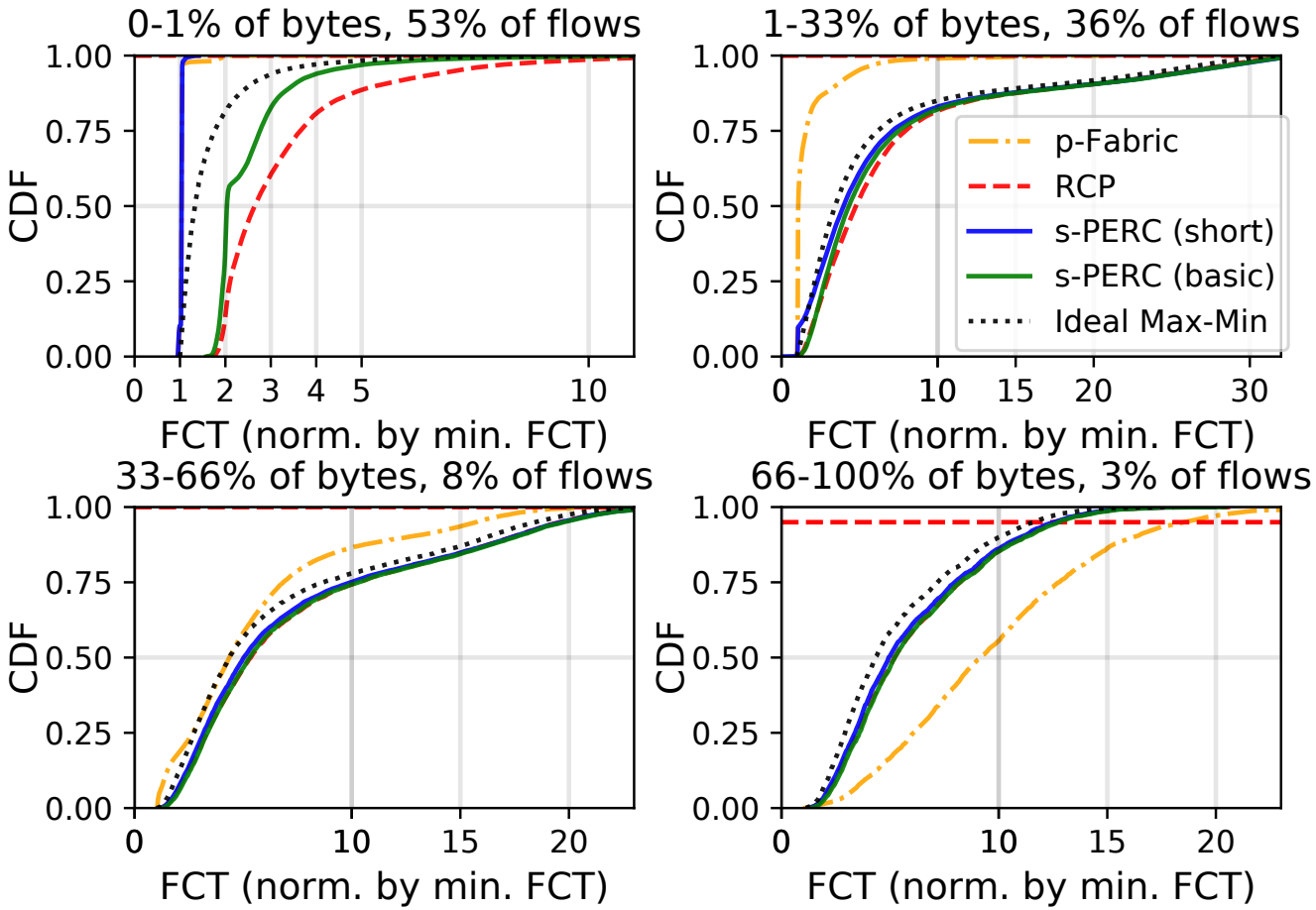


k-Waterfilling (k=1, 2)

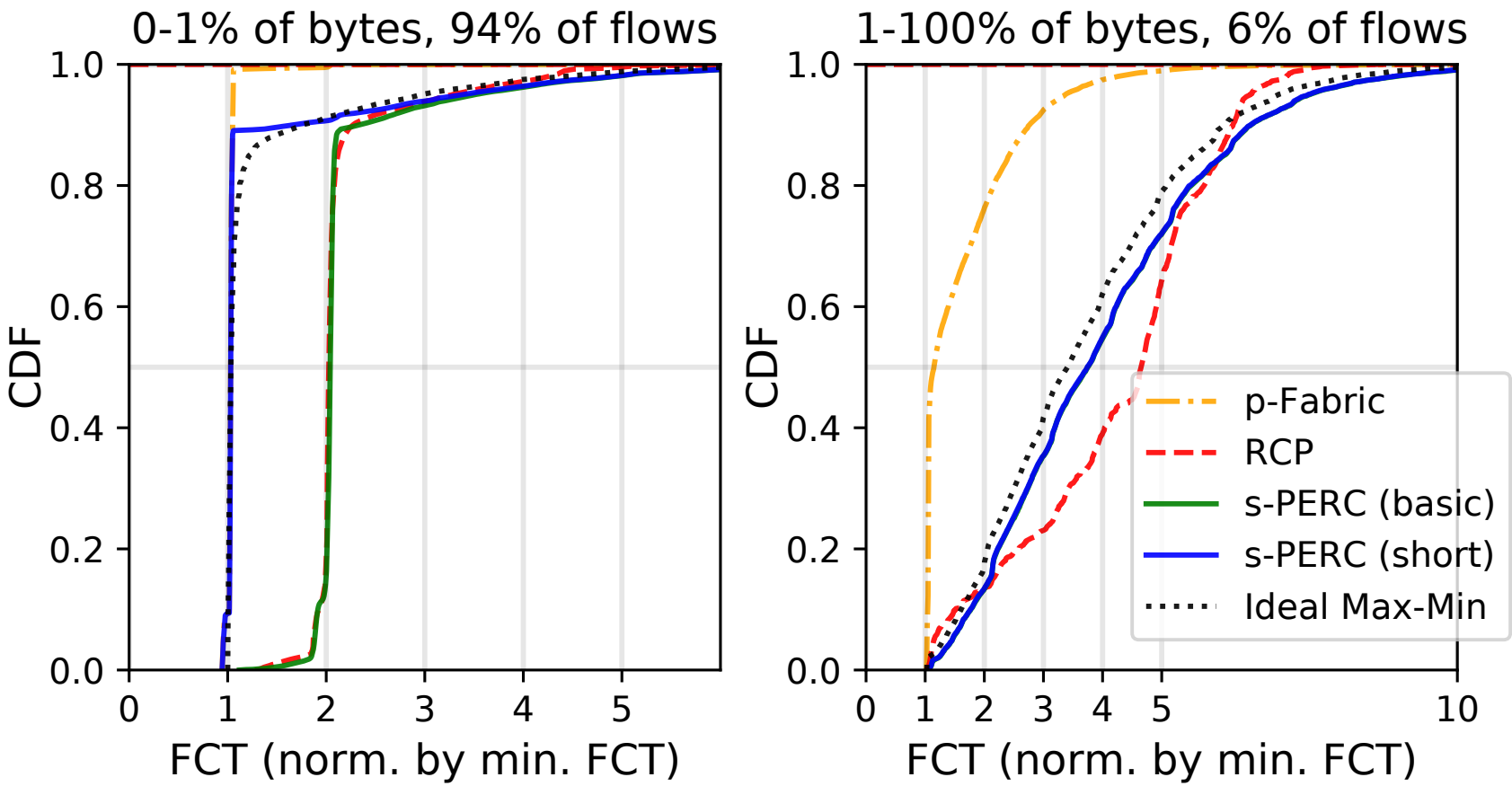




FCT (DC, 100G, search, 60%)



FCT (DC, data-mining, 60%)



Testbed, 2-level experiment

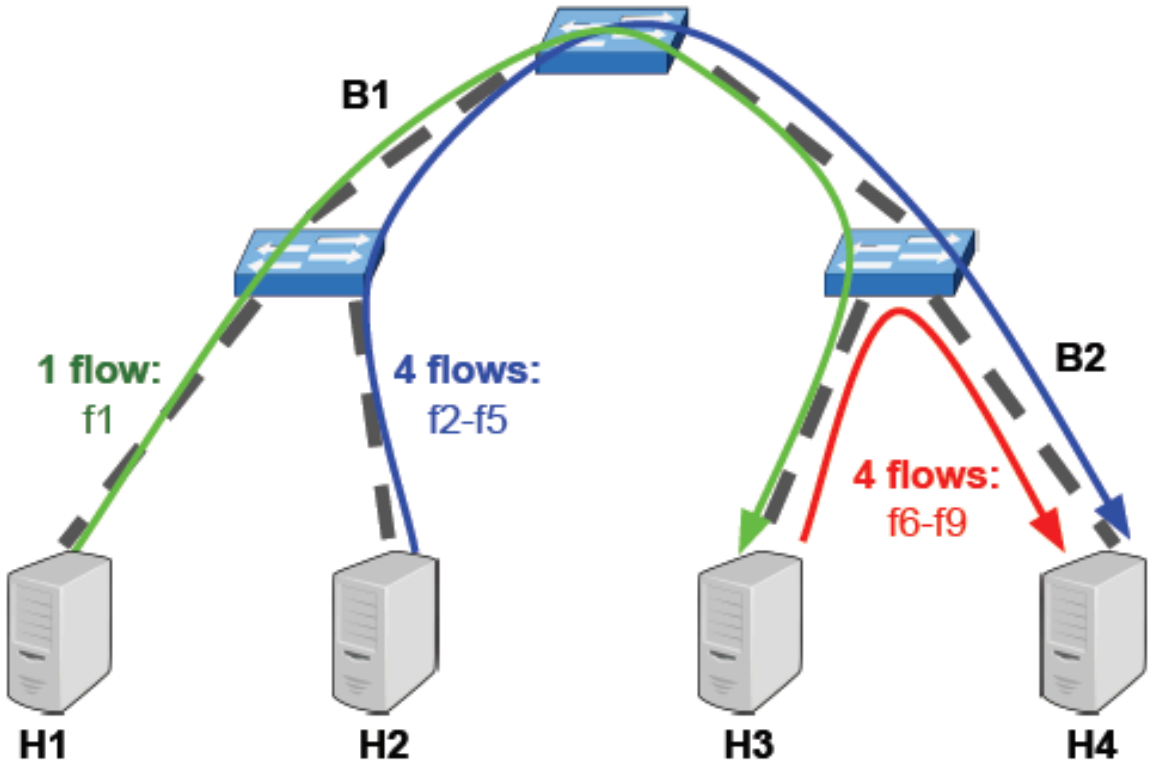


Fig. 11. The topology and traffic pattern used for the two-level dependency chain experiment.

Why 6 RTTs per level?

Time	Updates of flow f at link k	Updates of flows f, g at link l (1st-deg.)	Updates of flow g at link j (2nd-deg.)
T		$\mathbf{b} \geq \mathbf{c}_k/\mathbf{n}_k$ or $\mathbf{i} = 1$	$\mathbf{b} \geq \mathbf{c}_k/\mathbf{n}_k$ or $\mathbf{i} = 1$
$T + 1 \cdot round$	$\mathbf{e} \geq \mathbf{c}_k/\mathbf{n}_k$	$\mathbf{e} \geq \mathbf{c}_k/\mathbf{n}_k$	
$T + 2 \cdot round$			
$T + 3 \cdot round$	$\mathbf{b} = \mathbf{c}_k/\mathbf{n}_k$ $\mathbf{s} = \mathrm{B}$ $\mathbf{x} = \mathbf{b} = \mathbf{c}_k/\mathbf{n}_k$	$\mathbf{b} > \mathbf{c}_k/\mathbf{n}_k$	
$T + 4 \cdot round$	$\mathbf{i} = 0$		
$T + 5 \cdot round$		$\mathbf{e} = \mathbf{c}_k/\mathbf{n}_k$ $\mathbf{s} = \mathrm{E}$ $\mathbf{x} = \mathbf{e} = \mathbf{c}_k/\mathbf{n}_k$ (for flow f)	

$$\frac{c_k}{n_k} < \frac{c_j}{n_j} < \frac{c_l}{n_l}$$

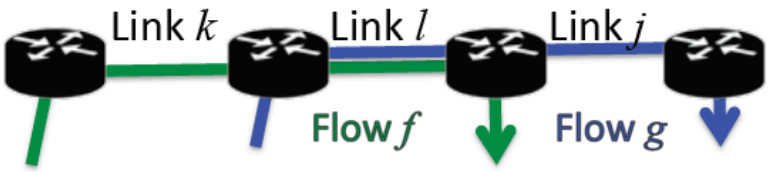


Fig. 5. Links k and j share flows f and g respectively with link l.

Maintaining MaxE with constant state.

- Timeout every RTT (e.g., 10^5 cycles)
- Initially,
  - MaxE' = 0 ("maximum E alloc. in this RTT")
  - MaxE = 0 ("maximum E alloc. in last two RTTs")
- On timeout:
  - MaxE reset to MaxE' (so it doesn't increase forever)
  - MaxE' reset to 0 (reflects max. E since last timeout)