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

LAVANYA JOSE, Stanford University, USA
STEPHEN IBANEZ, Stanford University, USA
MOHAMMAD ALIZADEH, MIT CSAIL, USA
NICK MCKEOWN, Stanford University, USA

Most congestion control algorithms, like TCP, rely on a reactive control system that detects congestion, then marches carefully towards a desired operating point (e.g. by modifying the window size or adjusting a rate). In an effort to balance stability and convergence speed, they often take hundreds of RTTs to converge; an increasing problem as networks get faster, with less time to react.

This paper is about an alternative class of congestion control algorithms based on proactive-scheduling: switches and NICs "pro-actively" exchange control messages to run a *distributed* algorithm to pick "explicit rates" for each flow. We call these Proactive Explicit Rate Control (PERC) algorithms. They take as input the routing matrix and link speeds, but not a congestion signal. By exploiting information such as the number of flows at a link, they can converge an order of magnitude faster than reactive algorithms.

Our main contributions are (1) s-PERC ("stateless" PERC), a new practical distributed PERC algorithm without per-flow state at the switches, and (2) a proof that s-PERC computes exact max-min fair rates in a known bounded time, the first such algorithm to do so without per-flow state. To analyze s-PERC, we introduce a parallel variant of standard waterfilling, 2-Waterfilling. We prove that s-PERC converges to max-min fair in $6N$ rounds, where N is the number of iterations 2-Waterfilling takes for the same routing matrix.

We describe how to make s-PERC practical and robust to deploy in real networks. We confirm using realistic simulations and an FPGA hardware testbed that s-PERC converges 10-100x faster than reactive algorithms like TCP, DCTCP and RCP in data-center networks and 1.3–6x faster in wide-area networks (WANs). Long flows complete in close to the ideal time, while short-lived flows are prioritized, making it appropriate for data-centers and WANs.

CCS Concepts: • **Networks → Transport protocols**; **In-network processing**; *Network resources allocation*; *Cloud computing*; *Data center networks*;

Keywords: congestion control; max-min fairness; data center networks

ACM Reference Format:
Lavanya Jose, Stephen Ibanez, Mohammad Alizadeh, and Nick McKeown. 2019. A Distributed Algorithm to Calculate Max-Min Fair Rates Without Per-Flow State. In *Proc. ACM Meas. Anal. Comput. Syst.*, Vol. 3, 2, Article 21 (June 2019). ACM, New York, NY. 42 pages. https://doi.org/10.1145/3326135

Authors' addresses: Lavanya Jose (contact author), 353 Serra Mall, Room 314, Stanford, CA 94305, USA; Stephen Ibanez, 353 Serra Mall, Room 314, Stanford University, Stanford, CA 94305, USA; Nick McKeown, 353 Serra Mall, Room 344, Stanford University, Stanford, CA 94305, USA; Mohammad Alizadeh, 32 Vassar Street, 32-G920, MIT CSAIL, Cambridge, MA 02139, USA.

2476-1249/2019/6-ART21 $15.00
https://doi.org/10.1145/3326135

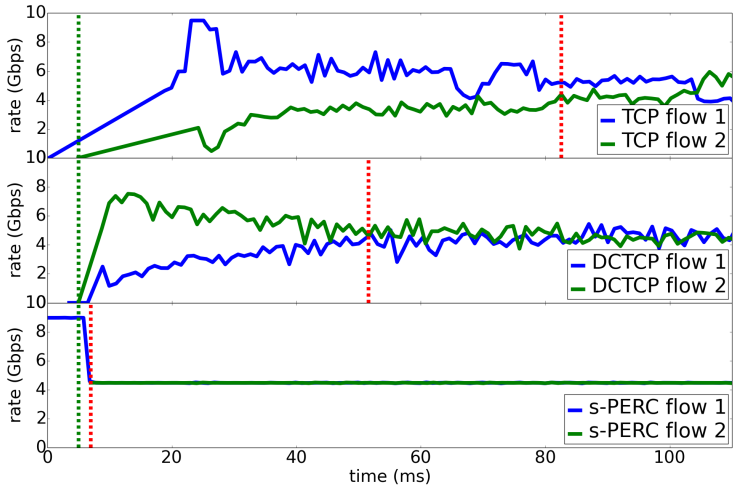


Fig. 1. Typical convergence behavior for TCP, DCTCP, and s-PERC running on our NetFPGA testbed. Two flows share a 10Gb/s link. The round-trip time (RTT) is 1ms.

1 INTRODUCTION

Cloud data-centers today host thousands of applications on networks interconnecting hundreds of thousands of servers. A congestion control algorithm has to balance the needs of short flows, which need low latency, and long flows, which need high throughput, regardless of other applications sharing the network.

Cloud providers typically use reactive congestion-control algorithms inside and between their data-centers. This class of algorithms, best represented by TCP, *reacts* to congestion signals, and can take hundreds of RTTs to converge, even for simple topologies. Figure 1 illustrates the problem in our 10Gb/s hardware testbed. Two servers send a TCP flow over a bottleneck link. After the second flow is added (first vertical line), it takes TCP 400 RTTs to converge to a fair allocation (second vertical line). Even DCTCP takes 250 RTTs.

In real networks, many flows are short-lived and the set of flows changes every millisecond, if not more often, suggesting that instantaneous flow rates in today's networks never have time to converge and are far from optimal. As link speeds increase this problem becomes worse, since flows can finish even faster. For example, at 100Gb/s a typical 1MB flow from a data-center search workload (used in §8.2.2) can finish in 80μs, just a few RTTs; hardly enough time for an algorithm to react.

An alternative would be to use scheduling algorithms like WFQ [16] or PGPS [40], which instantaneously share link capacity fairly across all flows using a link. But maintaining per-flow state is generally considered too expensive in data-center switch ASICs with limited on-chip memory. For example, if a switch ASIC needs 8B of state for each flow, it would need 8MB for 1 million flows. A significant fraction of the chip would need to be reclaimed from lookup-tables and the packet buffer, both of which are in high demand. Moreover, any per-flow state solution would require a hard limit on the number of flows to be baked into silicon, limiting the scale. We would prefer switches to avoid any per-flow state in the first-place.

Another approach, taken by FlowTune [41], is to calculate a fair rate allocation for each flow in a centralized server, and schedule the flows to be sent at these rates. But a centralized scheduler is a bottleneck in a large network, with a rapidly changing set of flows.

We therefore seek fast congestion control algorithms that are practical, do not require per-flow state, allowing them to scale to any number of flows, and are distributed, allowing them to scale to any network size. The algorithms must converge quickly to fair rates for any number of flows. They must remain stable and fast in the face of sudden changes in the traffic matrix. Finally, they must be general enough to work for arbitrary topologies at WAN or DC scale.

Reactive algorithms: Why do reactive algorithms take hundreds of RTTs to converge? TCP (and variants such as DCTCP [3], RCP [17], XCP [28], Timely [35]) were designed to primarily operate at the end host with minimal knowledge of the network topology or current conditions. They do not know the link capacities or the number of flows on the links. These decentralized algorithms (usually) run on the end-host, where the algorithm measures congestion signals (e.g. packet drops or ECN marks) to iteratively choose a sending rate by successive approximation. This class of algorithms reacts to congestion signals and hence we call them *reactive* algorithms. They do not know the correct rates but they know the direction in which to adjust. To remain stable, they must take many small, cautious steps, and hence they take a long time to converge. At 100Gb/s most flows could have finished long before reactive algorithms find the correct rates.

Proactive algorithms: Our approach is to use distributed *proactive* (rather than reactive) algorithms to directly calculate the ideal flow rates. Building on previous PERC (Proactive Explicit Rate Control) algorithms [26], our main contribution is a new, practical, scalable algorithm called s-PERC. This is the first PERC algorithm that provably converges to max-min fair rates in known, bounded time without needing to keep per-flow state.

Introducing a new congestion control algorithm like s-PERC into a network presents many challenges. But as we have seen in recent years, cloud providers seem willing to invest the effort, given their homogeneous infrastructure and single administrative domain. Recent programmable switches make it practical to implement simple distributed algorithms at switches, that collect information about flows proactively and act on it quickly.

The s-PERC algorithm is a deceptively simple distributed algorithm: End hosts send and receive control packets that carry four fields (< 7B total) per link, and switches use these control packets to locally calculate the exact max-min fair flow rate, using a constant amount of state (8B per link) at the switch itself. s-PERC was designed to work with one particular fairness metric (max-min fairness) because it is a widely-used objective for congestion control algorithms. We believe it is possible to design proactive algorithms that use a different fairness metric and will converge quickly, but we have not done so.

Convergence result: The tricky part is understanding *why* s-PERC always converges in bounded time. To help our convergence proofs we introduce a family of *centralized* algorithms called k-Waterfilling algorithms (§3.1). The well-known sequential water-filling algorithm [11] is the special case when $k = \infty$. As we make k smaller, the algorithm becomes more parallel and needs fewer iterations to compute max-min fair rates. To set the stage for s-PERC, the paper first reviews an existing algorithm, called *Fair*, which uses per-flow state at the links. It is known that by calculating local max-min fair rates at every link, *Fair* leads to a global max-min fair allocation [43] (§4). Previous work shows that the convergence behavior of *Fair* can be analyzed using 1-Waterfilling [42], the fastest and most parallel of our water-filling algorithms.

In this paper, we show that the convergence behavior of s-PERC can be analyzed using 2-Waterfilling, the second most parallel water-filling algorithm. Specifically, we show that s-PERC is guaranteed to converge to the correct max-min fair rate allocation within $6N$ rounds, where N is the number of iterations 2-Waterfilling takes for the same routing matrix (§C.4). This is a tighter bound than we can obtain using the standard water-filling algorithm ($k = \infty$), but looser than the $k = 1$ bound for *Fair*. The intuition is that because *Fair* uses more state (specifically, per-flow state at every link) it can be made more parallel and therefore converges faster. But s-PERC also

runs in parallel and can converge much faster than standard water-filling, despite requiring no per-flow state at the links. Numerical simulations of a few thousand randomly-generated routing matrices indicate that s-PERC converges 10-40% slower than *Fair* in practice, and the worst-case convergence time for s-PERC was 2-3x faster than the $6N$ bound that we prove (§8.1). We evaluate s-PERC using packet-level simulations with realistic workloads in §8.2.1 and demonstrate that it converges an order of magnitude faster than existing reactive algorithms in data center networks, and achieves close to ideal throughput for large flows, and near-minimum latency for the smallest flows (§8.2.2). In WAN networks, s-PERC converges 1.3–6x faster than existing reactive algorithms. We enhance the basic s-PERC algorithm in §7.1 to make it robust in real WAN or DC networks (§8.1.2), and built a hardware prototype using the 40Gb/s NetFPGA SUME platform (§8.3).

2 DEFINITIONS

PERC algorithms figure out the max-min fair rate allocations by pro-actively exchanging messages (control packets) with switches along the path, over multiple rounds. The flow allocations are carried as explicit rates in the control packets.

We make the following assumptions about the network and operating conditions.

1. An arbitrary network with M links, with fixed link capacities denoted by vector $\mathbf{c}$, carrying a set of N flows.

2. An $M \times N$ 0-1 routing matrix $A = [a_{lf}]$ indicates which links a flow uses, i.e. $a_{lf} = 1$ iff flow f uses link l.

3. $A_{l\star}=\{f \mid a_{lf} = 1\}$ is the set of flows that use link l, and $n_l=|A_l|$ is the number of flows that use link l.

4. Similarly, $A_{\star f}=\{l \mid a_{lf} = 1\}$ is the set of links, used by flow f.

5. Flow rates are only limited by link capacity, not by constraints at the source or destination.

6. At any instant, each flow has exactly one control packet in flight. The control packet goes back and forth between the sender and receiver across the same set of links $A_{\star f}$ as the flow's data packets.

7. Control packets are in flight as long as a flow is active, and the control packet is never dropped or garbled. (We will relax this assumption later).

8. The control packet is only modified by the links; the end host does not modify the control packet, except to signal that a flow is starting or ending.

9. There is no coordination between the links, and the arrival order of control packets at the links can be arbitrary.

10. A flow's control packet carries allocated rate x for each link along its path. When a sender receives a control packet, it sets the sending rate equal to the minimum allocated rate in the control packet.

11. Control packets have a bounded round trip time. We define a *round* to be the maximum round trip time of all control packets; *i.e.* in one round, a link is guaranteed to see at least one control packet for every flow crossing it.

12. We define convergence time as the time from when the set of flows stabilizes until every flow has been allocated its correct max-min fair rate on every link.

2.1 Max-Min Fairness

Definition 1. A flow f is *bottlenecked* at link l if (i) the capacity of link l is fully utilized, and (ii) flow f gets the maximum rate of all flows that cross link l. A rate allocation is max-min fair *iff* every flow is bottlenecked at some link. There always exists a unique max-min fair allocation [31].

Example: Consider the network in Figure 2 with M=3 links carrying N=2 flows. The max-min fair allocation is 12Gb/s for (green) flow 2, which is bottlenecked at link l_{12}, and 18Gb/s for (blue) flow

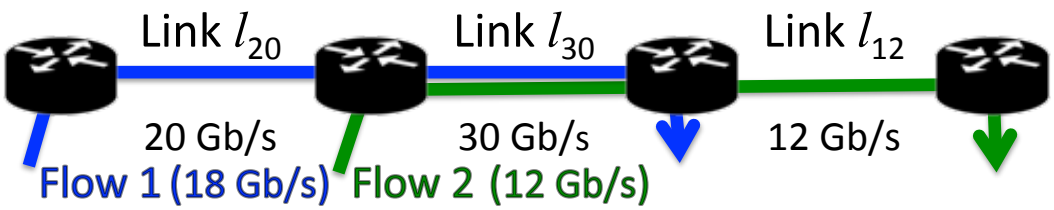


Fig. 2. A Running example. Flow 2 (green) is bottlenecked to 12Gb/s at link l_{12}; flow 1 (blue) is bottlenecked to 18Gb/s at link l_{30}.

1, which is bottlenecked at link l_{30}. Link l_{20} has spare capacity since (green) flow 2 is bottlenecked elsewhere.

Notation: Vector $\mathbf{x}^*$ is the max-min fair rate allocation for the flows. The max-min fair rate of a *link* is the max-min fair allocation of flows that are bottlenecked at the link. This is well defined for links whose capacities are fully utilized, i.e. $\{l \mid (A\mathbf{x}^*)_l = \mathbf{c}_l\}$. If a link is not fully utilized, we say its max-min rate is unbounded. Vector $\mathbf{r}^*$ denotes the max-min fair rate of the links.

3 PROACTIVE (PERC) ALGORITHMS

Our main contribution is s-PERC, a distributed PERC algorithm that does not maintain per-flow state at the switches.

In order to motivate s-PERC, we start with an existing algorithm called *Fair* which requires per-flow state, then describe an algorithm called n-PERC where we naively get rid of the per-flow state. We will see that n-PERC runs into transient problems that delay convergence. By fixing the transient problem in n-PERC, we arrive at the s-PERC algorithm that converges in a known bounded time.

Only *Fair* and s-PERC are proven to converge to max-min in a known bounded time. Simulations suggest that n-PERC converges but can take arbitrarily long because of the transient problems.

The three algorithms have a common template. The control packet of a flow carries an allocated rate x and a bottleneck rate b for each link along the flow's path. These are calculated when the packet is updated by a link and saved in the control packet at the end of the update.

Links update control packets in four steps. **Step 1**: The link assumes the flow is going to be bottlenecked here and computes bottleneck rate b for the flow, which is the link's estimate of its max-min rate $\mathbf{r}^*$ based on local information. **Step 2:** The link determines e, the lowest bottleneck rate that the flow gets from any *other* link in the network. We call e the *limit rate* for the flow. **Step 3:** The link calculates the allocated rate x for the flow, as the minimum of $\{b, e\}$. **Step 4:** Finally, the link updates its local state, and the control packet, and forwards the control packet. The source end-host updates the actual sending rate of the flow each time a control packet is received. The sending rate is set to the minimum allocated rate at any link ($\min \mathbf{x}$).

Each algorithm differs in the amount of state it maintains at the links, and how it computes the bottleneck rate. *Fair* maintains per-flow state, whereas n-PERC and s-PERC do not. *Fair* uses per-flow state to store the limit rates of all flows at a link, and it uses this state to compute a locally max-min fair bottleneck rate b for a flow upon receiving its control packet. n-PERC relies instead on two aggregate counters to calculate b: $NumB$, the number of flows *estimated* to be bottlenecked at the link; $SumE$, the sum of allocated rates of flows *estimated* to be bottlenecked elsewhere. The

rate calculation is based on the following characterization of a link's actual max-min fair rate:

$$\mathbf{r}^* = \frac{\mathbf{c} - SumE^*}{NumB^*}, \quad (1)$$

where $NumB^*$ is the number of flows *actually* **B**ottlenecked at the link, and $SumE^*$ is the sum of max-min fair rates of flows not bottlenecked at the link, but bottlenecked **E**lsewhere. In other words, for any link that is fully used, every flow bottlenecked at the link gets an equal share of the remaining capacity, after we remove the allocations of flows that are not bottlenecked at the link. This folows from Definition 1 of max-min fairness. Upon receiving a flow's control packet, an n-PERC link uses its *estimates*, *SumE* and *NumB*, to calculate the bottleneck rate in one-shot as b=($\mathbf{c} - SumE$)/$NumB$.

Without per-flow state, inaccurate estimates of which flows are bottlenecked at a link lead to transient problems in n-PERC, where a link propagates a bottleneck rate that is too low and causes other links to make errors. s-PERC fixes this problem by keeping a bit more state at each link: *MaxE*, an estimate of the maximum bandwidth allocated to flows that are assumed to be bottlenecked elsewhere. As shown later, a link can use *MaxE* to determine when it is safe to propagate bottleneck rate information to other links, and this is enough to ensure a bounded convergence time. Table 1 summarizes the the information carried in the control packets (second column, bold) and the state maintained at the links by each algorithm (third column.) The information in the control packet is carried for each link, and directly reflects the variables used by the link to compute the flow's latest allocation.

Table 1. Summary of variables used by PERC algorithms in this paper. Bold variables are carried by the control packet, per-link.

PERC Algorithm	Variables (**Packet State**)	Link State	Convergence Time (rounds)
Fair [43]	**b**, e, **x**	e, per flow	$4N_1$
n-PERC [15]	**b**, e, **x**, **s**	$SumE$, $NumB$	bound unknown
s-PERC	**b**, e, **x**, **s**, **i**	$SumE$, $NumB$ $MaxE$, $MaxE'$	$6N_2$

Notation. b: bottleneck rate, e: limit rate, x: allocated rate, s: bottleneck state ∈{E, B}, i: ignore bit (i=0 to propagate b), N_k: number of iterations k-Waterfilling takes.

3.1 k-Waterfilling algorithms

Our goal is to find a convergence bound for s-PERC. We could easily find an upper-bound $O(N)$ (for *Fair* and s-PERC) if there are N bottleneck links. We are going to find a tighter bound with the help of k-Waterfilling algorithms, a family of centralized iterative algorithms that compute max-min fair allocations, defined in Algorithm 1. The algorithm takes as input a fixed routing matrix and a vector of link capacities. In each iteration, a k-Waterfilling algorithm computes a fair share rate for each link (line 5), and picks a subset of the bottlenecked links (a function of k) to remove (lines 6–8). It assigns the fair share rate of the selected bottlenecked links to their flows (lines 9–11), and removes the links and the flows from the network (lines 12–17). The rates of the flows that are removed are subtracted from the remaining link capacities (line 13), and this yields new fair share rates in the next iteration. Which links are removed depends on k, as described below. The algorithm terminates when there are no more flows.

Algorithm 1 The k-Waterfilling algorithm to compute max-min rates.

1: A : routing matrix, $\mathbf{c}$: vector of link capacities ▷ Input
2: $\mathbf{n} \leftarrow A\mathbf{1}$ ▷ Number of flows per link
3: $\mathbf{x} \leftarrow \mathbf{0}$ ▷ Rate allocation per flow
4: **while** $|\mathbf{n}| > 0$ **do**
5: $\quad \mathbf{r} \leftarrow \mathbf{c}/\mathbf{n}$ ▷ Compute fair share rate
6: $\quad$ **if** $k = \infty$ **then** $\mathcal{L} \leftarrow \{l \mid \mathbf{r}[l] = \min \mathbf{r}\}$ ▷ Select links with lowest rates globally
7: $\quad$ **else**
8: $\quad\quad S \leftarrow (AA^T)^k$ ▷ $S[l, m] > 0$ iff links l, m within distance k in Link Dependency Graph
9: $\quad\quad \mathcal{L} \leftarrow \{l \mid \mathbf{r}[l] = \min_{\{m|S[l,m]>0\}} \mathbf{r}[m]\}$ ▷ Select links with lowest rates in k-neighborhood
10: $\quad$ **for all** $l \in \mathcal{L}$ **do** ▷ Set of links to remove
11: $\quad\quad$ **for all** $f \in A[l, \cdot]$ **do**
12: $\quad\quad\quad \mathbf{x}[f] \leftarrow \mathbf{r}[l]$ ▷ Assign fair share rate of link to flow
13: $\quad \mathcal{F} \leftarrow \{f \mid \mathbf{x}[f] > 0\}$ ▷ Set of flows to remove
14: $\quad \mathbf{c} \leftarrow \mathbf{c} - A\mathbf{x}$ ▷ Subtract rate of flows to remove ($\mathcal{F}$) from capacities of all links on their path
15: $\quad \mathbf{c} \leftarrow \mathbf{c}(\mathcal{L})$, ▷ Remove $\mathcal{L}$, $\mathcal{F}$ from vector of link capacities,
16: $\quad A \leftarrow A(\mathcal{L}; \mathcal{F})$ ▷ ... routing matrix,
17: $\quad \mathbf{x} \leftarrow \mathbf{x}(\mathcal{F})$ ▷ ... and vector of rate allocations.
18: $\quad \mathbf{n} \leftarrow A\mathbf{1}$ ▷ Update vector of number of flows per link

To understand which links (and their flows) are removed in each iteration, it helps to consider an alternate representation that we call the *Link Dependency Graph*. Each vertex represents a link, and there is an edge between two vertices if and only if the corresponding links share a flow. Figure 3 shows an example Link Dependency Graph, where vertices 1 and 2 share an edge because links 1 and 2 share a (blue) flow.

In each iteration, the fair share rate of a link is defined to be c/n, where c is the remaining link capacity, and n is the number of flows using the link (line 5). The k-Waterfilling algorithm removes a link in an iteration if it has the lowest fair share rate of all links within distance k in the Link Dependency Graph — links that we call kth-degree neighbors (Lines 7–8, noting that AA^T indicates adjacent links in the Link Dependency Graph).

Figure 4 (left side) shows the progression of the 1-Waterfilling algorithm for the example. Link 1 is removed in the first iteration because its first-degree neighbor, link 2, has a fair share rate that is less than or equal (both fair share rates are 5Gb/s). On the other hand, link 2 is not removed in the first iteration because its fair share rate is larger than its first-degree neighbor, link 3.

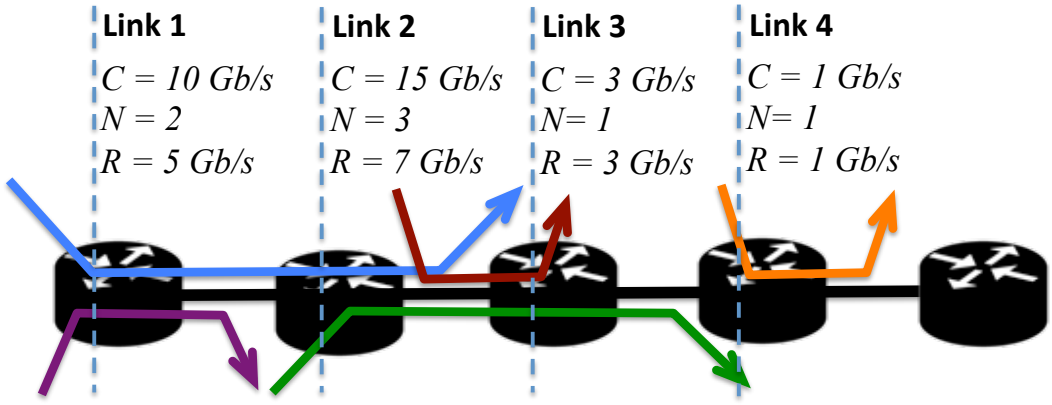
All k-Waterfilling algorithms compute max-min fair rates in a finite number of iterations.

Theorem 2 (k-Waterfilling Correctness)**.** *The k-Waterfilling algorithm (Algorithm 1) terminates after a finite number of iterations N_k, and all flows in the network are allocated their max-min fair rates.*

Proof. See Appendix A.1. □

While all k-Waterfilling algorithms converge to the same, unique set of max-min fair rates, they differ in how they get there. k-Waterfilling algorithms partition the set of links in the network, based on the iteration in which a link is removed. The number and sequence of partitions depends on k. For example, in Figure 4, there are N_1=2 sets for 1-Waterfilling, and the first set of bottleneck links (gray links in the first row) contains three links, 1, 3 and 4, each of which have the lowest fair share rate in their k=1 neighborhood. But if k=2, there are N_2=3 sets, and the first set has two links, 3 and 4, whose fair share rate is the lowest of second-degree neighbors.

It should be clear that 1-Waterfilling converges in the fewest iterations because local calculations depend only on the immediately adjacent neighbor. At the other extreme, k=∞ (the serial centralized waterfilling algorithm introduced by Bertsekas [11]), takes the longest to converge. In each iteration



(a) Topology and set of flows.

(b) Link Dependency Graph.

Fig. 3. Example network and Link Dependency Graph. There is an edge between vertices 1 and 2 in the Link Dependency Graph because links 1 and 2 share a common (blue) flow. We say that link 2 and link 3 and first- and second-degree neighbors respectively, of link 1.

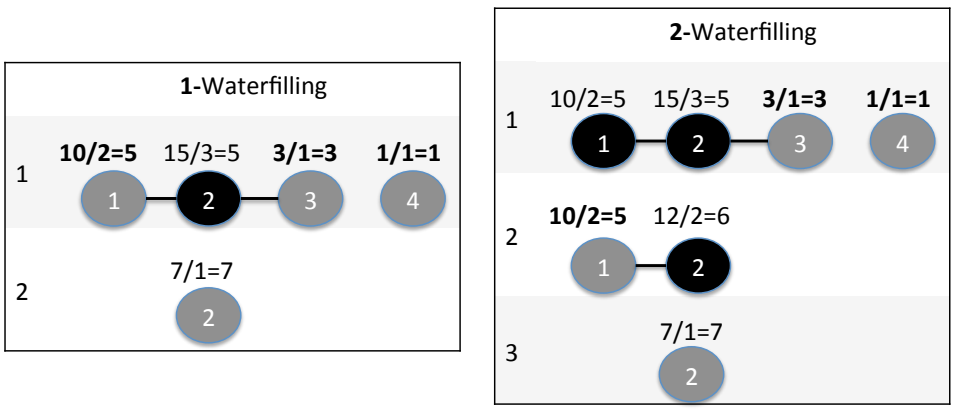


Fig. 4. k-Waterfilling (k=1 and k=2) for example in Figure 3. Gray links have the lowest fair share rates in their k-neighborhood and are removed at the end of the iteration.

a link is removed if it has the lowest fair share rate out of *all* links that remain in the iteration, not just its neighbors.

k-Waterfilling algorithms are useful for proving properties of the *Fair* and s-PERC algorithms because, as we will show, flows converge to their correct max-min allocations in these algorithms in exactly the same sequence that they are removed in a corresponding k-Waterfilling algorithm.

We can therefore prove convergence bounds for these distributed PERC algorithm in terms of the appropriate k-Waterfilling algorithm, as shown by the next two theorems.

Theorem 3 (Convergence of *Fair*)**.** *With Fair, given a fixed* $(A, \mathbf{c})$*, once all flows have been seen at their links, every flow converges to its max-min fair rate in less than or equal to* $4N_1$ *rounds, where* N_1 *is the number of iterations that* 1*-Waterfilling takes for* $(A, \mathbf{c})$*.*

Proof. See [43] or Appendix B.2. □

Theorem 4 (Convergence of s-PERC)**.** *With s-PERC, given a fixed* $(A, \mathbf{c})$*, once all flows have been seen at their links, every flow converges to its max-min fair rate in less than or equal to* $6N_2$ *rounds, where* N_2 *is the number of iterations that* 2*-Waterfilling takes for* $(A, \mathbf{c})$*.*

Proof. See Appendix C.4. □

In PERC algorithms, a link propagates its bottleneck rate information to adjacent links in the Link Dependency Graph via the control packets of the flows that it shares with those links. This information helps the links along the path of a flow estimate where the flow is bottlenecked, enabling them to update their own bottleneck rates appropriately. When a flow has converged to the correct max-min fair (sending) rate, its bottleneck link has calculated a bottleneck rate equal to the flow's max-min rate, and every other link has calculated a higher bottleneck link rate (because the flow is not bottlenecked there). The crux of Theorems 3 and 4 is that a link in *Fair* converges (i.e. all of its flows converge) if it has the lowest fair share rate (c/n) among its first-degree neighbors in the Link Dependency Graph, whereas in s-PERC the link must *also* have the lowest fair share rate of its second-degree neighbors. The main difference between *Fair* and s-PERC is in their bottleneck rate calculations and how (and when) they propagate the rate to other links. *Fair* always calculates a bottleneck rate greater than or equal to its fair share rate, c/n. *Fair* can do this because it maintains per-flow state for the limit rates of the flows and can therefore perform a locally max-min bottleneck rate calculation at each link (see §4). Informally, the consequence is that a link, say l, that has the *lowest* fair share of its first-degree neighbors, converges because every first-degree neighbor of l will compute a bottleneck rate that is greater than or equal to link l's fair share rate. Therefore, every flow crossing link l will be classified correctly at both link l (as bottlenecked here) and the first-degree neighbors of l (as bottlenecked elsewhere).

In s-PERC, however, there is no guarantee that the bottleneck rate computed at a link is greater than or equal to its fair share rate. And so to protect against propagating rates that might confuse a neighbor, s-PERC ensures that the rate *propagated* to other links is greater than or equal to the link's fair share rate (see §6.2). This means that if a link l has the lowest fair share rate of its first-degree neighbors only, its flows are allocated this rate at link l but not necessarily at other links along their path (the first-degree neighbors of link l), because the bottleneck rate computed at those links might be lower than link l's fair share rate. If, however, link l has the lowest rate of both its first- and second-degree neighbors, the first-degree neighbors of l will allocate the correct rate, because the rates propagated to them by their neighbors (the second-degree neighbors of link l) is at least link l's fair share rate.

We discuss the connection between *Fair* and s-PERC to the 1-Waterfilling and 2-Waterfilling algorithms respectively in more detail in §6.2.

4 *FAIR*: A MAX-MIN PERC ALGORITHM WITH PER-FLOW STATE

We start with *Fair*, which uses per-flow state and follows from a well-known result [43] that computing local max-min fair allocations at every link can lead to the global max-min fair allocation

Algorithm 2 *Fair* alg. at link l to process control packet of flow f.

```
1: b, x : vector of bottleneck and allocated rates in control packet (initially, ∞, 0, respectively)
2: e : vector of limit rates at link (initially empty)
3: e[f] ← ∞                          ▷ Start of local max-min fair calculation. Assume flow f is not limited.
4: Sort e and let e^(i) denote ith largest rate
5: SumE ← 0, NumB ← |e|, i ← 0
6: while (c − SumE)/NumB > e^(i) do
7:     SumE ← SumE + e^(i)
8:     NumB ← NumB − 1
9:     i ← i + 1
10: b ← (c − SumE)/NumB              ▷ End of local max-min calculation
11: b[l] ← ∞                         ▷ Assume the link's own bottleneck rate is ∞ to get limit rate
12: e ← min b
13: x ← min(b, e)
14: b[l] ← b, x[l] ← x               ▷ Save variables to packet and link
15: if flow is leaving then del e[f] ▷ Remove flow f from vector of limit rates
16: else e[f] ← e
```

for the network.[1] *Fair* and its convergence analysis follow directly from existing work by Ros-Giralt *et al.* [42–44]. The *Fair* control packet carries for each link, a bottleneck rate b and the bandwidth allocated to the flow by the link, x. The link uses per-flow state to store the limit rates of all flows. See Algorithm 2 for a complete description of the control packet and the algorithm. *Fair* provably converges to the global max-min fair rates in $4N_1$ rounds, where N_1 is number of iterations that 1-Waterfilling takes for the given set of flows and links.

4.1 The local max-min fair rate

Fair calculates the local max-min fair share rate as follows: a link starts by assuming all flows are bottlenecked here (line 5 in Algorithm 2) with initial fair rate $\mathbf{c}/\mathbf{n}$. The link sorts flows by their limit rates, and iteratively moves flows out of the bottlenecked set (lines 7–8) until the resulting fair rate $(\mathbf{c} - SumE)/NumB$ no longer exceeds the limit rate of the flow (line 6). Eventually, the fair share rate at a link uniquely solves:

$$b = \frac{\mathbf{c} - \sum_{f \in E_b} \mathbf{e}_f}{|B_b|}, \tag{2}$$

where $E_b = \{f \mid \mathbf{e}_f < b\}$ is the set of flows with limit rates smaller than b, and $B_b = \{f \mid \mathbf{e}_f \geq b\}$ is the set of flows with limit rates at least b. The rate b is the max-min rate for the link, assuming each flow that uses the link crosses one additional link whose capacity is equal to the flow's limit rate. The flow that is being updated is assumed to have limit rate $e = \infty$. This ensures that Eq. (2) has a unique solution.

4.2 The *Fair* Algorithm in Action

Let's walk through an example to understand how the rates evolve in the *Fair* algorithm (Table 2) in our running example from Figure 2. Later, we will contrast it with how the rates evolve (badly) when we don't have per-flow state.

The numbers in parentheses refer to the index of the update in table 2. (1) Flow 1 is first seen at link l_{20}. The link assumes the flow is not limited elsewhere, computes a bottleneck rate of 20Gb/s,

[1] *Fair* has minor differences from d-CPG [43], as described in Appendix B.1. We present *Fair* instead because it is easier to contrast with the two PERC algorithms, since they have a common template.

Table 2. Control packet updates for the first two rounds of *Fair*.

Rounds	Update	Flow / link		b	e	x
1	1	1	l_{20}	20	∞	20
	2		l_{30}	30	20	20
	3	2	l_{30}	15	∞	15
	4		l_{12}	12	15	12
2	5	2	l_{30}	15	12	12
	6		l_{12}	12	15	12
	7	1	l_{20}	20	30	20
	8		l_{30}	18	20	18

and allocates it to the flow. (2) The flow is then seen at link l_{30}. The link computes a bottleneck rate of 30Gb/s, sees that the flow is limited to 20Gb/s, and allocates 20Gb/s.

(3) When flow 2 is seen at link l_{30}, the link knows that one other flow is limited to 20Gb/s. It assumes flow 2 is not limited (e=∞), and a local max-min fair calculation yields a bottleneck rate of 15Gb/s ($SumE = 0$, $NumB = 2$ in line 9.) In terms of Eq. 2, both flows are in B_b, with limit rates at least 15Gb/s. The bottleneck rate 15Gb/s is smaller than the actual limit rate (∞), and the link allocates 15Gb/s to flow 2. (4) Next, the flow is seen at link l_{12}, where its limit rate 15Gb/s exceeds the bottleneck rate 12Gb/s, and the flow gets it correct max-min fair allocation. A key property of the *Fair* algorithm is that the bottleneck rate calculation at each link is *locally max-min fair*, a property lacking in algorithms that don't use per-flow state. During update (3), given a limit rate of 20Gb/s for flow 1 and ∞ for flow 2, an allocation of the bottleneck rate 15Gb/s for both flows is (locally) max-min fair. They are both bottlenecked at the link, the link is fully used and both flows get the maximum rate.

As Table 2 shows, at the end of the second round of updates, when each flow has been updated at each link twice, *Fair* converges to the max-min rates for flow 2. Flow 1 needs one more update at link l_{20} to be allocated its max-min rate of 18Gb/s (not shown).

4.3 Convergence of the *Fair* Algorithm

Let $\mathcal{F}$ and $\mathcal{L}$ denote the set of flows and links removed in the *first* iteration of 1-Waterfilling. We can show that within a round every flow $f \in \mathcal{F}$ is allocated a bottleneck rate at least $\mathbf{x}^*_f$. This follows from (i) the property of the local max-min fair calculation (it is at least $\mathbf{c}/\mathbf{n}$), and (ii) the 1-Waterfilling property that if a flow is removed by link l, then the fair share rate of any link that carries f must be at least $\mathbf{c}_l/\mathbf{n}_l = \mathbf{x}^*_f$. By the end of round two, the limit rate of all flows at every link $l \in \mathcal{L}$ is at least $\mathbf{c}_l/\mathbf{n}_l$, because any flow that the link carries is in $\mathcal{F}$, and such flows get a bottleneck rate of at least $\mathbf{c}_l/\mathbf{n}_l$ at all other links on their paths. Therefore, all flows are bottlenecked at link $l \in \mathcal{L}$ with local max-min fair rate $\mathbf{c}_l/\mathbf{n}_l$. In the third round, flows in $\mathcal{F}$ pick up their local max-min rate from links in $\mathcal{L}$, and they are allocated this rate by the remaining links by the fourth round. Thus after four rounds, the flows in $\mathcal{F}$ and the links in $\mathcal{L}$ have converged. Next we remove these flows and links and repeat the same analysis. Eventually all flows are updated correctly within $4N_1$ rounds. See Appendix B.2 for a formal proof.

5 N-PERC: A NAIVE PERC ALGORITHM WITHOUT PER-FLOW STATE

In our search for a stateless algorithm, the n-PERC algorithm is an obvious place to start. Given a max-min fair allocation, each link carries two sets of flows, one set of flows is **B**ottlenecked at the link, which we denote using B^*, the other set of flows is not bottlenecked at l, but is bottlenecked

Algorithm 3 n-PERC. Link l processes control packet of flow f.

```
1: b, x, s : vector of bottleneck, allocated rates and bottleneck states in packet (initially ∞, 0, E, respectively)
2: SumE, NumB : sum of limit rates of E flows, and number of B flows at link (initially 0 each)

3: if s[l] = E then                          ▷ Assume flow is not limited, for bottleneck rate calculation
4:     s[l] ← B
5:     SumE ← SumE − x[l]
6:     NumB ← NumB + 1
7: b ← (c − SumE)/NumB
8: b[l] ← ∞                                  ▷ Assume the link's own bottleneck rate is ∞
9: e ← min b
10: x ← min(b, e)
11: b[l] ← b, x[l] ← x                       ▷ Save variables to packet
12: if flow is leaving then NumB ← NumB − 1  ▷ Remove flow f
13: else if e < b then
14:     s[l] ← E
15:     SumE ← SumE + x
16:     NumB ← NumB − 1
```

Elsewhere, E^*. This was implicit when we characterized the max-min rate of a link as (reproducing Eq. (1) in §3):

$$\mathbf{r}^* = \frac{\mathbf{c} - SumE^*}{NumB^*}.$$

In n-PERC, each link aggregates per-flow state into two variables: $SumE$ is the sum of allocated rates of flows *estimated* to be in E^*; $NumB$ is the number of flows *estimated* to be in B^*. In other words, these are estimates of $SumE^*$ and $NumB^*$. The control packet carries for each link, a bottleneck rate, b; the bandwidth allocated to the flow by the link, x; and a 1-bit variable, $s \in \{B, E\}$, indicating whether or not the flow is believed to be bottlenecked at this link. As shown in Algorithm 3, upon receiving a control packet, links use the above equation to compute a bottleneck rate (line 7), replacing the actual values $SumE^*$ and $NumB^*$ by the estimates $SumE$ and $NumB$, respectively, to get:

$$b = \frac{\mathbf{c} - SumE}{NumB}. \tag{3}$$

The link then compares the bottleneck rate with the latest limit rate of the flow and re-classifies the flow into E or B accordingly (lines 13–16 if E, otherwise B by default because of lines 3–6). In other words, in the n-PERC algorithm, instead of maintaining per-flow state, the link determines E and B (estimates of E^* and B^*) based on the state carried in the control packets. Therefore, $SumE$ and $NumB$ are also determined by the control packet state.

5.1 The n-PERC Algorithm in Action

There are cases where n-PERC converges very slowly. The following example illustrates the problem; we will see that it is the link with the lowest max-min rate that struggles to accurately identify its bottleneck flows and slows down the convergence.

Let us walk through the example in Table 3. We will use the same topology and workload as for *Fair* (Figure 2). Examining the first four updates will be sufficient for the discussion that follows. The numbers in parentheses refer to the update in Table 3. The allocations start off the same as *Fair*. (1, 2) Flow 1 is first seen at link l_{20} and then link l_{30}. It is classified into B at the first link, and allocated 20Gb/s, and classified into E at the second link, and allocated its limit rate 20Gb/s, which is less than the bottleneck rate 30Gb/s. Now things start to differ from *Fair*. (3) When flow 2 is first seen at link l_{30}, the link knows that flows in E (flow 1) have been allocated 20Gb/s. It assumes flow

Table 3. Control packet updates for the first three rounds of n-PERC.

Rounds	Update	Flow	/ link	b	e	x	s
1	1	1	l_{20}	20	∞	20	B
	2		l_{30}	30	20	20	E
	3	2	l_{30}	**10**	∞	10	B
	4		l_{12}	12	**10**	10	E
2	5	2	l_{30}	10	12	10	B
	6		l_{12}	12	10	10	E
	7	1	l_{20}	20	30	20	B
	8		l_{30}	15	20	15	B
3	9	2	l_{30}	15	12	12	E
	10		l_{12}	12	15	12	B
	11	1	l_{30}	18	20	18	B
	12		l_{20}	20	18	18	E

2 is not limited ($NumB$=1) and uses Eq. (3) with $NumB$=1 and $SumE$=20 to compute a bottleneck rate of b=10Gb/s. Since the flow has not been seen anywhere else yet, its limit rate is ∞, and the link classifies the flow into B, allocates the bottleneck rate 10Gb/s (bolded in table), and updates the control packet to reflect this. This is different from $Fair$, which computed a bottleneck rate of b=15Gb/s and allocated x=b=15Gb/s to the flow.

(4) When flow 2 is next seen at its actual bottleneck link l_{12}, its limit rate is e=10Gb/s, which is the bottleneck rate propagated by link l_{30} (bolded in table). This is lower than the link's correct bottleneck rate b=12Gb/s and the link fails to recognize its bottleneck flow. The link assumes the flow is bottlenecked elsewhere and allocates only 10Gb/s. This is a problem because despite having the lowest max-min rate of all links, link l_{12} cannot immediately recognize flow 2 as a bottlenecked flow.

5.2 Transient Problems With n-PERC

The difference between n-PERC and $Fair$ gives us insight into why it is hard to bound n-PERC's convergence time. If we consider flow 2's update at link l_{30} (where, in update (3), it is labeled B and allocated b=10Gb/s) and the subsequent update at l_{12}, (where it is labeled E and allocated e=10Gb/s), two transient problems with n-PERC become apparent.

1. Sub-optimal local rates: The first problem is that the B flows are allocated a lower rate than the locally max-min fair rate. To see this, consider the rate allocation at link l_{30} at the end of update 3. Link l_{30} considers flow 1 bottlenecked elsewhere to 20Gb/s and flow 2 bottlenecked here to 10Gb/s. A B flow is allocated less than an E flow, which should not happen. n-PERC cannot avoid this problem, because link l_{30} does not know the rates of individual E flows. It only knows that their total rate is $SumE$ = 20Gb/s. Thus it has no way of knowing whether the flows in E have a rate lower or higher than its estimated bottleneck rate of 10Gb/s.

2. Bad bottleneck rate propagation: The second problem is that when a flow's sub-optimal bottleneck rate is propagated to its *actual* bottleneck, the true bottleneck link may not realize the flow is bottlenecked. For example, flow 2 is actually bottlenecked at link l_{12}, but picks up a rate of b=10Gb/s from link l_{30}, which becomes its limit rate at l_{12}. At l_{12}, since the limit rate e=10Gb/s is lower than the bottleneck rate b=12Gb/s, the flow is wrongly classified into E and allocated only e=10Gb/s. Link l_{12} does not immediately recognize the flow as a bottleneck flow and must wait until update 9, when l_{30}'s bottleneck rate for flow 2, b=15Gb/s, is high enough.

In comparison, the *Fair* algorithm does not propagate bad rates. Since link l_{12} has the lowest max-min rate 12Gb/s, any other link has at least 12Gb/s available for each of its flows, including l_{12}'s bottleneck flows. So the local max-min fair rate at every other link is at least 12Gb/s. Notice that the (last seen) limit rates of the flows at link l_{30} during update 3 are identical for both the *Fair* algorithm and n-PERC. Yet, while *Fair* computes bottleneck rate 15Gb/s, which is higher than l_{12}'s max-min fair rate, n-PERC computes a bottleneck rate that is lower. A local max-min fair rate is always a good bottleneck rate and high enough to propagate (see proof of convergence of *Fair* in Appendix B.2). But it is impossible to calculate a locally max-min fair rate without storing the rate of every flow.

Despite n-PERC's transient problems, extensive numerical simulations suggest that the algorithm still *eventually* stabilizes (see §8.1), but there is no known upper bound for how long it can take. A similar algorithm has been proved to stabilize eventually [15], but it has no upper bound either.

6 S-PERC: A STATELESS ALGORITHM WITH KNOWN CONVERGENCE TIME

s-PERC overcomes n-PERC's slow convergence problem by not propagating a bottleneck rate if it thinks it might be too low. Clearly, a link *should* propagate the bottleneck rate once it is correct and equals the max-min rate. But if the bottleneck rate is not yet correct, especially if it suspects a flow is bottlenecked at a different link with a lower max-min rate, it would be better to say that the flow is not limited here. This is the idea behind s-PERC.

s-PERC maintains a new variable at each link called $MaxE$, which is an estimate of the maximum bandwidth allocated to a flow assumed to be in E^*. s-PERC propagates a bottleneck rate b only when $b \geq MaxE$. Why does this work? The bottleneck rate is just the remaining capacity after removing allocations of all flows in E, (i.e. $\mathbf{c} - SumE$), divided evenly among the flows in B. If $b < MaxE$, then a flow in E has been allocated more than the bottleneck rate (as in the n-PERC example) and that flow in E may need to be reclassified into B. In other words, if $b < MaxE$ then we have misclassified the flows in E, and we should not propagate the low bottleneck rate.

Algorithm 4 describes the s-PERC algorithm running at the links, each time a control packet is received. Algorithm 5 is run periodically every round. The control packet now carries a new per-link field, called an *ignore* bit **i**, which is set if the link wants the bottleneck rate to be ignored, and say instead, that the flow is not limited at the link. The remaining fields **b**, **x**, and **s** are the same as n-PERC. The state maintained by the link includes $SumE$ and $NumB$ (same as n-PERC) and two new variables, $MaxE$ and $MaxE'$, which are used to *estimate* the maximum allocation of a flow in E at any time: $\max_{f \in E} \mathbf{x}_f$. s-PERC does not track the exact maximum because it would require per-flow state [5].

The per-packet algorithm at the link is similar to n-PERC except for the following. The link uses the ignore bits to infer **p**, the "propagated bottleneck rate" of other links, that is, the rate that the flow is limited to at other links. If the ignore bit is set for a link j ($i[j] = 1$), it is assumed that the flow is not limited at that link, $p[j] = \infty$, otherwise the flow is assumed to be limited to the bottleneck rate, $p[j] = b[j]$ (lines 10–11). The link sets the flow's limit rate e to be the lowest *propagated bottleneck rate* from other links. If there is no other link, or none of the other links propagates a bottleneck rate, the flow's limit rate is set to $e = \infty$ (lines 12–13). To avoid propagating too low a rate, if $b < MaxE$, it sets the ignore bit in the control packet (line 16).

The new variables, $MaxE$ and $MaxE'$ are updated when a flow with a higher limit rate is classified into E (lines 21–22 in Algorithm 4), and they are periodically reset at the end of each round (Algorithm 5). This simple algorithm ensures that $MaxE$ is always greater than or equal to the largest allocation of a flow in E (Lemma 9). Further, Lemma 10 proves that if the maximum value stabilizes at time t, then $MaxE$ correctly reflects it within 2 rounds.

Algorithm 4 s-PERC: link l processing control packets for flow f.
Differences from n-PERC are highlighted.

b, **x**, **s** : vector of bottleneck, allocated rates, bottleneck states in packet (initially, ∞, 0, E, respectively)
i: vector of ignore bits in packet (initially, 1)
$SumE$, $NumB$: sum of limit rates of E flows, and number of B flows at link
$MaxE$, $MaxE'$: max. allocated rate of flows classified into E since last round (and in this round, respectively) at link

if **s**$[l]$ = E **then** ▷ Assume flow is not limited, for bottleneck rate calculation
    s$[l] \leftarrow$ B
    $SumE \leftarrow SumE -$ **x**
    $NumB \leftarrow NumB + 1$
$b \leftarrow$ (**c** $- SumE)/NumB$
foreach link j:
 if **i**$[j] = 0$ **then** **p**$[j] \leftarrow$ **b**$[j]$ **else** **p**$[j] \leftarrow \infty$ ▷ Propagated rates
p$[l] \leftarrow \infty$ ▷ Assume the link's own propagated rate is ∞
$e \leftarrow \min$ **p**
$x \leftarrow \min(b, e)$
b$[l] \leftarrow b$, **x**$[l] \leftarrow x$ ▷ Save variables to packet
if $b < MaxE$ **then** **i**$[l] \leftarrow 1$ **else** **i**$[l] \leftarrow 0$ ▷ Indicate if rate **b**$[l]$ should be ignored.
if flow is leaving **then** $NumB \leftarrow NumB - 1$ ▷ Remove flow f
else if $e < b$ **then**
    s$[l] \leftarrow$ E
    $SumE \leftarrow SumE + x$
    $NumB \leftarrow NumB - 1$
    $MaxE \leftarrow \max(x, MaxE)$; $MaxE' \leftarrow \max(x, MaxE')$.

Algorithm 5 Timeout action at link l for s-PERC, every round

$MaxE \leftarrow MaxE'$; $MaxE' \leftarrow 0$.

Table 4. Control packet updates for first three rounds of s-PERC.[2]

Rounds	Update	Flow	link	MaxE	b	e	x	s	**i**
1	1	1	l_{20}	0*	20	∞	20	B	
	2		l_{30}	0*	30	20	20	E	
	3	2	l_{30}	20	**10**	∞	10	B	1
	4		l_{12}	0*	12	∞	12	B	
2	5	2	l_{30}	20	10	12	10	B	1
	6		l_{12}	0	12	∞	12	B	
	7	1	l_{20}	0	20	30	20	B	
	8		l_{30}	20	15	20	15	B	1
3	9	2	l_{30}	0*	15	12	12	E	
	10		l_{12}	0	12	15	12	B	
	11	1	l_{30}	12	18	20	18	B	
	12		l_{20}	0	20	18	18	E	

6.1 s-PERC in Action

Let's walk through an example to understand how the rates evolve in the s-PERC algorithm (Table 4). We will use the same topology and workload we used for n-PERC and *Fair* (Figure 2).

[2]Note asterisk indicates $MaxE$ was reset before the update, and **i**=0 if not shown .

The numbers in parentheses refer to the update Table 3. (1, 2) Flow 1 is first seen at link l_{20} and then link l_{30}. It is classified into B at the first link, and allocated 20Gb/s, and classified into E at the second link, and allocated its limit rate 20Gb/s, which is smaller than the bottleneck rate 30Gb/s. So far the allocations are exactly the same as the *Fair* and n-PERC algorithms.

(3) The next update is the same as in n-PERC, up until where the link decides to not propagate the rate. When flow 2 is first seen at link l_{30}, the link uses Eq. (3) — with $NumB$=1, and $SumE$=20 — to compute a bottleneck rate of b=10Gb/s. Since the the actual limit rate is ∞, the link classifies the flow into B, allocates the bottleneck rate 10Gb/s, and updates the control packet to reflect this. *However*, because $MaxE$=20Gb/s is higher than b, the link guesses that b may be too low to propagate to the next link and additionally sets the ignore bit to 1 in the control packet.

(4) When flow 2 is next seen at its actual bottleneck link l_{12}, its bottleneck rate 10Gb/s from link l_{30} is ignored and the link assumes flow 2 is not limited (e=∞). The link computes a bottleneck rate b=12Gb/s. Since the limit rate is greater, the link correctly assumes the flow is bottlenecked at the link and allocates b=12Gb/s. By not propagating its bad bottleneck rate, link l_{30} enables link l_{12} to immediately identify its bottleneck flow 2.

The allocation at link l_{30} for flow 2 is still 10Gb/s however. It is not until after update (8) (when both flows are B) that the bottleneck rate at link l_{30} for flow 2 increases from b=10Gb/s to b=15Gb/s. (9) During update (9), b=15Gb/s exceeds e=12Gb/s and flow 2 is correctly classified into E at link l_{30}. We say the flow 2 has converged at this point, since we can show that it is forever marked B at its bottleneck link l_{12}, E at link l_{30} and allocated exactly its max-min fair rate 12Gb/s at both links.

We skip the discussion of flow 1, though interested readers may refer to table 4 for details.

6.2 Convergence of the s-PERC algorithm

As the previous example demonstrates, an s-PERC link may calculate a bottleneck rate that is too low. If propagated, the bottleneck rate would confuse its neighbors and potentially prevent convergence. The following important property of s-PERC, proved in Appendix C.1, allows it to converge in a bounded time where n-PERC cannot:

Lemma 5 (Good Rate Propagation). *The bottleneck rate* **propagated** *by a link is at least* $\mathbf{c}/\mathbf{n}$*, the fair share rate of the link, irrespective of the limit rates of the flows seen at the link.*

This property allows us to show that flows removed in the first-iteration of 2-Waterfilling will converge immediately at all their links (within a bounded number of rounds).

To see why flows removed in the first-iteration of 2-Waterfilling will converge immediately, recall that we define a flow to have converged to the correct rate when (and only when) it has been allocated the correct max-min fair rate at all of its links, and it has been correctly classified into sets B or E.

Let's see how this happens by considering one flow, f, that is removed in the first-iteration of 2-Waterfilling because it is bottlenecked at link k. Let l be some other link on flow f's path which has a higher fair share rate than link k (see Figure 5). Given that link k has the lowest fair-share rate of its second-degree neighbors, and any link on the path of a flow carried by link l is, by definition, a second-degree neighbor of link k, then the limit rates of all flows at link k and link l must be at least $\mathbf{c}_k/\mathbf{n}_k$.

As we will show, this means flow f is allocated its max-min fair rate (the fair share rate of link k) at **both** links, and is classified correctly into B at link k and E at link l.

Given a lower bound $\mathbf{c}_k/\mathbf{n}_k$ on the limit rates of all flows at links k and l, we can show that the bottleneck rate at link k equals $\mathbf{c}_k/\mathbf{n}_k$, and the bottleneck rate at link l strictly exceeds $\mathbf{c}_k/\mathbf{n}_k$. This follows from a straightforward property of the bottleneck rate calculation in s-PERC (and n-PERC) proven in Appendix C.2:

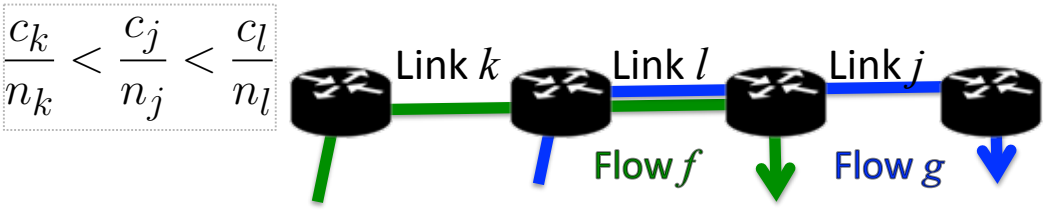


Fig. 5. Links k and j share flows f and g respectively with link l.

Lemma 6 (Local Bottleneck Rate Property). *Suppose limit rates at a link are at least the link's fair share rate,* $\mathbf{c}/\mathbf{n}$*, from time* T*, then the bottleneck rate computed by the link equals* $\mathbf{c}/\mathbf{n}$ *by time* $T + 2 \cdot round$*. If on the other hand, limit rates are bounded below by a smaller value* $e_{min} < \mathbf{c}/\mathbf{n}$*, the bottleneck rate strictly exceeds* e_{min} *by time* $T + 2 \cdot round$*.*[3]

It then follows that flow f, which is removed in the first-iteration along with link k, is allocated its max-min fair rate and classified correctly at both its links. This is because at link k, its limit rate is *at least* the link's fair share rate, and link k's computed bottleneck rate *equals* the fair share rate. So the flow is allocated link k's fair share rate and classified into B at link k. The bottleneck rate from link k is propagated to link l because like f, all flows at link k are classified into B, and the value of $MaxE$ at link k eventually drops to 0 (so $MaxE <$ B). Flow f's limit rate at link l equals link k's fair share rate, and the bottleneck rate computed by link l strictly *exceeds* link k's fair share rate. So the flow is allocated link k's fair share rate and correctly classified into E at link l.

Our proof in Appendix C.4 shows that in subsequent iterations of 2-Waterfilling, the remaining flows and links are correctly classified in turn, and their max-min fair share rates are correctly allocated.

Why 2-Waterfilling? Notice that in order for a flow's sending rate to converge to the max-min fair rate, the flow must be allocated its max-min fair rate at *all* links, not just the bottleneck link. This observation is key to understanding why it is natural to analyze *Fair* using 1-Waterfilling and s-PERC using 2-Waterfilling.

Consider Figure 5 again. Links k and j both share a flow with link l. Link k has the lowest fair share rate followed by link j and then link l; i.e. $c_k/n_k < c_j/n_j < c_l/n_l$. Links k and j are both removed in the first iteration of 1-Waterfilling, but only link k is removed in the first iteration of 2-Waterfilling.

In the *Fair* algorithm, flows f and g can converge in parallel because links j and k have the lowest fair share rate of the links their flows cross (their first-degree neighbors); i.e. $c_j/n_j < c_l/n_l$ and $c_k/n_k < c_l/n_l$. The relevant local property of *Fair* is that links compute a bottleneck rate that is locally max-min fair, and trivially at least the link's fair share rate, $\mathbf{c}/\mathbf{n}$. Link l computes a bottleneck rate (the limit rate for flows f and g) which exceeds the fair share rate of links k and j, allowing them to correctly compute a (locally max-min fair) bottleneck rate equal to their fair share rate, and allocate this to their flows. Flows f and g are also allocated the correct max-min fair rate by link l because the local bottleneck rate that link l computes for them is higher, and link l notices that links k and j give the flows a lower limit rate.

In contrast, s-PERC requires flow f bottlenecked at link k to converge *before* flow g bottlenecked at link j, because link k has the lowest fair share rate of its second-degree neighbors. While flows f

[3]Variable $round$ is the duration of a round in seconds. Thus $T + 2 \cdot round$ refers to the time two rounds after T.

and g are allocated their max-min fair rate at their bottleneck links in parallel (flow f at link k, flow g at link j), flow f is allocated its max-min fair rate at the non-bottleneck link l before flow g.

Link l has a higher fair share rate than its first-degree neighbor links k and j, and therefore link l propagates a bottleneck rate to its neighbors that is higher than their fair share rates (lemma 5). Therefore, links k and j compute a bottleneck rate that is equal to their fair share rate (lemma 6), allocate this rate to flows f and g and classify the flows into B.

But link j's flow cannot converge before link k's flow. For flow g to converge, link l must also allocate the correct rate to the flow (and classify the flow into E). Flow g's limit rate at link l already reflects the correct rate it picks up from link j. Now link l only needs to compute a bottleneck rate that is *higher*, in order to allocate the correct rate to the flow.

But link l may compute a bottleneck rate that is *lower*, and wrongly conclude that g is bottlenecked at link l (classifying it into B). If we can guarantee that link l's first-degree neighbors (like link k) have fair share rates greater than (or equal to) link j, we can prove (using lemma 5) that the limit rates of all flows at link l are at least link j's fair share rate, and (using lemma 6) that link l computes a bottleneck rate for flow g that is strictly greater. Requiring that both link l and link l's first-degree neighbors have fair share rates that are at least link j's fair share rate is the same as saying that link j has the lowest fair share rate of its second-degree neighbors. This is exactly why flow f can converge immediately—it is bottlenecked at link k, which has the lowest fair-share rate of first- and second-degree neighbors.

Note that we have not rigorously ruled out the possibility of using 1-Waterfilling to analyze s-PERC. We have only proven that 2-Waterfilling is sufficient to find a bound on the convergence time of s-PERC.

7 MAKING S-PERC PRACTICAL

The basic s-PERC algorithm works well in simulation, but to make sure it is practical we built a hardware prototype and collected measurements from it. This forced us to answer several questions along the way, such as what happens if control packets get lost? How precise do the rate calculations need to be for them to converge? And so on. We answer these questions and use them to design a practical version of s-PERC that we call s-PERC* to distinguish it from the base algorithm.

7.1 Design and Implementation of s-PERC*

Any practical implementation of s-PERC needs to address the following questions; we list our s-PERC* design choices as guidance.

Signaling the start and end of flows: The end host signals a new flow by sending a control packet initialized with $\mathbf{i}$=0, $\mathbf{s}$=E, $\mathbf{x}$=0, and $\mathbf{b}$=∞ (line 1 of algorithm 4). To signal the end of a flow the end host tags the control packet with a *FIN*.

Limiting control overhead: We want control packets to zip through the network in order to calculate rates quickly. However, we do not want the control packets to take too much bandwidth away from the data packets. Hence, control packets are prioritized at every link, but rate-limited to a fraction (2%-5%) of the link capacity.

Optimizing latency for short flows: s-PERC* starts short flows (< 1 BDP) at line rate and prioritizes them at every link (with priority second to control packets, for a total of three priority levels for all traffic). In §8.2.2 we evaluate this optimization for short flows in heavy tail workloads (e.g., web search, data mining). Since short flows start immediately at line-rate and finish within the same RTT, we do not send control packets to reserve bandwidth for them. We have to be careful about the remaining flows though: when they are about to end, they too may not have enough bytes to send for a full RTT. To avoid allocating more bandwidth than necessary, an s-PERC* control

packet carries a bottleneck rate field $\mathbf{b}_{l_0}$ for a *virtual link* l_0 (unique to each flow) on the flow's path, with the value remaining bytes/RTT.

Estimating *round*: Links must reset $MaxE$ and $MaxE'$ once per *round*, which is determined locally by each link, depending on how often control packets are seen. *Round* should not be too small; if $MaxE$ is reset prematurely, the link may wrongly propagate a bottleneck rate and delay convergence. Control packets carry the longest period (RTT) the end host has seen on the flow's path, Tx_{ctrl}; links use this field to set the local value of *round*.

Handling dropped control packets: The source end-host detects dropped control packets using a timeout, and restarts the flow from the next unacknowledged data packet. The links will continue to allocate the same rate to the flow, which is OK for a dropped control packet, but not OK if the flow has finished. s-PERC* links gradually phase out the old state from inactive flows using shadow variables $NumB'$ and $SumE'$ to re-compute the latest values of $NumB$ and $SumE$ on the side and sync up with the actual values every round (see algorithms 6 and 7 in Appendix D). To avoid double-counting a flow in one round, control packets carry the round number in a per-link field (the number of resets) so that a link can recognize if it has already considered it this round.

Headroom: A queue can briefly build up, when a flow's end-host is asked to increase its rate but before other flows that share the bottleneck link have decreased their rates. s-PERC* leaves a small fraction (e.g., 1%-2%) of the capacity unallocated at each link as headroom, to allow transient queues to drain quickly.

Control packet size: The s-PERC* control packet has five per-link fields: the allocation $\mathbf{x}$, bottleneck state $\mathbf{s}$, rate $\mathbf{b}$, the ignore bit, and the *round*. Together with the flow size information, the FIN tag and the control-packet period Tx_{ctrl}, a control packet in a two-level, four hop network has 46 bytes (table 9 in Appendix D). The control packet state could be reduced using techniques in [37], but we have not done so.

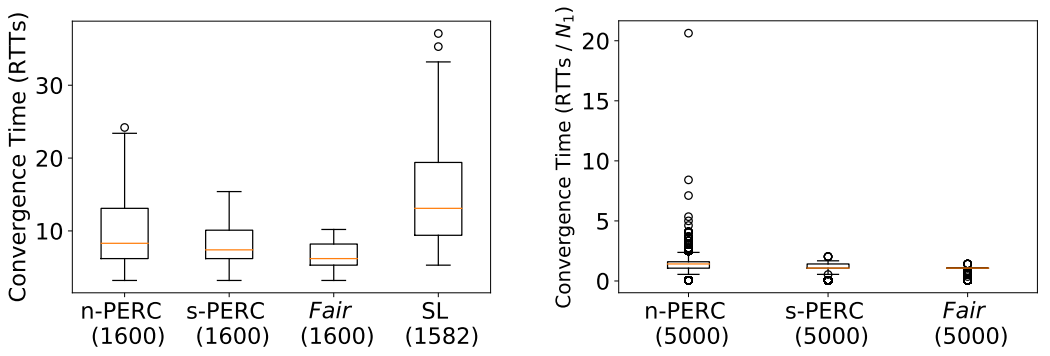
Division in hardware: The switch needs to perform the *division* $b = (\mathbf{c} - SumE)/NumB$ once per control packet to propagate the bottleneck rate and update the link state ($SumE$, $NumB$). This is hard to do at line-rate, so in our hardware prototype we break it into two parts. First we atomically update the link state in one clock cycle based on the comparison $(\mathbf{c} - SumE) < e * NumB$, which only requires a much simpler integer multiplication. The second step is to calculate (and propagate) the new bottleneck rate, which requires a division, but fortunately can be pipelined over multiple clock cycles. We use the common technique of approximating division using a lookup table [45]. Switch ASICs already have hundreds of megabytes of lookup tables, and it is reasonable to assume we could use a small fraction (say, 1%) for a division lookup table. Our prototype uses this approach and we found it works well with small tables; our results measured from our hardware prototype use an 84KB division lookup table (32Kb of exact match; and 2048 TCAM entries, where each entry has a 32b key and 10b result.)

8 EVALUATION

In this section we evaluate s-PERC with numerical simulations (§8.1), packet-level simulations (§8.2), and a hardware prototype (§8.3).

8.1 Numerical Simulations

8.1.1 Convergence time. We simulated different PERC algorithms for more than 6,000 different routing matrices, to compare the convergence times of the different schemes in practice to the worst-case bounds from theory. We compared *Fair*, n-PERC and s-PERC, and a fourth algorithm SL [46], which uses per-flow state to calculate bottleneck rates differently from *Fair*. In SL each link stores the limit rate *and* bottleneck state for every flow. While SL does not have a proof of



(a) 1600 sparse routing matrices

(b) 5000 dense routing matrices

Fig. 6. Convergence times of different PERC algorithms with random routing matrices. s-PERC: 1 round=1 RTT.

convergence, it modifies Charny *et al.* [13], which was shown to converge in $4N_\infty$ rounds (N_∞ is the number of iterations that standard waterfilling [11] takes to converge).

Setup: For each simulation run, we randomly generate a routing matrix with M links and N flows, where each flow crosses P randomly picked links. For *sparse* networks (where flows typically cross only a few links) we ran 200 simulations for each of the following 16 settings: M=100 $\times$ $N\in\{10^2, 10^3, 10^4\} \times P\in\{5, 10\}$ and M=1000 $\times$ N=1000 $\times$ $P\in\{5, 10\}$. All flows start at once and see a random delay d at each link to evaluate arbitrary packet orderings.

We also simulated several thousand dense routing matrices (M=100, N=100, P=80), which have only a few bottleneck links.

Results: Figure 6a shows a summary of convergence times for sparse routing matrices. The median convergence time of *Fair*, s-PERC, n-PERC are similar while SL takes longer (6, 7, 8 and 13 RTTs respectively.) At the tail (95th percentile), *Fair* and s-PERC have similar convergence times at 9 and 12 RTTs, while n-PERC and SL are 2–3 times slower than *Fair* at 18 and 26 RTTs. SL failed to converge for 18 runs out of 1600. For the dense routing matrices (Figure 6b), the maximum convergence time (normalized by N_1) is 1.4 RTTs for *Fair*, 2.0 RTTs for s-PERC and 20.6 RTTs for n-PERC. The median convergence times are comparable: 1.1 RTTs for s-PERC and *Fair* and 1.4 RTTs for n-PERC. SL (not shown) failed to converge for 72 runs, and took as long as 50-100 RTTs for at least two runs out of 5000. **Worst-case convergence times:** We verified that s-PERC takes no more than $6N_2$ RTTs to converge. For n-PERC, while we didn't find a simulation that didn't converge, we found that for some runs where the routing matrix is dense and there are only a few bottleneck links, convergence can take long. The worst case we observed was 42 RTTs ($21N_2$ RTTs) for a dense routing matrix with four bottleneck links (and N_1=N_2=N_∞=2.) For the same example, s-PERC took 3 RTTs.

8.1.2 Robustness. s-PERC needs to be robust to control packet drops and imprecise rate calculations in hardware. To evaluate s-PERC's robustness, we enhance the simulator in the previous section to model random control packet drops and approximate division based on lookup-tables, as described below. We used a subset (10%) of the sparse routing matrices for these simulations.

Control packet drops: s-PERC* can recover from control packet drops by recomputing the aggregate state using shadow variables, which are synced with *NumB* and *SumE* every round (§7).

We test robustness to control packet drops as follows: Given a routing matrix, we start all flows at once, and run the simulation for 100 RTTs. During the first 20 RTTs, control packets are dropped at every link with probability $p = 1\%$ or $p = 0.1\%$. Over the next 80 RTTs, no more control packets are dropped and we watch as the algorithm converges again. We found that in every run, all flows converged again quickly (median 10 RTTs) to their max-min fair allocation after the last packet drop.

Imprecise rate calculations: Our NetFPGA hardware prototype (§8.3) uses a lookup table for division instead of a floating point divider. In numerical simulation we tried different table sizes (20, 80, 320 KB) and observed that as the table sizes get smaller the flows stabilize to values that are farther from the optimal or have wider oscillations around the optimal. We found that a table size of 320kB was sufficient for all the workloads to converge to within 10% of the optimal rates, and in the same convergence time. A 320kB table is easily accommodated in today's switch hardware, which have hundreds of megabytes of lookup tables.

8.2 Packet-level Simulations

We used ns-2[4] packet-level simulations to evaluate s-PERC in datacenter and wide-area networks and for realistic workloads.

8.2.1 Convergence Times. We compared proactive s-PERC* with the reactive RCP (Rate Control Protocol) [17] to see how fast they converge for long-lived flows. We use RCP as our representative reactive algorithm because it is max-min fair and was designed to converge faster than TCP [47]. In RCP, every link computes a fair share rate every RTT, and advertises the rate to every flow that shares the link. A flow is then transmitted at the smallest advertised fair-share rate along its path. We say that RCP is *reactive* because the fair share rate calculation is based on measuring and reacting to the rate at which traffic arrives and the queue builds up at the link. **Setup:** To represent a data center network, we simulate a 144-server 3-level fat-tree topology [4] with 100 and 400 Gb/s links. We start flows between random pairs of servers, with N=20 flows per server on average.[5] To represent a WAN network we use Google's B4 inter–data-center network topology [30] with 1Gb/s and 10Gb/s links and N=100 flows per server.

For both the DC and WAN network we evaluate how s-PERC and RCP converge with a dynamic workload: we wait for the rates to converge, then replace 40% of the flows with new ones. We replace all flows at once to create a big transient jolt, to see how robust they are and how long they take to converge again. Figures 7 and 8 show CDFs of convergence times for 1,000 tests (ten changes per run, and 100 runs[6]) for the DC and WAN network respectively. Table 5 shows the average time it took for flows to stabilize at the beginning of each run in the WAN setting (100 runs, except for RCP at 10Gb/s, where we used 200 runs).

Metrics: We say that the flow rates have converged when most of the flows remain within a small distance of the ideal max-min values for at least N consecutive RTTs. We look at different degrees of accuracy, for example when 95% of flows are within 20% of ideal, or when 99% of flows are within 10% of ideal.

Results: In the data-center setting (Figure 7), s-PERC* converges at least ten times faster than RCP at the median and tail. s-PERC* gets within 10% of the ideal rates for at least 99% of flows in only 14 RTTs while RCP takes 174 RTTs. It only takes s-PERC* 4 RTTs (median) to get within 20% of the ideal rates for at least 95% of flows, while RCP needs 45 RTTs. In the WAN setting (Figure 8),

[4]The Network Simulator (ns-2), https://www.isi.edu/nsnam/ns/

[5]We chose 20 flows because it approximates the number of simultaneous active flows we observed on congested links in simulations of realistic workloads (search and data mining) using Poisson arrivals at 80% load (FCT experiments in §8.2.2).

[6]for RCP in WAN at 10Gb/s, we simulated five changes per run for 200 runs

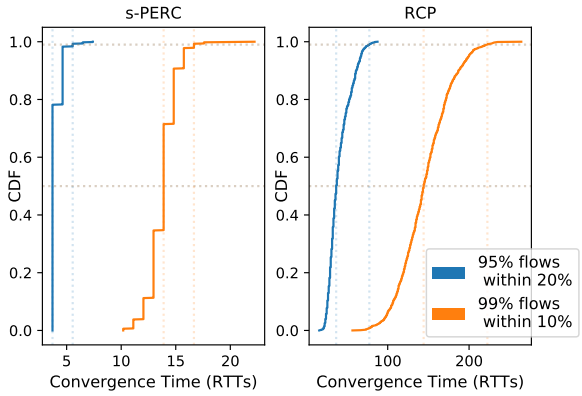


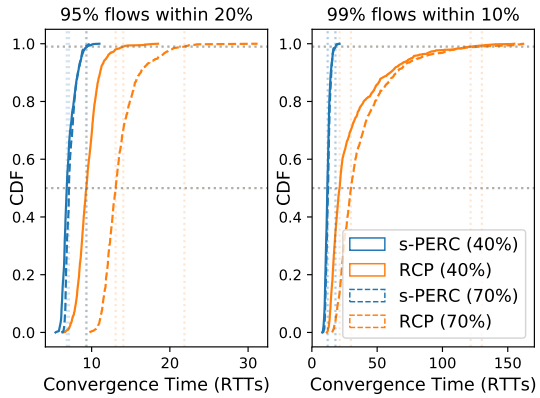
Fig. 7. CDF of convergence times for for RCP and s-PERC* in the data-center network. Note difference in x-axis scales. Settings in Table 6.

convergence times and distributions for both link speeds are similar. When 40% of the flows are changed at once, s-PERC* is 1.3–1.7 times faster than RCP at the median, and 1.5–6 times faster at the tail, depending on the metric. When 70% of the flows are changed at once (Figure 8a), s-PERC* is 1.8–2.5 times faster at the median, and 2.4–7.2 times faster at the tail. Table 5 shows that when all flows start at once, s-PERC* converges about five times faster (on average) than RCP for the WAN topology, for both link rates.

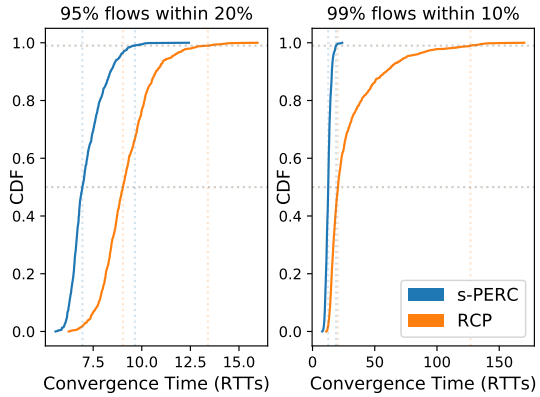
Discussion: As expected, s-PERC* converges much faster than RCP in all our experiments. How much faster depends on the kind of network (DC or WAN), and the magnitude of the transient jolt from new flows. Let's consider each factor in turn. Comparing the DC and WAN, we see that the median convergence time of s-PERC is 10 times faster than RCP in the DC, but only 30–70% faster in the WAN. We see two reasons for this: (1) s-PERC benefits greatly from short RTTs in the DC, with little variance. This is because s-PERC's convergence is bounded by a number of rounds which must be set to the longest RTT. In the WAN, RTTs vary by two orders of magnitude, and round must be set to the largest RTT which is over 100ms. (2) s-PERC is further slowed in the WAN because control packets are only updated in the forward direction, increasing convergence time by a factor of two. On the other hand, RCP's convergence time in the WAN depends on the average (not the max) flow RTT with each flow operating independently, allowing RCP to close the gap slightly in the WAN. Note that increasing the data-rate in the WAN does not improve convergence time. This is because in the DC RTTs are dominated by serialization delay, which are reduced by faster links. But in the WAN, RTTs are dominated by propagation delay which is unaffected by data-rate.

Next, we consider how the algorithms react to sudden changes in traffic. The biggest change is when the simulation starts and all flows start at the same time. Here, s-PERC converges five times faster on average (Table 5). When we change 40% of the flows during a run, s-PERC is 1.3–1.7 times faster than RCP. When we change 70% of the flows, s-PERC is 1.8–2.5 times faster than RCP. This is because the convergence time of s-PERC is a function of the traffic matrices only, whereas RCP has to converge cautiously so as to remain stable in the face of large transient jolts.

Finally, we note that not only does s-PERC converge more quickly, the distribution of convergence times is narrower, with much better tail latency performance than RCP. Figure 8b shows that while



(a) WAN convergence time experiments for 1Gb/s links. Solid lines are for experiments in which 40% of the flows are replaced at once, while dashed lines correspond to replacing 70% of the flows at once. For 40% change 95% of s-PERC flows converge within 20% of max-min in 7 RTTs at median and 9 RTTs for 99%-ile tail. RCP takes 9 and 14 RTTs. For 99% flows to converge within 10% of max-min, s-PERC takes 12 and 18 RTTs at median and tail, while RCP takes 21 and 115 RTTs respectively.



(b) WAN convergence time experiments for 10Gb/s links. For 95% flows to converge within 20% of max-min, s-PERC takes 7 RTTs at median and 9 RTTs for 99%-ile tail. RCP takes 9 and 13 RTTs. For 99% flows to converge within 10% of max-min, s-PERC takes 13 and 19 RTTs at median and tail, while RCP takes 20 and 114 RTTs respectively.

Fig. 8. CDF of convergence times following each change (40% flows replaced at once) for RCP and s-PERC* in the WAN network at 1Gb/s and 10Gb/s. Convergence time is in terms of the average RTT of all flows seen during the run, on average 70ms. Dotted lines in the background indicate 50th and 99th percentiles for each scheme. Settings in Table 6.

Table 5. Comparison of convergence time (in RTTs) when all flows start at once. WAN topology with different link rates. Each entry is averaged over 100 runs, except for RCP at 10Gb/s, which used 200 (short) runs.

Link rate	95% flows, 20% acc.		99% flows, 10% acc.	
	RCP	**s-PERC**	**RCP**	**s-PERC**
1Gb/s	53	10	91	15
10Gb/s	50	10	91	16

Table 6. Settings for Convergence Time and FCT experiments (§8.2).

Algorithm	Buffer	Ctrl packet	Other settings
s-PERC*	128KB	64B	initial round=20μs (DC), 100ms (WAN), headroom=2%, ctrl rate=2% (FCT), 4% (CT)
RCP (DC)	128KB	40B	α=0.4, β=0.2, update interval=100ms
RCP (WAN)	256KB	40B	α=0.9, β=0.1, update interval=20ms
p-Fabric	360KB	n/a	initial cwnd=120, rto=45μs

RCP takes only 20 RTTs at the median to get 99% of the flows within 10% of max-min fair, at least 20% of the time RCP took more than 50 RTTs. s-PERC flows converge quickly, with little variance.

8.2.2 Flow Completion Times. We have seen that s-PERC converges to fair rates an order of magnitude faster than reactive algorithms. While cloud providers care about fairness [30, 35], users care about application level metrics like the FCT, which captures latency for short flows and throughput for large flows. Both fairness and application level metrics are important for s-PERC if it is to be deployed in a real datacenter or WAN. Next, we see how s-PERC affects flow completion time.

Workloads: We simulated two workloads, one for search, the other for data mining with loads of 60% and 80%. We assume Poisson flow arrivals and flow size distributions from prior measurements of actual workloads [4]. Both workloads are heavy-tailed, with few large flows carrying most of the bytes.

Metrics: We measure the flow completion times (FCTs) then normalize them to the time it would take to transmit the flow in an otherwise empty network. We bin flows according to their size and the fraction of total bytes they carry. The first bin contains the smallest flows that contribute 1% of the total bytes. Most flows are small and belong to the first bin. The remaining bins contain equal fractions (by bytes contributed) of the remaining flows.

Algorithms: We compare RCP, s-PERC*, p-Fabric [4], and an ideal max-min allocator. RCP and s-PERC* are fast rate-allocation schemes that target max-min fairness; the ideal max-min allocator serves as a reference for both. p-Fabric is a popular DC congestion control scheme that aims to emulate SRPT (shortest remaining processing time); switches schedule packets in increasing order of remaining flow size. We evaluate two versions of s-PERC*. The basic version does not optimize latency for short flows; flows must wait for the rate algorithm to run before sending any packets and all flows get the same priority at the links. s-PERC* 'short' starts short flows at line rate and prioritizes them at the links.

Results: Figure 9 compares FCTs for *search* at 60% load. For example, the max-min fair schemes (RCP, basic s-PERC* and max-min allocator) are almost identical for long flows (bottom right graph), as expected. For very short flows (top left) basic s-PERC* flows complete 25-50% sooner than RCP, because s-PERC* queues are kept shorter.[7] With s-PERC* 'short', the smallest flows start at line rate and see no queues (because they are high priority); hence they complete in a quarter of

[7] s-PERC* and RCP start at x=2, because of the extra RTT it takes to get a starting rate.

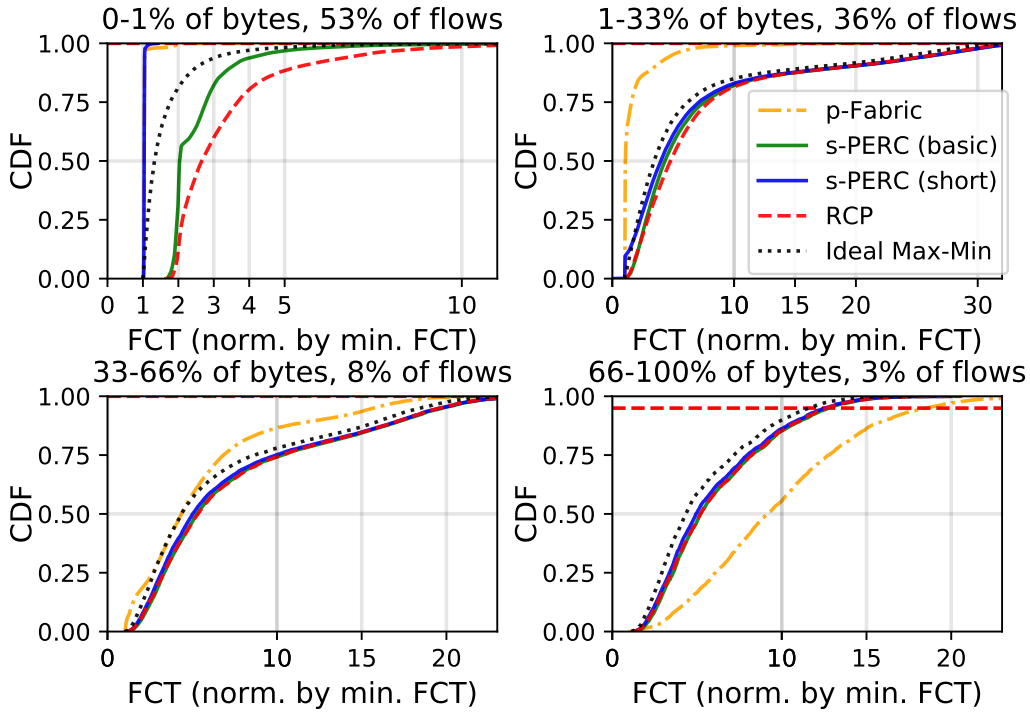


Fig. 9. FCTs for search workload at 60% load. Settings in Table 6. Note bin 4 (lower right) has fewer than 30 samples for the 95th percentile and above (horizontal red line).

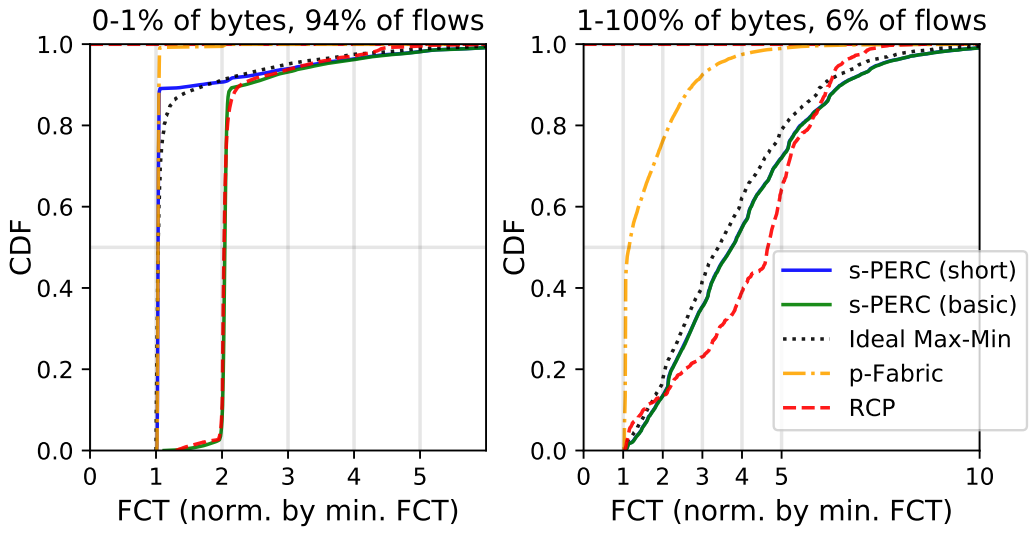


Fig. 10. FCTs for data-mining workload at 60% load. Settings in Table 6.

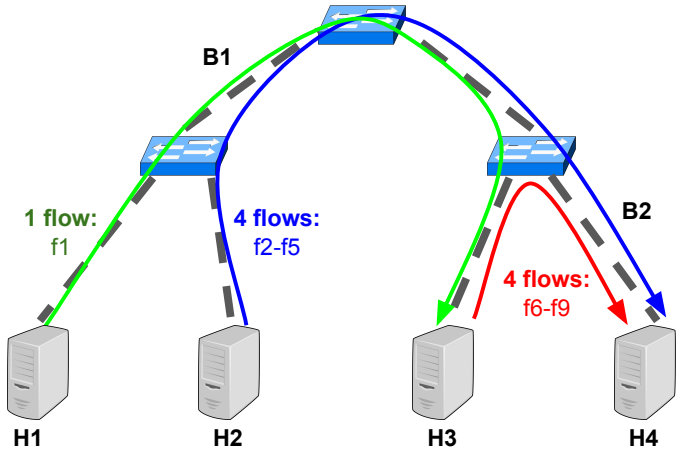


Fig. 11. The topology and traffic pattern used for the two-level dependency chain experiment.

the time as the basic s-PERC*, confirming that the short-flow optimization is effective for search and data mining workloads.

pFabric shines for short flows because it emulates greedy SRPT, which is known to achieve close to optimal *average* FCTs [4, 9]. This comes at the expense of the largest flows; the largest flows get almost 200% worse throughput. For the smallest flows, optimized s-PERC* is slightly better than p-Fabric at the tail, and both are close to the minimum possible at the median.

Figure 10 compares FCTs with the *data-mining workload* at 60% load. pFabric shines again for very short flows, while s-PERC* 'short' tracks ideal max-min very closely. Results at 80% load are qualitatively similar (Figures 12, 13 in Appendix D).

8.3 Hardware Prototype and Evaluation

To confirm that s-PERC* is practical in hardware at high link speeds, we implemented the s-PERC* switch on the 4x10Gb/s NetFPGA SUME platform [49] with a 200MHz hardware clock. The end host uses the MoonGen DPDK packet processing library [18]. Our hardware testbed network with three NetFPGA switches connecting four servers at 10Gb/s is shown in Figure 11. We used it to compare TCP Reno, DCTCP (ECN bits set by the switches), and s-PERC.

We ran two experiments to compare the convergence time of the three algorithms:

Incast: Two senders transmit to one receiver causing congestion at the switch; the bottleneck is the 10Gb/s link at the receiver. Figure 1 compares the convergence behavior of TCP, DCTCP, and s-PERC. TCP generally takes longer than DCTCP for the flow rates to stabilize, and s-PERC converges about an order of magnitude faster than TCP and DCTCP.

Dependency chain: In this experiment we created a *dependency chain* between two links. The green and blue flows start first, with the tightest bottleneck at link B1. When we add the red flows the bottleneck moves to link B2 and the green flow can increase its rate. For this traffic pattern the 2-Waterfilling algorithm takes two iterations to compute the max-min fair rates, eliminating B2 first, then B1. In this experiment, s-PERC converges in 1.56 ± 0.23 ms, and DCTCP converges in 65 ± 30 ms. TCP failed to converge at all in this experiment. The performance of s-PERC is dictated by the timeout value used by the switch to reset the *MaxE* state, which must exceed the control packet RTT. According to Theorem 4, in a situation where 2-Waterfilling takes two iterations, and the maximum RTT is 1 ms, s-PERC's convergence time is upper-bounded by 12 ms. Hence even at

the upper bound, s-PERC converges more than five times faster than DCTCP. We expect s-PERC to converge even faster if control packets are prioritized by the NIC.

9 RELATED WORK

Some *reactive* algorithms have been shown to converge to max-min fair [1, 32, 47]. Some *proactive* algorithms have been shown to converge asymptotically [15, 19, 22, 25, 36], but only a few have been proven to compute exact max-min fair rates in a known finite time [2, 10, 12, 24, 38, 43]. But these are not practical because they require per-flow state at the switches or require updates to be synchronized [2, 20]. Some distributed algorithms have been proven to converge to approximate max-min fair rates [7, 34], while others target different but related problems like α-fairness for finite values of α [33] or the concurrent multi-commodity flow problem [6]. Note that although the algorithms proposed by Marasevic *et al.* [33] and Awerbuch *et al.* [6] are "stateless", they assume a model [6] where agents (servers or flows) have local clocks that are synchronized. This is more restrictive than our model, which like previous work (mostly for ATM networks) [10, 12, 24, 38, 43], does not require clocks to be synchronized. Some algorithms converge fast in simulations but have not been proven to converge to max-min fair rates in a general multiple bottleneck link setting [27, 29, 37, 46]. Our scheme builds upon Bartal's ABR algorithm [10], which addresses the problem of artificially low bottleneck rates by propagating the maximum of the bottleneck rate and MAXSAT, the maximum allocation of a satisfied (E) flow. Note that their proof of convergence defines and uses *MAXSAT* as the 'maximum satisfied allocation' which requires per-flow state to maintain. We use assumptions of bounded RTTs for control packets to do away with the per-flow state requirement without sacrificing stability or provably fast convergence. The 1-Waterfilling algorithm was introduced by Ros-Giralt *et al.* [43] to analyze the convergence behavior of d-CPG (similar to *Fair*), which requires per-flow state. We generalize this algorithm to the family of algorithms called k-Waterfilling and show that we can use the 2-Waterfilling algorithm to analyze the convergence behavior of s-PERC.

For data centers: pFabric [4] and PIAS[8] leverage priority queues in switches to approximate SRPT or SJF. We borrow this idea for short flows in our practical implementation of s-PERC. PDQ [23] and D3 [48] are proactive congestion control algorithms, although their goal is to minimize mean FCT and the number of missed deadlines by dynamically prioritizing some flows over others. PDQ [23] is particularly interesting as it can emulate centralized scheduling algorithms such as shortest-job-first and earliest-deadline-first. PDQ requires some per-flow state for the top active flows, and in a general setting this state could be very large. PASE [39] is a hybrid scheme, where flows get an initial rate from a software controller at the TOR switch and then switch to a reactive scheme. NDP [21] is a receiver-driven transport protocol, where the sender's flow rate is dictated by the receiver, based on the sender's demand as well as the over-subscription at the receiver. s-PERC can complement NDP by providing explicit flow rates that also take into consideration hot spots *within* the network, not just at the edge. ExpressPass [14] is a credit-based congestion control scheme that uses credit packets that traverse the same path as the data traffic, to aim for fast convergence. The difference is that s-PERC control packets are used by the network to calculate explicit rates directly, while ExpressPass credit packets must themselves go through an aggressive reactive control loop before converging to credit rates that the data traffic can use.

10 CONCLUSION

The problem we are addressing is quite alarming. Pretty much every network today uses reactive congestion control algorithms that converge timidly and more slowly than their owners realize. This is bad news when the network is congested: For a (long) time before rates converge, flows can operate far from their fair share rate, links are either under-used or overloaded, and buffers

can be much fuller than necessary, increasing delay and dropping packets (and hence restarting the algorithm) unnecessarily. Many flows complete long before the algorithm has converged. The behavior of reactive algorithms will get worse as future networks get faster.

s-PERC points to the intriguing potential of an order of magnitude tighter and faster control over how our links are shared among flows, and it is the first to do so with provable bounds on convergence. Of course, much more work is required before concluding that s-PERC should be deployed widely, for example it requires adoption of switches that natively (or can be programmed to) support control packet processing. But once PERC algorithms can be deployed, one can imagine more sophisticated PERC algorithms that optimize over more traffic types and sharing policies. We think of s-PERC as a step towards this possibility.

ACKNOWLEDGMENTS

We thank our shepherd, Kostya Avrachenkov and the anonymous SIGMETRICS reviewers for their valuable comments. We thank Radhika Mittal, Srinivas Narayana, Sundar Iyer, Manikanta Kotaru, and Renata Teixeira for their feedback on early drafts of this paper. This work was supported by Intel Corporation and the Stanford Platform Lab.

REFERENCES

[1] Yehuda Afek, Yishay Mansour, and Zvi Ostfeld. 1996. Phantom: A Simple and Effective Flow Control Scheme. In *Conference Proceedings on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '96)*. ACM, New York, NY, USA, 169–182. https://doi.org/10.1145/248156.248172

[2] Yehuda Afek, Yishay Mansour, and Zvi Ostfeld. 1999. Convergence Complexity of Optimistic Rate-Based Flow-Control Algorithms. *Journal of Algorithms* 30, 1 (Jan. 1999), 106–143. https://doi.org/10.1006/jagm.1998.0970

[3] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data Center TCP (DCTCP). In *Proceedings of the SIGCOMM 2010 Conference (SIGCOMM '10)*. ACM, New York, NY, USA, 63–74. https://doi.org/10.1145/1851182.1851192

[4] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. 2013. pFabric: Minimal Near-optimal Datacenter Transport. In *Proceedings of the ACM SIGCOMM 2013 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '13)*. ACM, New York, NY, USA, 435–446. https://doi.org/10.1145/2486001.2486031

[5] Noga Alon, Yossi Matias, and Mario Szegedy. 1999. The Space Complexity of Approximating the Frequency Moments. *J. Comput. System Sci.* 58, 1 (Feb. 1999), 137–147. https://doi.org/10.1006/jcss.1997.1545

[6] Baruch Awerbuch and Rohit Khandekar. 2007. Greedy Distributed Optimization of Multi-commodity Flows. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing (PODC '07)*. ACM, New York, NY, USA, 274–283. https://doi.org/10.1145/1281100.1281140

[7] B. Awerbuch and Y. Shavitt. 1998. Converging to approximated max-min flow fairness in logarithmic time. In *Proceedings. IEEE INFOCOM '98, the Conference on Computer Communications. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Gateway to the 21st Century (Cat. No.98*, Vol. 3. 1350–1357 vol.3. https://doi.org/10.1109/INFCOM.1998.662951

[8] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. 2017. PIAS: Practical Information-Agnostic Flow Scheduling for Commodity Data Centers. *IEEE/ACM Transactions on Networking* 25, 4 (Aug. 2017), 1954–1967. https://doi.org/10.1109/TNET.2017.2669216

[9] Amotz Bar-Noy, Magnús M Halldórsson, Guy Kortsarz, Ravit Salman, and Hadas Shachnai. 2000. Sum multicoloring of graphs. *Journal of Algorithms* 37, 2 (2000), 422–450.

[10] Yair Bartal, Martin Farach-Colton, Shibu Yooseph, and Lisa Zhang. 2002. Fast, fair and frugal bandwidth allocation in atm networks. *Algorithmica* 33, 3 (2002), 272–286.

[11] Dimitri Bertsekas and Robert Gallager. 1987. *Data Networks*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[12] Anna Charny, David D Clark, and Raj Jain. 1995. Congestion control with explicit rate indication. In *Communications, 1995. ICC'95 Seattle,'Gateway to Globalization', 1995 IEEE International Conference on*, Vol. 3. IEEE, 1954–1963.

[13] Anna Charny, KK Ramakrishnan, and Anthony Lauck. 1996. Time scale analysis scalability issues for explicit rate allocation in ATM networks. *IEEE/ACM Transactions on Networking* 4, 4 (1996), 569–581.

[14] Inho Cho, Keon Jang, and Dongsu Han. 2017. Credit-Scheduled Delay-Bounded Congestion Control for Datacenters. In *Proceedings of the 2017 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '17)*. ACM, New York, NY, USA, 239–252. https://doi.org/10.1145/3098822.3098840

[15] Jorge A. Cobb and Mohamed G. Gouda. 2011. Stabilization of Max-min Fair Networks Without Per-flow State. *Theoretical Computer Science* 412, 40 (Sept. 2011), 5562–5579. https://doi.org/10.1016/j.tcs.2010.11.042

[16] A. Demers, S. Keshav, and S. Shenker. 1989. Analysis and Simulation of a Fair Queueing Algorithm. In *Symposium Proceedings on Communications Architectures &Amp; Protocols (SIGCOMM '89)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/75246.75248

[17] Nandita Dukkipati. 2008. *Rate Control Protocol (Rcp): Congestion Control to Make Flows Complete Quickly*. Ph.D. Dissertation. Stanford University, Stanford, CA, USA. Advisor(s) Mckeown, Nick. AAI3292347.

[18] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. 2015. Moongen: A scriptable high-speed packet generator. In *Proceedings of the 2015 ACM Conference on Internet Measurement Conference*. ACM, 275–287.

[19] Eli Gafni and Dimitri Bertsekas. 1984. Dynamic control of session input rates in communication networks. *IEEE Trans. Automat. Control* 29, 11 (1984), 1009–1016.

[20] Dimitris Giannopoulos, Nikos Chrysos, Evangelos Mageiropoulos, Giannis Vardas, Leandros Tzanakis, and Manolis Katevenis. 2018. Accurate Congestion Control for RDMA Transfers. In *2018 Twelfth IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*. IEEE, 1–8.

[21] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W Moore, Gianni Antichi, and Marcin Wójcik. 2017. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the 2017 ACM Conference on Special Interest Group on Data Communication*. ACM, 29–42.

[22] Howard P Hayden. 1981. *Voice flow control in integrated packet networks*. Ph.D. Dissertation. Massachusetts Institute of Technology.

[23] Chi-Yao Hong, Matthew Caesar, and P. Brighten Godfrey. 2012. Finishing Flows Quickly with Preemptive Scheduling. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '12)*. ACM, New York, NY, USA, 127–138. https://doi.org/10.1145/2342356.2342389

[24] Y. Thomas Hou, Shivendra S. Panwar, and Henry H. Y. Tzeng. 2004. On Generalized Max-Min Rate Allocation and Distributed Convergence Algorithm for Packet Networks. *IEEE Trans. Parallel Distrib. Syst.* 15, 5 (May 2004), 401–416. https://doi.org/10.1109/TPDS.2004.1278098

[25] Jeffrey Jaffe. 1981. Bottleneck flow control. *IEEE Transactions on Communications* 29, 7 (1981), 954–962.

[26] Lavanya Jose, Lisa Yan, Mohammad Alizadeh, George Varghese, Nick McKeown, and Sachin Katti. 2015. High Speed Networks Need Proactive Congestion Control. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks (HotNets-XIV)*. ACM, New York, NY, USA, Article 14, 7 pages. https://doi.org/10.1145/2834050.2834096

[27] Shivkumar Kalyanaraman, Raj Jain, Sonia Fahmy, Rohit Goyal, and Bobby Vandalore. 2000. The ERICA switch algorithm for ABR traffic management in ATM networks. *IEEE/ACM Transactions on Networking* 8, 1 (2000), 87–98.

[28] Dina Katabi, Mark Handley, and Charlie Rohrs. 2002. Congestion Control for High Bandwidth-delay Product Networks. In *Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '02)*. ACM, New York, NY, USA, 89–102. https://doi.org/10.1145/633025.633035

[29] Yuseok Kim, Wei Kang Tsai, Mahadevan Iyer, and Jordi Ros-Giralt. 1999. Minimum rate guarantee without per-flow information. In *Network Protocols, 1999.(ICNP'99) Proceedings. Seventh International Conference on.* IEEE, 155–162.

[30] Alok Kumar, Sushant Jain, Uday Naik, Anand Raghuraman, Nikhil Kasinadhuni, Enrique Cauich Zermeno, C. Stephen Gunn, Jing Ai, Björn Carlin, Mihai Amarandei-Stavila, Mathieu Robin, Aspi Siganporia, Stephen Stuart, and Amin Vahdat. 2015. BwE: Flexible, Hierarchical Bandwidth Allocation for WAN Distributed Computing. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '15)*. ACM, New York, NY, USA, 1–14. https://doi.org/10.1145/2785956.2787478

[31] Jean-Yves Le Boudec. 2000. Rate adaptation, congestion control and fairness: A tutorial. (2000).

[32] Tae-Jin Lee and G. De Veciana. 1998. A decentralized framework to achieve max-min fair bandwidth allocation for ATM networks. In *IEEE GLOBECOM 1998 (Cat. NO. 98CH36250)*, Vol. 3. 1515–1520 vol.3. https://doi.org/10.1109/GLOCOM.1998.776608

[33] Jelena Marasevic, Cliff Stein, and Gil Zussman. 2015. A Fast Distributed Stateless Algorithm for alpha-Fair Packing Problems. *arXiv preprint arXiv:1502.03372* (2015).

[34] Alain Mayer, Yoram Ofek, and Moti Yung. 1996. Approximating max-min fair rates via distributed local scheduling with partial information. In *Proceedings of the Fifteenth annual joint conference of the IEEE computer and communications societies conference on The conference on computer communications (INFOCOMM'98)*, Vol. 2. IEEE, 928–936.

[35] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. 2015. TIMELY: RTT-based Congestion Control for the Datacenter. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '15)*. ACM, New York, NY, USA, 537–550. https://doi.org/10.1145/2785956.2787510

[36] Jeannine Mosely. 1984. *Asynchronous distributed flow control algorithms*. Ph.D. Dissertation. Massachusetts Institute of Technology.

[37] Alberto Mozo, Jose Luis López-Presa, and Antonio Fern'ndez Anta. 2012. SLBN: A Scalable Max-min Fair Algorithm for Rate-Based Explicit Congestion Control. In *Network Computing and Applications (NCA), 2012 11th IEEE International Symposium on*. IEEE, 212–219.

[38] Alberto Mozo, JosÃĺ Luis LÃşpez-Presa, and Antonio FernÃąndez Anta. 2018. A distributed and quiescent max-min fair algorithm for network congestion control. *Expert Systems with Applications* 91 (2018), 492 – 512. https://doi.org/10.1016/j.eswa.2017.09.015

[39] Ali Munir, Ghufran Baig, Syed M. Irteza, Ihsan A. Qazi, Alex X. Liu, and Fahad R. Dogar. 2014. Friends, Not Foes: Synthesizing Existing Transport Strategies for Data Center Networks. In *Proceedings of the 2014 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '14)*. ACM, New York, NY, USA, 491–502. https://doi.org/10.1145/2619239.2626305

[40] Abhay K Parekh and Robert G Gallager. 1993. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Transactions on Networking* 1, 3 (1993), 344–357.

[41] Jonathan Perry, Hari Balakrishnan, and Devavrat Shah. 2017. Flowtune: Flowlet Control for Datacenter Networks. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation (NSDI'17)*. USENIX Association, Berkeley, CA, USA, 421–435. http://dl.acm.org/citation.cfm?id=3154630.3154665

[42] Jordi Ros-Giralt. 2003. *A Theory of Lexicographic Optimization for Computer Networks*. Ph.D. Dissertation. University of California, Irvine.

[43] Jordi Ros-Giralt and Wei Kang Tsai. 2001. A theory of convergence order of maxmin rate allocation and an optimal protocol. In *Proceedings IEEE INFOCOM 2001 Conference on Computer Communications*. IEEE, 717–726.

[44] Jordi Ros-Giralt and Wei K Tsai. 2010. A lexicographic optimization framework to the flow control problem. *IEEE Transactions on Information Theory* 56, 6 (2010), 2875–2886.

[45] Naveen Kr. Sharma, Antoine Kaufmann, Thomas Anderson, Changhoon Kim, Arvind Krishnamurthy, Jacob Nelson, and Simon Peter. 2017. Evaluating the Power of Flexible Packet Processing for Network Resource Allocation. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation (NSDI'17)*. USENIX Association, Berkeley, CA, USA, 67–82. http://dl.acm.org/citation.cfm?id=3154630.3154637

[46] Fabian Skivée and Guy Leduc. 2004. A distributed algorithm for weighted max-min fairness in MPLS networks. In *International Conference on Telecommunications*. Springer, 644–653. build on hou to reduce rm cells and use faster update architecture to improve convergence time by a factor of 2 or 3 depending on precision, asynchronous, needs per-flow state, proactive.

[47] T. Voice and G. Raina. 2009. Stability Analysis of a Max-Min Fair Rate Control Protocol (RCP) in a Small Buffer Regime. *IEEE Trans. Automat. Control* 54, 8 (Aug 2009), 1908–1913. https://doi.org/10.1109/TAC.2009.2022115

[48] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. 2011. Better Never Than Late: Meeting Deadlines in Datacenter Networks. In *Proceedings of the ACM SIGCOMM 2011 Conference (SIGCOMM '11)*. ACM, New York, NY, USA, 50–61. https://doi.org/10.1145/2018436.2018443

[49] Noa Zilberman, Yury Audzevich, G Adam Covington, and Andrew W Moore. 2014. NetFPGA SUME: Toward 100 Gbps as research commodity. *IEEE Micro* 34, 5 (2014), 32–41.

A SUPPLEMENTARY MATERIAL FOR CENTRALIZED WATERFILLING ALGORITHMS

A.1 Correctness of k-Waterfilling algorithms

It can be shown that all k-Waterfilling algorithms compute max-min fair rates for the flows, and the fair share rate of a link when it is removed is exactly the max-min fair rate of its bottleneck flows.

We can prove the following property of a k-Waterfilling algorithm:

Lemma 7. [Monotonic Rate Behavior] *Let link l be removed in iteration n of a k-Waterfilling algorithm. The fair share rate computed for link l (line 8 of Algorithm 1) is non-decreasing from iterations 1 through n.*

Proof. Consider any iteration $1 \leq i < n$. There are two possibilities. If no flows are removed from link l during iteration i, the fair share rate is unchanged. Otherwise, let $\mathbf{c}, \mathbf{n}$ denote the capacity and number of flows carried by link l at the beginning of the iteration, and let $\mathcal{F}$ denote the set of flows removed from link l, at the end of the iteration (omitting l subscript from $\mathcal{F}_l$) . We will show that every flow removed from link l is allocated at most the fair share rate, $\mathbf{c}/\mathbf{n}$, so that the fair share rate in the next iteration increases (or stays the same.) A flow removed during the iteration because of some link k, is allocated no more than the fair share rate of link l, $\mathbf{c}/\mathbf{n}$, because links k

and l are first-degree neighbors. In the next iteration link l's capacity, which is reduced by the sum of allocations of flows in $\mathcal{F}$, is at least $\mathbf{c} - |\mathcal{F}|\mathbf{c}/\mathbf{n}$, while the number of flows is exactly $\mathbf{n} - |\mathcal{F}|$. Hence, the new fair share rate is $\geq (\mathbf{c} - |\mathcal{F}|\mathbf{c}/\mathbf{n})/(\mathbf{n} - |\mathcal{F}|) = \mathbf{c}/\mathbf{n}$. □

This result will allow us to prove that the algorithm does compute max-min fair rates.

Proof of theorem 2 (k-Waterfilling Correctness). *The k-Waterfilling algorithm (Algorithm 1) terminates after a finite number of iterations N_k, and all flows in the network are allocated their max-min fair rates.*

First, we will show that the algorithm terminates after a finite number of iterations, and every flow is allocated a rate. In each iteration, a link is removed if it has the lowest rate of its neighbors (as defined in lines 6–9 of Algorithm 1). So at least one link (such as the link with the minimum fair share rate) is removed in each iteration. The algorithm proceeds until there are no more links with active flows. Hence, the algorithm terminates after a finite number of iterations, which we call N_k, and every flow is allocated a rate.

Next, we will show that the rate allocation is max-min fair. Following definition 1, we will prove that every flow is bottlenecked at some link. Consider a flow f that is removed in iteration j because of some link l, and allocated the fair share rate of the link in iteration j, which we denote as $\mathbf{r}^j$ (lines 5 and 12). We will show that flow f is bottlenecked at link l: (i) link l is fully utilized, and (ii) flow f gets the maximum rate of all flows carried by the link, $A_{l\star}$. Link l is fully utilized because when the link is removed, its remaining capacity is shared equally among all flows that it carries (line 5). Flow f gets the maximum rate because any other flow g that is removed from link l in a previous iteration $i < j$, because of some other link, is allocated at most the fair share rate of link l in iteration i, $\mathbf{r}^i$, because link l and the other link are first-degree neighbors (they share g) and the other link is removed (lines 9 and 12); and the fair share rate of link l is non-decreasing from iteration i to j, $\mathbf{r}^i \leq \mathbf{r}^j$ (Lemma 7), Hence, flow f, removed because of link l in iteration j, is allocated the maximum rate of all flows carried by link l. □

The following lemma shows that in the 2-Waterfilling algorithm, for any link, flows in E^* are removed strictly before flows in B^*, and when a flow in E^* removed, it is allocated strictly less than the link's fair share rate at the time.

Lemma 8. [E^* Flows Removed Before B^*] *Consider the progression of k-Waterfilling for $k \geq 2$ on $(A, \mathbf{c})$. Let link l be removed in iteration n and let $n' \leq n$ denote the first iteration when the link has the lowest fair share rate in its k=1 neighborhood.*

(1) A flow removed from link l in iteration $i < n'$ is allocated strictly less the link's fair share rate in iteration i. As a result, the fair share rate increases in the next iteration. The flow belongs to E^*.

(2) A flow removed from link l in iterations $n' \leq i$ is allocated exactly the link's fair share rate in iteration i. As a result, the fair share rate stays the same in the next iteration (for $i < n$.) The flow belongs to B^*, and the fair share rate equals the max-min fair rate.

Proof. We shall use $\mathbf{r}_l$ to denote the fair share rate of a link l in iteration i. Let f be a flow removed from link l in iteration $i < n'$ because of link m, and allocated $\mathbf{r}_m$. Since link l does not yet have the smallest rate in its k=1 degree neighborhood, there exists a first-degree neighbor of link l, link m' with a strictly smaller fair share rate. Since links m and m' are at most two degrees apart, and link m is removed, $\mathbf{r}_m \leq \mathbf{r}_{m'} < \mathbf{r}_l$. Hence, the allocation of any flow f removed from link l in iteration $i < n'$ is strictly less than the fair share rate of link l. If $\mathcal{F}$ denotes the set of flows removed from link l in iteration i, link l's capacity in the next iteration is strictly greater than $\mathbf{c} - |\mathcal{F}| \cdot \mathbf{r}_l$ and the number of flows is exactly $\mathbf{n}_l - |\mathcal{F}|$. Hence, the fair share rate in the next iteration is $> \frac{\mathbf{c}_l - |\mathcal{F}| \cdot \mathbf{r}_l}{\mathbf{n}_l - |\mathcal{F}|} = \mathbf{r}_l$

Next, consider iteration n'. If no flows are removed from link l, the the fair share rate is unchanged. Otherwise, let f be a flow removed from link l in iteration n' because of link m, and allocated $\mathbf{r}_m$. Since link l has the lowest rate in its k=1 degree neighborhood, $\mathbf{r}_l \leq \mathbf{r}_m$. On the other hand, since link m is removed, $\mathbf{r}_m \leq \mathbf{r}_l$. Hence, the allocation of any flow f removed from link l in iteration n' equals the fair share rate of link l. If $\mathcal{F}$ denotes the set of flows removed from l in the iteration, link l's fair share rate in the next iteration is $\frac{\mathbf{c}_l - |\mathcal{F}| \cdot \mathbf{r}_l}{\mathbf{n}_l} = \mathbf{r}_l$. Recall, from the Monotonic Rate Behavior, that the fair share rate of l's first-degree neighbors is non-decreasing, so that link l still has the lowest fair share rate in its k=1 neighborhood in the next iteration $n' + 1$. Hence, we can repeat the same analysis for iterations n' to n to show that any flow removed from link l is allocated its fair share rate, and the fair share rate is unchanged from one iteration to the next.

Since the fair share rate of link l when it is removed in n equals the max-min fair rate, by theorem 2 (k-Waterfilling Correctness), all flows removed from link l between iterations n' and n belong to B^*. □

B SUPPLEMENTARY MATERIAL FOR *Fair*

B.1 *Fair* vs Existing Work

There are three differences between *Fair* and the distributed algorithm called d-CPG described by Ros-Giralt et al. (see Figures 1–4 in Chapter IV of [42]). First, the "limit rate" calculation in *Fair* and d-CPG have different implementations albeit the results are the same. Second, the "bottleneck rate" calculation in *Fair* is slightly different from d-CPG–when a link l computes a bottleneck rate for a flow f, it temporarily assumes that the flow is not limited. Without this change, the bottleneck rate calculation in d-CPG ("ComputeAR()" procedure in Figure 8 of [42]) is undefined when the sum of limit rates is less than the link capacity. Finally, the proof of the CPG algorithm claims that it takes no more than half a round trip for information about a change in the state of one link to propagate to another (Proof of theorem 4.1 in [42]), whereas the counter-example in table 7 suggests that it can take up to 1.5 RTTs. For the *Fair* algorithm, we assume that it can take up to 2 RTTs (or *rounds*) for new information to propagate between links rather than half a round trip. Hence, the convergence bound of *Fair* in theorem 3 is four times longer than the convergence bound of d-CPG as stated in theorem 4.1 in [42].

B.2 Proof of Convergence of *Fair*

Proof of Theorem 3. *With Fair, given a fixed* $(A, \mathbf{c})$*, and once all flows have been seen at least once by all their links, every flow converges to its max-min fair allocation in less than or equal to* $4N_1$ *rounds, where* N_1 *is the number of iterations that the* 1*-Waterfilling algorithm takes for* $(A, \mathbf{c})$.

Let $\mathcal{F}$ and $\mathcal{L}$ denote the set of flows and links removed in the *first* iteration of 1-Waterfilling. We will show that within four rounds, the flows in $\mathcal{F}$ are allocated their max-min rates at all links, and this is reflected in the limit rates stored at the links.

First, we show that within a round every flow $f \in \mathcal{F}$ is picks up a bottleneck rate that is at least $\mathbf{x}_f^*$ from all its links. Consider link m on the flow's path. For any link, including link m, the

bottleneck rate satisfies:

$$
\begin{aligned}
b &= \frac{\mathbf{c} - \sum_{\{g|\mathbf{e}_g < b\}} \mathbf{e}_g}{|\{g \mid \mathbf{e}_g \geq b\}|}. \\
b &\geq \frac{\mathbf{c} - |\{g \mid \mathbf{e}_g < b\}| \cdot b}{|\{g \mid \mathbf{e}_g \geq b\}|} && (\mathbf{e}_g < b). \\
b &\geq \frac{\mathbf{c}}{\mathbf{n}}. && (4)
\end{aligned}
$$

Since link m and the link because of which flow f is removed, call it link l, are first-degree neighbors, the fair share rate of link m is at least the fair share rate of link l, which is exactly the allocation of flow f, $\mathbf{c}/\mathbf{n} \geq \mathbf{x}^*_f$.

By the end of the second round, the limit rate of all flows at every link $l \in \mathcal{L}$ is at least $\mathbf{c}_l/\mathbf{n}_l$, because any flow that the link carries is in $\mathcal{F}$, and the flow gets a bottleneck rate of at least $\mathbf{x}^*_f = \mathbf{c}_l/\mathbf{n}_l$ from all links on its path. If all limit rates are at least $\mathbf{c}/\mathbf{n}$, the bottleneck rate at link l can be no more than $\mathbf{c}/\mathbf{n}$:

$$
\begin{aligned}
b &= \frac{\mathbf{c} - \sum_{\{g|\mathbf{e}_g < b\}} \mathbf{e}_g}{|\{g \mid \mathbf{e}_g \geq b\}|}. \\
b &\leq \frac{\mathbf{c} - |\{g \mid \mathbf{e}_g < b\}| \cdot \mathbf{c}/\mathbf{n}}{|\{g \mid \mathbf{e}_g \geq b\}|} && (\mathbf{e}_g \geq \mathbf{c}/\mathbf{n}). \\
b &\leq \frac{\mathbf{c}}{\mathbf{n}}. && (5)
\end{aligned}
$$

Equations (4) and (5). imply that the bottleneck rate at link $l \in \mathcal{L}$ is exactly $\mathbf{c}_l/\mathbf{n}_l$. Hence, in the third round, flows in $\mathcal{F}$ pick up their local max-min rate from links in $\mathcal{L}$, and they are allocated this rate by the remaining links by the fourth round. Next we remove flows in $\mathcal{F}$ and links in $\mathcal{L}$ and repeat the same analysis for the reduced network, which consists of the remaining flows and links. Eventually all flows are updated correctly within $4N_1$ rounds.

It is justified to repeat the analysis in the reduced network because the updates (bottleneck rate calculation) for the remaining flows are identical in the original and reduced network.

□

C SUPPLEMENTARY MATERIAL FOR S-PERC

C.1 Good Rate Propagation Property

Proof of Lemma 5 (Good Rate Propagation). *The bottleneck rate* propagated *by a link is at least* $\mathbf{c}/\mathbf{n}$*, the fair share rate of the link, irrespective of the limit rates of the flows seen at the link.*

Note that the statement is well-defined for every update, we say that the *propagated* bottleneck rate is ∞ when the ignore bit is set and b otherwise. Consider an update of flow f at link l. We will show that the bottleneck rate *propagated* by the link is at least $\mathbf{c}/\mathbf{n}$.

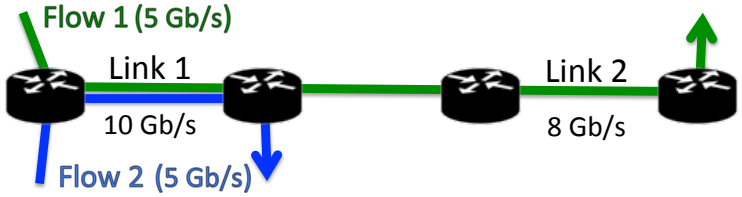
If $MaxE \geq \mathbf{c}/\mathbf{n}$, we are done, since if the bottleneck rate is propagated, it satisfies $b > MaxE$, which is at least $\mathbf{c}/\mathbf{n}$. Hence, we consider the case where $MaxE < \mathbf{c}/\mathbf{n}$. We show that the bottleneck rate is at least $\mathbf{c}/\mathbf{n}$:

$$
b = \frac{\mathbf{c} - SumE}{NumB} = \frac{\mathbf{c} - \sum_{f \in E} \mathbf{x}_f}{|B|} \geq \frac{\mathbf{c} - |E| \cdot \mathbf{c}/\mathbf{n}}{|B|} = \mathbf{c}/\mathbf{n}.
$$

Here, we used the fact that $MaxE$ is an upper bound to allocations of flows in E. □

Table 7. An example sequence of updates with d-CPG [43] which takes 1.5RTTs for new information to propagate from link 1 to link 2 via the control packet of their shared flow 2.

Time (RTTs)	Control Packet at Link 1	Control Packet at Link 2	Notes
0.00	Flow 1 (fwd.) UB=∞ ER=10Gb/s		Flow 1 picks up an explicit rate (ER) 10 Gb/s from link 1 with a control packet in the forward direction.
0.01	Flow 2 (fwd.) UB=∞ ER=10Gb/s		Link state at link 1 changes: number of flows increases from 1 to 2.
0.50		Flow 1 (fwd.) UB=10Gb/s ER=8Gb/s	Link 2 learns about the old state at link 1: flow 1 is limited to upstream bandwidth (UB) 10 Gb/s .
0.51		Flow 1 (rvs.) DB=∞ ER=8Gb/s	
0.99	Flow 1 (rvs.) DB=8Gb/s ER=5Gb/s		Flow 1 picks up a new explicit rate (ER) 5 Gb/s from link 1 with a control packet in the reverse direction.
1.00	Flow 1 (fwd.) UB=∞ ER=5Gb/s		Flow 1 picks up a new explicit rate (ER) 5 Gb/s from link 1 with a control packet in the forward direction.
1.50		Flow 1 (fwd.) UB=5 Gb/s ER=5 Gb/s	Link 2 learns about the new state at link 1: flow 1 is limited to upstream bandwidth (UB) 5 Gb/s .



d-CPG counter example setup.

C.2 Properties of Bottleneck Rate Calculation

Proof of Lemma 6 (Local Bottleneck Rate Property). *Suppose limit rates at a link are at least* $\mathbf{c}/\mathbf{n}$ *from time* T*, then the bottleneck rate computed by the link equals* $\mathbf{c}/\mathbf{n}$ *by time* $T + 2 \cdot round$*. If on the other hand, limit rates are bounded below by a smaller value* $e_{min} < \mathbf{c}/\mathbf{n}$*, the bottleneck rate strictly exceeds* e_{min} *by time* $T + 2 \cdot round$*.*

Case 1: Assume that limit rates satisfy $e \geq \mathbf{c}/\mathbf{n}$ from time T. We will show that for all updates from time $T + 2 \cdot round$,[8] the bottleneck rate satisfies $b = \mathbf{c}/\mathbf{n}$.

[8]Note that the variable $round$ denotes the duration of a round in seconds. Thus $T + 2 \cdot round$ refers to the time 2 rounds after time T.

(1) All flows are seen at the link between times T and $T + 1 \cdot round$ with limit rates at least $\mathbf{c}/\mathbf{n}$. They are either classified into B, or classified into E and allocated at least $\mathbf{c}/\mathbf{n}$. Hence, from time $T + 1 \cdot round$, the control packet carries either $s = \text{B}$ or $x \geq \mathbf{c}/\mathbf{n}$, and since the link state, $SumE$ and $NumB$, reflects the control packet state of the link's flows, the bottleneck rate in any update satisfies $b \leq \mathbf{c}/\mathbf{n}$:

$$b = \frac{\mathbf{c} - SumE}{NumB} = \frac{\mathbf{c} - \sum_{f \in E} \mathbf{x}_f}{|B|} \leq \frac{\mathbf{c} - |E| \cdot \mathbf{c}/\mathbf{n}}{|B|} = \mathbf{c}/\mathbf{n}.$$

(2) All flows are seen at the link between times $T + 1 \cdot round$ and $T + 2 \cdot round$ with $b \leq \mathbf{c}/\mathbf{n} \leq e$, and classified into B. Hence, from time $T + 2 \cdot round$, the control packet carries $s = \text{B}$, and the bottleneck rate in any update satisfies $b = \mathbf{c}/\mathbf{n}$ because $SumE = 0$ and $NumB = \mathbf{n}$.

Note that the last time a flow is classified into E with a limit rate at least $\mathbf{c}/\mathbf{n}$ must be before time $T + 1 \cdot round$, since $b \leq \mathbf{c}/\mathbf{n}$ after.

Case 2: Assume that limit rates satisfy $e \geq e_{min}$, and we are only given that $e_{min} < \mathbf{c}/\mathbf{n}$. We will show that for all updates from time $T + 2 \cdot round$, the bottleneck rate satisfies $b > e_{min}$.

(1) All flows are seen at the link between times T and $T + 1 \cdot round$ with limit rates at least e_{min}, and either classified into B or classified into E and allocated at least e_{min}.

(2) First we show that the link state following some update between times $T + 1$ and $T + 2 \cdot round$ satisfies:

$$\mathbf{c} - SumE > e_{min} \cdot NumB.$$

Consider the first time after $T + 1 \cdot round$ that control packets of all flows are seen at the link. If $b \leq e$ in all updates, then following the last update the link state satisfies $NumB = \mathbf{n}$ and $SumE = 0$, so that $(\mathbf{c} - SumE)/NumB = \mathbf{c}/\mathbf{n} > e_{min}$. On the other hand, it is possible that during some update, a flow is moved to E because $b = (\mathbf{c} - SumE)/NumB > e$, where $SumE$ and $NumB$ represent the values used to compute b (line 9 of Algorithm 4). Since the flow is moved to E, the new values of $SumE$ and $NumB$ at the link following the update are $SumE + e$ and $NumB - 1$. Now notice that:

$$\begin{aligned} c - (SumE + e) &> NumB \cdot e - e, \\ &= e \cdot (NumB - 1), \\ &\geq e_{min} \cdot (NumB - 1). \end{aligned}$$

Here, we have used the fact $(\mathbf{c} - SumE)/NumB > e$ in the beginning, which follows from the flow's update.

(3) Next, we show that after $T + 1 \cdot round$, if the link state satisfies $\mathbf{c} - SumE > NumB \cdot e_{min}$ before an update, it satisfies the inequality after the update as well. If flow is moved to E during the update ($b > e$), we can follow the same reasoning as in point (2). Otherwise, if $b \leq e$ and the flow was previously in E, then the new values of $SumE$ and $NumB$ at the link are $SumE - \mathbf{x}$ and $NumB + 1$, where $\mathbf{x}$ is the old rate allocated to the flow. The old allocation is at least e_{min}, since it was made within the last round, and limit rates are at least e_{min} from time T. Since $\mathbf{c} - SumE > NumB \cdot e_{min}$ and $\mathbf{x} \geq e_{min}$, we have:

$$\begin{aligned} c - (SumE - \mathbf{x}) &> NumB \cdot e_{min} + \mathbf{x}, \\ &\geq NumB \cdot e_{min} + e_{min}, \\ &= e_{min} \cdot (NumB + 1). \end{aligned}$$

(4) Finally, we show that $\mathbf{c} - SumE > NumB \cdot e_{min}$ implies that the bottleneck rate during an update satisfies $b > e_{min}$. If the flow being updated was previously in B, that is $\mathbf{s} = B$ before

the update, then it immediately follows $b = (\mathbf{c} - SumE)/NumB > e_{min}$. Otherwise, letting $\mathbf{x}$ denote the old rate allocated to the flow before the update, we have:

$$b = \frac{\mathbf{c} - (SumE - \mathbf{x})}{NumB + 1} > \frac{NumB \cdot e_{min} + e_{min}}{NumB + 1} = e_{min}.$$

□

The following fact also trivially follows from the proof (for Case 1), and will be useful to understand the behavior of $MaxE$ later in the convergence proof of s-PERC.

Corollary 8.1. *Suppose limit rates at a link are at least* $\mathbf{c}/\mathbf{n}$ *from time* T*, then the last control packet, for which a flow was classified into* E *with limit rate at least* $\mathbf{c}/\mathbf{n}$*, must be seen before time* $T + 1 \cdot round$*.*

C.3 Approximating the Maximum E Allocation

In the s-PERC algorithm, every link l starts to reset their $MaxE$ and $MaxE'$ independently at some time, and thereafter resets $MaxE'$ and $MaxE$ after every *round* seconds according to Algorithm 5. The variable $MaxE$ satisfies two key properties:

Lemma 9. *Suppose link* l *starts resetting* $MaxE$ *and* $MaxE'$ *at time* T*. From time* $T + 1 \cdot round$ *onward, the variable* $MaxE$ *is at least the maximum allocation of any flow in* E*, that is,* $MaxE \geq \max_{f \in E} \mathbf{x}_f$*.*

Proof.

(1) At time $T + 1 \cdot round$, right before the reset, $MaxE'$ is at least the maximum rate allocated to any E flow: $MaxE' \geq \max_{f \in E} x_f$. Therefore, immediately after the reset, $MaxE \geq \max_{f \in E} x_f$.
(2) During the next round, until time $T + 2 \cdot round$, following any update in which a flow g is classified as E and is allocated rate x_g, the link updates $MaxE$ to $\max(MaxE, x_g)$ (line 22 in Algorithm 4). Therefore, after every update: $MaxE \geq \max(\max_{f \in E \setminus g} x_f, x_g) = max_{f \in E} x_f$.
(3) At time $T + 2 \cdot round$, once again, $MaxE' \geq \max_{f \in E} x_f$ right before the reset, and $MaxE \geq \max_{f \in E} x_f$ immediately after the reset. Hence, we have the same argument from time $T + 2 \cdot round$ onward by induction.

Therefore, we have established that from time $T + 1 \cdot round$ onward, we always have $MaxE \geq \max_{f \in E} x_f$.

□

Lemma 10. *If the maximum allocation of any* E *flow at the link stabilizes at time* T*, then* $MaxE$ *reflects this value by time* $T + 2 \cdot round$*.*

Proof. Let the first reset *after* time T occur at time R.

(1) At time $R + 1 \cdot round$, right before the reset, $MaxE' = \max_{f \in E} x_f$. Therefore, immediately after the reset, $MaxE = \max_{f \in E} x_f$.
(2) During the next round, until time $R + 2 \cdot round$, $MaxE = \max_{f \in E} x_f$ after any update, since the maximum allocation of E flows does not change from time T onward.
(3) At time $R + 2 \cdot round$, once again, $MaxE' = \max_{f \in E} x_f$ right before the reset, and $MaxE = \max_{f \in E} x_f$ immediately after the reset. Hence, we have the same argument for time $R+2 \cdot round$ by induction.

We have established that from time $R + 1 \cdot round$ onward, we always have $MaxE = \max_{f \in E} x_f$. The result follows by observing that $R \leq T + 1 \cdot round$, since a reset occurs every *round* seconds. □

C.4 Proof of Convergence of s-PERC

Proof of Theorem 4. *With s-PERC, given a fixed* $(A, \mathbf{c})$*, once all flows have been seen at their links, every flow converges to its max-min fair rate in less than or equal to* $6N_2$ *rounds, where* N_2 *is the number of iterations that* 2*-Waterfilling takes for* $(A, \mathbf{c})$.

Let T denote the first time at which all flows have been seen at their links. Let $\mathcal{F}$ and $\mathcal{L}$ denote the set of flows and links removed in the *first* iteration of 2-Waterfilling. We will show that by time $T + 6 \cdot round$, the flows in $\mathcal{F}$ are allocated their max-min rates, and classified correctly into B or E at all links, and this is reflected in the bottleneck state and allocations stored in the control packets, and hence in the aggregate state at the links.

Let $f \in \mathcal{F}$ be removed because of some link $k \in \mathcal{L}$. We will use "link l" to refer to other links on the path of flow f, which have fair share rates equal to link k or greater (since link $k \in \mathcal{L}$, by definition, has the lowest fair share rate). We will consider both cases.

Step 1. *From time* $T + 1 \cdot round$*: (1) limit rates of all flows at link* k *are at least* $\mathbf{c}_k/\mathbf{n}_k$*; (2) the limit rates of all flows at any other link* l *on the path of flow* f *are also at least* $\mathbf{c}_k/\mathbf{n}_k$.

Proof. First, let's consider link k. Since all flows have been seen at least once by all their links by time T, they pick up a bottleneck rate from all their links and by Lemma 5 (Good Rate Propagation), limit rates of all flows at link k satisfy $e \geq \mathbf{c}_k/\mathbf{n}_k$ by time $T + 1 \cdot round$. This property follows from Lemma 5 because the limit rate of any flow at link k is the smallest bottleneck rate propagated by any other link on the flow's path, which is a first-degree neighbor of link k. Since link k has the smallest fair share of first-degree neighbors, the limit rate of the flow is at least the fair share rate of link k. A similar argument shows that the limit rates of *all* flows at any other link l on the path of flow f are at least $\mathbf{c}_k/\mathbf{n}_k$. The limit rate of any flow at link l is a bottleneck rate propagated by a second-degree neighbor of link k. Since link k has the smallest fair share of second-degree neighbors, the limit rate of a flow at link l is at least the fair share rate of link k. □

Step 2. *From time* $T + 3 \cdot round$*: (1) the bottleneck rate at link* k *equals the fair share rate of link* k*,* $\mathbf{c}_k/\mathbf{n}_k$*; (2) the bottleneck rate at any other link* l *on the path of flow* f *with a strictly greater fair share than* $\mathbf{c}_k/\mathbf{n}_k$ *is strictly greater than* $\mathbf{c}_k/\mathbf{n}_k$*. If link* l *has a fair share rate equal to* $\mathbf{c}_k/\mathbf{n}_k$*, its bottleneck rate equals* $\mathbf{c}_k/\mathbf{n}_k$.

Proof. This follows from Lemma 6 (Local Bottleneck Rate Property), and the lower bound of $\mathbf{c}_k/\mathbf{n}_k$ on the limit rates of all flows at link k and at other links l on the path of flow f from time $T + 1 \cdot round$. □

Step 3. *From time* $T + 3 \cdot round$*: (1) flow* f *is classified correctly into* B *at link* k *and allocated the max-min rate* $\mathbf{c}_k/\mathbf{n}_k$*; (2) further, if any link* l *on flow* f*'s path has a fair share equal to* $\mathbf{c}_k/\mathbf{n}_k$*, flow* f *is also classified correctly into* B *and allocated* $\mathbf{c}_k/\mathbf{n}_k$ *at that link.*

Proof. From time $T + 3 \cdot round$, the bottleneck rate computed for flow f is exactly $\mathbf{c}_k/\mathbf{n}_k$ at links that have fair share rates equal to $\mathbf{c}_k/\mathbf{n}_k$ (including link k), while the limit rate is at least $\mathbf{c}_k/\mathbf{n}_k$. So flow f is classified into B at such links and allocated $\mathbf{c}_k/\mathbf{n}_k$. This is the max-min rate for flow f because of theorem 2 (k-Waterfilling Correctness). This is the correct classification because the links have the lowest fair share rate in their $k = 1$ neighborhood, and by lemma 8 (E^* Flows Removed Before B^*), the rate equals their max-min fair rate. □

Step 4. *From* $T + 4 \cdot round$*, the bottleneck rate from link* k *is propagated as is* ($\mathbf{i}_k = 0$) *to other links* l *on the path of flow* f.

Proof. Given that limit rates at link k are at least $\mathbf{c}_k/\mathbf{n}_k$ from $T + 1 \cdot round$, all flows are classified into B at link k from time $T + 2 \cdot round$, and the last control packet, for which a flow is classified into E, must be seen at the link before time $T + 2 \cdot round$ (Corollary 8.1 of Lemma 6). So $MaxE$ at link k drops to 0, and the bottleneck rate is propagated to other links from time $T + 4 \cdot round$, that is, the updates of all flows (including f) at link k satisfy $MaxE < b = \mathbf{x}^*_f$. From time $T + 4 \cdot round$, the control packets of all flows (including f) carries $\mathbf{i}_k = 0$ and $\mathbf{b}_k = \mathbf{c}_k/\mathbf{n}_k$, indicating that the correct bottleneck rate is propagated by link k. □

Step 5. *From time $T + 5 \cdot round$, at any link l with a higher fair share than link k, flow f's limit rate is exactly $\mathbf{c}_k/\mathbf{n}_k$, while the bottleneck rate computed by link l is strictly greater than $\mathbf{c}_k/\mathbf{n}_k$. So the flow is allocated $\mathbf{x}^*_f = \mathbf{c}_k/\mathbf{n}_k$, and classified correctly into E at the link.*

Proof. This follows from the fact that link k propagates its bottleneck rate of $\mathbf{c}_k/\mathbf{n}_k$ from time $T + 4 \cdot round$ (Step 4), and the fact that bottleneck rate at link l strictly exceeds $\mathbf{c}_k/\mathbf{n}_k$ (Step 2).

Link l's decision to classify flow f into E is correct, because the max-min rate of a link is at least the fair share rate of the link, and the fair share rate is higher than $\mathbf{c}_k/\mathbf{n}_k$. □

Hence, within six rounds of time T, all flows (like flow f) in $\mathcal{F}$ are allocated their max-min rates, and classified correctly into B or E at all their links, and this is reflected in the bottleneck state and allocations stored in the control packets, and hence in the aggregate state at the links (since $SumE$ and $NumB$ can be expressed in terms of the control packet state.)

From time $T + 6 \cdot round$, we can ignore $\mathcal{F}$ and $\mathcal{L}$ and consider the reduced network, comprising the remaining flows and their links, where for each link, we reduce the original capacity by the max-min fair allocations of any removed flows. The s-PERC algorithm updates in the reduced network for the remaining flows are identical to the updates in the original network.

We can repeat the same analysis to show that flows removed in the first iteration of the 2-Waterfilling algorithm for the *reduced* network are updated correctly at all their links within an additional 6 rounds. By induction, all flows removed in iterations one through N_2, are correctly updated, and their control packets carry the correct allocations and bottleneck state for all their links by time $T + 6 \cdot N_2 \cdot round$. □

C.4.1 Control Packet Updates in the Reduced Network. We explain why the updates of flows at a link l in the reduced network are identical to their updates in the original network. We use the prime symbol ($'$) to refer to variables in the the reduced network. We use the notation $\mathcal{F}_l$ to refer to link l's flows that are removed in the first iteration of the 2-Waterfilling algorithm in the original network.

(1) First, consider the bottleneck rate calculation. There are two cases, either $\mathcal{F}_l \subset B^*$ or they $\mathcal{F}_l \subset E^*$. In the first case, we have the result, from analyzing the original network, that all flows of link l, $A_{l\star}$ are permanently classified into B after $T + 6 \cdot round$. Given this, the bottleneck rate calculation in the reduced network is equal to that in the original network: $b' = \frac{\mathbf{c}'-SumE'}{NumB'} = \frac{\mathbf{c}'}{\mathbf{n}'} = \frac{\mathbf{c}-|\mathcal{F}_l|\cdot \mathbf{c}/\mathbf{n}}{\mathbf{n}-|\mathcal{F}_l|} = \frac{\mathbf{c}}{\mathbf{n}} = b$. In the second case, where $\mathcal{F}_l \subset E^*$, we have the result, from analyzing the original network, that all flows in $\mathcal{F}_l$ are permanently classified into E after $T + 6 \cdot round$. Given this, the bottleneck rate calculations are equal: $b' = \frac{\mathbf{c}'-SumE'}{NumB'} = \frac{\mathbf{c}'-\sum_{E\cap A'_{l\star}} \mathbf{x}}{|B\cap A'_{l\star}|} = \frac{\mathbf{c}-\sum_{E\cap A_{l\star}} \mathbf{x}}{|B\cap A_{l\star}|} = b$.

(2) Next consider the rate propagation, the bottleneck rate is propagated when $b \geq MaxE$. We can show that the value of $MaxE$ during updates of in the reduced network, *for flows in the reduced network*, is exactly the same as in the original network. We'll consider a link l for which the flows removed in the original network $\mathcal{F}_l \subset E^*$. We have the result from analyzing

Table 8. Outline of steps in the proof of convergence of the s-PERC algorithm. We assume we are given a network as in Figure 5 where link k has the lowest fair share rate of first- and second-degree neighbors like link l and link j respectively.

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 $\mathbf{i} = 1$
$T + 1 \cdot round$	$\mathbf{e} \geq \mathbf{c}_k/\mathbf{n}_k$	$\mathbf{e} \geq \mathbf{c}_k/\mathbf{n}_k$	
$T + 2 \cdot round$			
$T + 3 \cdot round$	$\mathbf{b} = \mathbf{c}_k/\mathbf{n}_k$ $\mathbf{s} = \text{B}$ $\mathbf{x} = \mathbf{b} = \mathbf{c}_k/\mathbf{n}_k$	$\mathbf{b} > \mathbf{c}_k/\mathbf{n}_k$	
$T + 4 \cdot round$	$\mathbf{i} = 0$		
$T + 5 \cdot round$		$\mathbf{e} = \mathbf{c}_k/\mathbf{n}_k$ $\mathbf{s} = \text{E}$ $\mathbf{x} = \mathbf{e} = \mathbf{c}_k/\mathbf{n}_k$ (for flow f)	

the original network, that flows in $\mathcal{F}_l$ converge to their correct limit rates, $\mathbf{x}^*$, at link l after $T + 6 \cdot round$, while the remaining flows have limit rates that are at least $\mathbf{x}^*$. Hence, whenever $MaxE$ or $MaxE'$ is updated by a flow in the reduced network, its value equals the value in the original network.

D SUPPLEMENTARY MATERIAL FOR MAKING S-PERC PRACTICAL

D.1 Flow Completion Times with Dynamic Data Center Workloads (continued)

Here, we include the results from Flow Completion Time experiments at 80% load for search (Figure 12) and data-mining (Figure 13) workloads. The results are similar to results at 60% load.

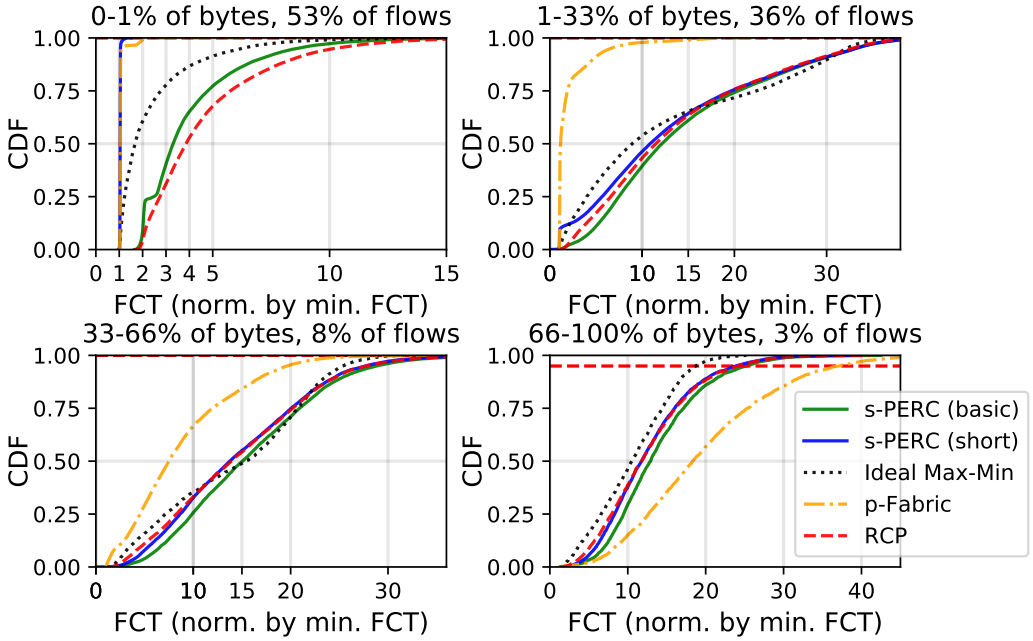


Fig. 12. FCTs for search workload at 80% load. Settings in table 6. Note bin 4 has < 30 samples for the 95th percentile and above (horizontal red line.)

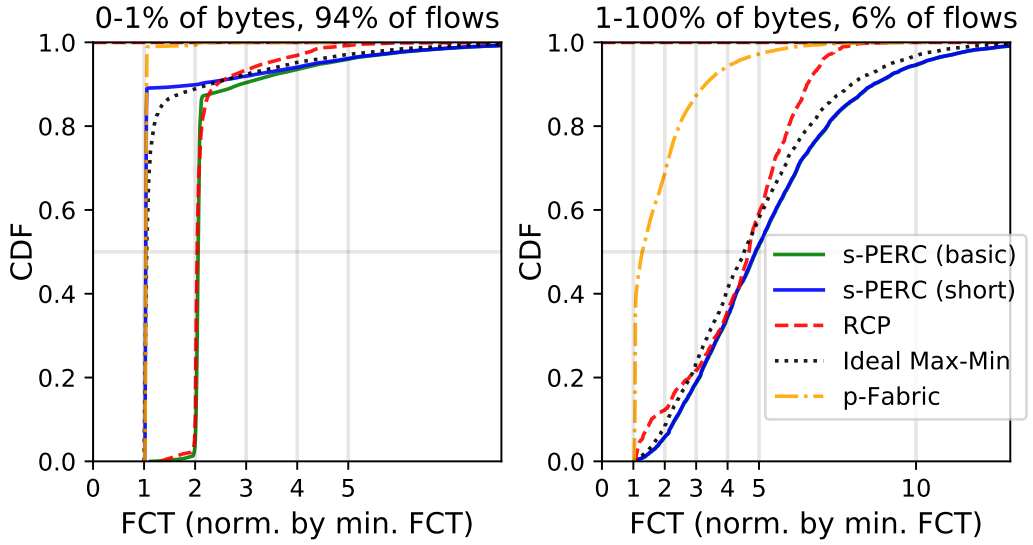


Fig. 13. FCTs for data-mining workload at 80% load. Settings in Table 6.

D.2 Pseudocode for s-PERC*

See algorithms 6 and 7 for a description of the s-PERC* algorithm running at the links. In order to handle control packet drops, the basic s-PERC algorithm has been enhanced with "shadow" variables (a term used by [15]) $SumE'$ and $NumB'$ corresponding to the original variables $SumE$ and $NumB$, respectively. The shadow variables are updated along with the original variables whenever a control packet is received (algorithm 6), and they are synced to the original variables and reset at the end of each round at the link (algorithm 7.) See table 9 for an example of an initial s-PERC* control packet sent by an end-host. The control packet carries a new per-link variable, **t**, which is a vector of the latest round numbers seen at each link on the flow's path.

Table 9. s-PERC* initial control packet for 10kB flow f_W after it has left the end-host, carrying per-link information. First row corresponds to virtual link l_0, carrying size information relevant for short flows.

Link	Bottleneck State (**s**) Allocation (**x**)	Bottleneck rate (**b**)	ignore	round (**t**)
l_0	$(E\ @\ 0)$	$\frac{10\ \text{MB}}{1\ \text{RTT}}$	0	(0)
l_{30}	$E\ @\ 0$	0	1	0
l_{12}	$E\ @\ 0$	0	1	0
		$Tx_{ctrl} : \infty$ $FIN : 0$		

Algorithm 7 Timeout action at link l for s-PERC, every round

1: $MaxE \leftarrow MaxE'$; $MaxE' \leftarrow 0$
2: $SumE \leftarrow SumE'$; $SumE' \leftarrow 0$
3: $NumB \leftarrow NumB'$; $NumB' \leftarrow 0$
4: $T \leftarrow T + 1$

Received February 2019; revised March 2019; accepted April 2019

Algorithm 6 s-PERC* control packet processing at link l showing additional steps for robustness to drops. For simplicity, we omit steps to calculate rate-limits. See also Algorithm 7.

b, x, s : vector of bottleneck, allocated rates, bottleneck states in packet (initially, ∞, 0, E, respectively.)
i: vector of ignore bits in packet (initially, 1.)
t : vector of round numbers in packet (initially, 0.)
$SumE$, $SumE'$: sum of limit rates of E flows since last round, and in this round.
$NumB$, $NumB'$: number of B flows at link since last round, and in this round.
$MaxE$, $MaxE'$: max. allocated rate of flows classified into E since last round, and in this round.
T: round number at this link, initially 1.
if $\mathbf{s}[l]$ = E **then** ▷ Assume flow is not limited, for bottleneck rate calculation
    $\mathbf{s}[l] \leftarrow$ B
    $SumE \leftarrow SumE - \mathbf{x}$
    if $\mathbf{t}[l] = T$ **then** $SumE' \leftarrow SumE' - \mathbf{x}$ ▷ Update if flow prev. seen in round.
    $NumB \leftarrow NumB + 1$
    if $\mathbf{t}[l] = T$ **then** $NumB' \leftarrow NumB' + 1$
else if $\mathbf{t}[l] \neq T$ **then**
    $NumB' \leftarrow NumB' + 1$ ▷ Update if flow not seen in round yet.
$b \leftarrow (\mathbf{c} - SumE)/NumB$
foreach link j**:**
 if $\mathbf{i}[j] = 0$ **then** $\mathbf{p}[j] \leftarrow \mathbf{b}[j]$ **else** $\mathbf{p}[j] \leftarrow \infty$ ▷ Propagated rates
$\mathbf{p}[l] \leftarrow \infty$ ▷ Assume the link's own propagated rate is ∞
$e \leftarrow \min \mathbf{p}$
$x \leftarrow \min(b, e)$
$\mathbf{b}[l] \leftarrow b$, $\mathbf{x}[l] \leftarrow x$ ▷ Save variables to packet.
if $b < MaxE$ **then** $\mathbf{i}[l] \leftarrow 1$ **else** $\mathbf{i}[l] \leftarrow 0$ ▷ Indicate if rate $\mathbf{b}[l]$ should be ignored.
if flow is leaving **then**
    $NumB \leftarrow NumB - 1$, $NumB' \leftarrow NumB' - 1$ ▷ Remove flow f
else if $e < b$ **then**
    $\mathbf{s}[l] \leftarrow$ E
    $SumE \leftarrow SumE + x$; $SumE' \leftarrow SumE' + x$.
    $NumB \leftarrow NumB - 1$; $NumB' \leftarrow NumB' - 1$.
    $MaxE \leftarrow \max(x, MaxE)$; $MaxE' \leftarrow \max(x, MaxE')$.
if $\mathbf{t}[l] \neq T$ **then** $\mathbf{t}[l] = T$ ▷ Update packet to say flow has been seen in this round.