Cebinae: Scalable In-network Fairness Augmentation

Liangcheng Yu
University of Pennsylvania
leoyu@seas.upenn.edu

John Sonchack
Princeton University
jsonch@princeton.edu

Vincent Liu
University of Pennsylvania
liuv@seas.upenn.edu

ABSTRACT

For public networks like the Internet and those of many clouds, end-host applications can use any congestion control protocol they wish. This protocol diversity and application autonomy are only increasing over time. While in-network support for fairness is an attractive solution for reigning in the inequity, existing solutions still have difficulty scaling to today's networks using today's devices.

In this paper, we present Cebinae, a mechanism for augmenting existing networks of legacy hosts with penalties for flows that exceed their max-min fair share. Cebinae is compatible with all of the congestion control protocols in today's Internet, is deployable on commodity programmable switches, and scales orders of magnitude beyond existing alternatives.

CCS CONCEPTS

• **Networks** → **Programmable networks**; **In-network processing**; **Network control algorithms**; **Public Internet**; **Transport protocols**; *Network monitoring*; Network management; Network dynamics; *Network performance analysis*;

KEYWORDS

Programmable networks, P4, Congestion control, Max-min fairness

ACM Reference Format:
Liangcheng Yu, John Sonchack, and Vincent Liu. 2022. Cebinae: Scalable In-network Fairness Augmentation. In *ACM SIGCOMM 2022 Conference (SIGCOMM '22), August 22–26, 2022, Amsterdam, Netherlands.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3544216.3544240

1 INTRODUCTION

Congestion control is one of the most fundamental components of the Internet. Pioneered in the 1980s to avert congestion collapse, congestion control—in its various forms—is still a core responsibility of nearly every transport-layer protocol in existence. In each case, the goal is to ensure that all of the disparate connections in the Internet (collectively) use its capacity efficiently and fairly.

Unfortunately, while TCP is generally effective at eventually consuming all available network capacity, it has never really been fair. Even in the early days of the Internet, when congestion control was relatively homogenous, differing RTTs (i.e., the typical case in the Internet) resulted and continue to result in unfair allocations. Different congestion control algorithm variants—or even operating systems' implementations or configurations of the same variant—can lead to similar results [44, 46, 52, 57]. To be clear, the above innovations in congestion control have made tangible improvements to the quality of experience of Internet users. Further, because applications can have diverse requirements for their transport-layer protocols (e.g., for backward compatibility, small-flow efficiency, or jitter), new protocols can serve important purposes in the continual evolution of the Internet.

Even so, as evidenced by the cited examples, the traditional approach of relying on end hosts to self-police their 'TCP friendliness' [40, 58] has led to inconsistent results. It is also important to note that many of the above instances of unfairness are more than just a new protocol that can ramp up to its fair share more quickly—they describe converged behavior and the possibility of persistent starvation.

Fundamentally, there are few incentives for hosts to prioritize others' fair share and even fewer consequences for failing to do so. We note that cloud deployments do not escape these challenges, especially given the trend toward user-space networking, where enforcing congestion control adherence becomes more difficult.

In the end, when fairness is required in either the public cloud or the Internet, the only actors with an incentive to effectuate it are the network operators. For that reason, there have been many proposals for network-supported fairness. In some, routers compute and communicate explicit rates that are guaranteed to be both fair and efficient [18, 29, 42]. In others, the routers implement specialized queuing disciplines like fair queuing [39, 47, 51] or preferential variants of AQM [38, 41] to achieve similar results.

Unfortunately, the router modifications required by the above approaches have, thus far, limited their practicality. For example, AFQ [47] recently showed how to approximate fair queuing in commodity programmable switches, which was a substantial improvement over the hardware requirements of prior work. Conceptually, fair queuing operates by assigning every flow to an independent queue, and serving each in per-bit round-robin order. AFQ follows prior work in virtualizing the per-flow queues [17]; however, to simulate a fair-queuing schedule, its accuracy still requires the exclusive use of many priority levels and restrictions on the amount of usable buffer for each flow—both limitations grow more stringent with higher flow count, RTT, and burstiness. Thus, for networks with existing needs for differentiated service and networks with many flows, WAN connections, or less-disciplined congestion control variants, e.g., the Internet and public clouds, the requirements on queuing resources can quickly exceed typical hardware resources. Other approaches, such as those that rely on ingress admission control rather than per-flow queuing (e.g., [38]), skip the expensive per-flow queue tracking but are insufficient for today's Internet, which contains congestion control algorithms that ignore loss (e.g., BBRv1 [14]).

ACM ISBN 978-1-4503-9420-8/22/08…$15.00
https://doi.org/10.1145/3544216.3544240

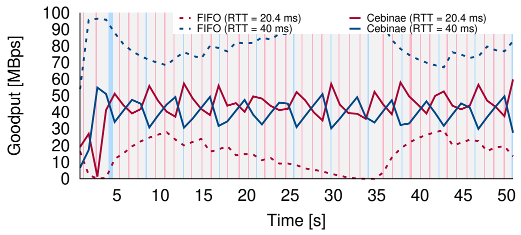


Figure 1: The fairness of two New Reno flows with differing RTTs with and without Cebinae (the background color indicates Cebinae's state: unsaturated, red-bottlenecked/blue-bottlenecked).

In this paper, we advocate for *Cebinae*[1], a simpler approach to eventual in-network fairness. Cebinae at a single router pushes the allocation of attached links towards max-min fairness regardless of the resident congestion control algorithms; applied across a network with congestion-controlled hosts, it provides equivalent steady-state global max-min fairness convergence properties as fair queuing and related techniques.

Cebinae is based on two key insights. First is the observation that byte- or packet-level scheduling at every instance in time is overkill for global convergence; instead, it suffices for the network to redistribute bandwidth from flows that have met/exceeded their fair share to flows that have not. The second is that this simplification enables extremely efficient approximations of advanced scheduling logic like leaky-bucket filters with minimal resources (just one extra queue priority) and greatly improves on the scalability of existing approaches. While gradual redistribution may not provide the rigid, packet-level guarantees of fair queuing, it preserves the capability of mitigating unfairness in practical settings.

We validate the feasibility and performance of Cebinae under a diverse set of network environments in both a hardware testbed and simulation. Figure 1 shows one such result, illustrating Cebinae's effect for two TCP New Reno flows with differing RTTs. This paper makes the following contributions:

- We present Cebinae, the first in-network mechanism that can scale fairness augmentation to the Internet and public clouds using only commodity programmable switches.
- We introduce a novel and general technique for shaping loss-, latency-, and ECN-based congestion control algorithms using minimal hardware resources (just one extra priority) and buffer conservation. Cebinae can, therefore, tolerate a wide range of congestion control algorithms.
- Finally, we develop prototype implementations for both Tofino programmable switches and NS-3 simulator. We use them to show that Cebinae can effectively mitigate unfairness in a wide range of scenarios. Cebinae source code is publicly available at `https://github.com/eniac/Cebinae`.

2 BACKGROUND

Network capacity is a limited resource. Thus, networks rely on congestion control protocols to ensure that the limited capacity is utilized and utilized well. Without them, the network can waste significant resources on packets that will be dropped and their resulting retransmissions; this effect would often severely limit goodput in the early Internet.

Objectives. Traditionally, congestion control algorithms are expected to provide at least two critical properties: efficiency and fairness. Efficiency in this context is typically defined as Pareto-efficiency. A given allocation of flow rates $\{r_1, \ldots, r_N\}$ is Pareto-efficient if and only if an increase in any rate r_i necessitates a decrease in another flow's rate r_j. Fairness has a wider range of possible objectives, e.g., max-min fairness [8, 28], proportional fairness [31], generalized utility [8], or more complex metrics [23, 56]. Among these, max-min fairness often serves as the canonical objective [28, 47, 56] as it achieves Pareto efficiency with the additional desirable property that an increase in any flow's rate necessitates a *decrease* in a *smaller* flow's rate.

The speed at which protocols converge to the above objectives forms a separate axis; however, particularly for fairness, the behavior of protocols after convergence is of paramount importance as persistent unfairness can lead to resource starvation.

Unfairness in the Internet. While modern congestion control algorithms are typically adept at quickly ramping up to efficient utilization of network capacity, they are often incongruously ineffective at achieving fairness. Even when competing with other connections using the same algorithm, differences in RTTs, for instance, can cause arbitrary levels of persistent unfairness.

Between different algorithms a continual push toward faster and more efficient bandwidth exploration/management has led to several well-known instances of fairness violations. TCP Cubic, for example, can outcompete New Reno flows to obtain up to 80% of the shared bottleneck link capacity [44]. More recently, researchers have shown that a single connection of the next-generation protocol TCP BBR ramps up to 40% of link capacity when placed against any number of Cubic or New Reno flows [44, 46, 57].

In-network fairness enforcement. In order to address some of the above weaknesses in end-to-end congestion control, many have proposed mechanisms for in-network fairness enforcement. For example, in fair queuing [39] every flow is assigned to a (conceptually) separate queue. These queues can be serviced in per-bit round-robin order, such that all flows with sufficient demand are guaranteed to receive the same proportion of bandwidth.

Traditionally, implementing this strategy required specialized hardware support for scheduling and dequeuing packets in the correct order. AFQ [47] represented a substantial step forward by demonstrating that it is possible to emulate fair queuing on commodity programmable switches without purpose-built fair-queuing hardware. It does so with a calendar queue approach [48] in which the switch allocates *nQ* queues/priorities representing future time slots of *BpR* bytes. AFQ switches track the bandwidth usage of every active flow. For each incoming packet, they (1) compute the time the packet would have been scheduled in an ideal fair-queuing system and (2) place it in the *BpR* bucket corresponding to the correct time slot. The switch will drop the packet if the target time bucket is more than *nQ* slots in the future.

While effective for local data center clusters running low-latency congestion control protocols, approaches like the above do not scale

[1]Named after capuchin monkeys, the animal species in the first experimental study that showed an understanding of fairness (inequity aversion) in non-humans.

to the Internet and public clouds, which have multiple network bottlenecks, a diverse range of congestion control algorithms, and high RTTs (in addition to being particularly susceptible to issues of unfairness). Consider, for instance, a switch with 10 active flows but with 9 of them bottlenecked at an upstream device. Because every flow in a fair-queuing system has an equal share of capacity[2], *BpR* must satisfy the following condition for every flow:

$$\mathit{buffer\_req} \leq \mathit{BpR} \times \mathit{Nq} \qquad (1)$$

where *buffer_req* is the buffer necessary for the flow's protocol (the bandwidth-delay product in the worst case). Higher latencies, link capacities, and burstiness necessitate higher *Nq* and *BpR* values.

Crucially, the above values need to be configured based on the largest *buffer_req* of any flow in the network, which may be extreme for public networks or those that carry WAN connections. Further, we note that network bandwidth is currently outpacing device memory buffers, putting additional pressure on these hardware requirements. Even with sufficient resources, however, reserving high values of *Nq* impinges on existing prioritization needs, and configuring high *BpR* is directly correlated to unfairness.

3 CONCEPTUAL FOUNDATIONS

In this section, we describe the conceptual foundations that inspire Cebinae's operating principle. In particular, to understand why Cebinae can achieve similar results to fair queuing in a more scalable fashion, we first re-examine the relevant formalism behind the max-min fairness objective in Section 3.1 before discussing its implications in Section 3.2.

3.1 Max-min Fairness

The classic definition of max-min fairness is provided below. We refer readers to [8] for a more complete treatment of the below definitions and proofs.

Definition 1. Let R be the set of all possible flow-rate allocations that satisfy the capacity constraints of the network. An allocation of rates $\vec{r} = \{r_1, \ldots, r_n\}$ in R is "max-min fair" if and only if, for all other allocations $\vec{s} \in R$ and all flows i:

$$s_i > r_i \implies \exists j : (r_j \leq r_i \text{ and } s_j < r_j)$$

In other words, there exists in the other allocations, $\vec{s}$, a smaller flow, r_j, that loses capacity.

Flow allocations that are max-min fair are provably Pareto-efficient and unique under common network assumptions.

The most common method of computing the above allocation is an iterative water-filling algorithm. Intuitively, the algorithm works by initializing all flows as 'unconstrained' with a rate of 0. In every iteration, the algorithm adds an equal amount to all unconstrained flows until at least one link in the network becomes saturated. All flows that traverse the saturated link are now considered 'constrained.' The algorithm iterates until all flows are constrained.

Traditional TCP congestion avoidance share many similarities to the water-filling algorithm. In the ideal case, the rate of all senders is increased by one MSS per RTT until they either satiate their demand or they detect congestion and become 'constrained.' In reality, however, TCP diverges from the above algorithm in numerous and significant ways. Flows are not simultaneously initialized with zero rates, rate increments are not simultaneous nor uniform (depending on RTTs), real applications do not have infinite demand, sending is often based on unacknowledged bytes rather than fixed rates, and connections can have heterogeneous congestion detection methods (e.g., loss, delay, ECN, hybrid, etc.) and increase/decrease algorithms [16]. These discrepancies can explain many of the issues with fairness in the modern Internet.

Fair queuing implements the water-filling framework at each output queue slightly more precisely by (again, ideally) granting each flow a single bit of capacity in round-robin order until it either satiates all flows or saturates the link's capacity.

3.2 The Cebinae Approach

We note that the water-filling algorithm and the provable uniqueness of max-min-fair allocations provide an alternative definition of max-min fairness to Definition 1:

Definition 2. An allocation of rates $\vec{r} = \{r_1, \ldots, r_n\}$ in R is "max-min fair" if and only if, for all flows r_i, there exists at least one bottleneck link for r_i, ℓ, that satisfies both the following properties:

- ℓ is saturated. Specifically, $\mathit{capacity}_\ell = \sum_{j \in L} r_j$, where L is the set of all flows utilizing ℓ.
- r_i is the largest flow. Specifically, $\forall j \in L : (r_i \geq r_j)$.

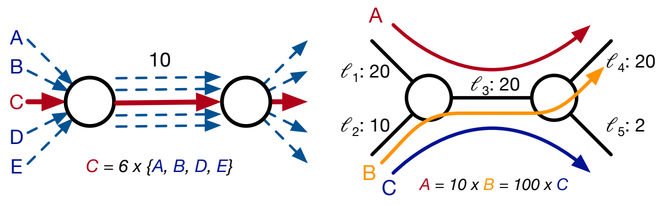
[8] includes a proof of this definition, but a reader can develop an intuition for the reasoning by relating it to the water-filling algorithm, which operates by finding the bottleneck link for each flow and, once found, setting the flow to the 'constrained' state while other flows continue to increase.

Cebinae makes the observation that, aside from being both necessary and sufficient, Definition 2 lends itself to an efficient and distributed verification of the max-min fairness of a network. Specifically, every link can determine the precise set of flows for which it is the bottleneck:

(1) If the link is not saturated, it is not a bottleneck for any flow currently using the link. As the link has additional capacity, any flow can capture more bandwidth without impacting any other flow (bigger or smaller).
(2) If the link is saturated, then for each flow i on the link:
    (a) If i has the largest rate among the locally competing flows, then this link is i's bottleneck. Even if i can capture more bandwidth, it can only do so at the cost of other flows. Note that multiple flows can be bottlenecked by the same link if their rates are equal.
    (b) If i does *not* have the largest rate on the link, then the link is *not* i's bottleneck. i may or may not have a bottleneck link elsewhere in the network.

Crucially, each of the above cases can be differentiated using only local information in the form of an aggregate byte counter (to differentiate cases 1 and 2) and heavy-hitter-only flow-size tracking (to differentiate 2a and 2b). Just as important, it centers around a classification with only two groups: bottlenecked and not bottlenecked. As we will see in Section 4, a two-group classification lends itself to fundamentally more efficient programmable-switch implementations than any approach with more differentiation.

[2]WFQ introduces a weight to the fair share, but a similar argument applies.



(a) Single bottleneck **(b) Multiple bottlenecks**

Figure 2: Two examples of unfairness. One over a single bottleneck and another in a network with multiple potential bottlenecks. In (a), flow A can capture and hold bandwidth 6× as efficiently as all other flows; In (b), flow A can get 10× as much bandwidth as B and 100× as much as C.

At a high level, Cebinae uses the above classification by imposing a limitation on only flows with a bottleneck link (i.e., that have already met or exceeded their fair share). Cebinae prevents these flows from claiming additional bandwidth while others can continue to grow at their expense.

Strawman solution. A naïve version of this approach would be for every router in a network to detect the saturation of their links and impose a token-bucket rate limit on all flows of the maximal size; limits are released when aggregate demand drops below the link's capacity. There are two main issues with this strawman.

The first is that, while the strawman can take an existing max-min-fair allocation and prevent flows from taking additional bandwidth unfairly, it cannot make an already-unfair allocation fair. To illustrate this effect, consider the example in Figure 2a, with a TCP variant that can take and hold 6× as much bandwidth as a competing variant. Given the converged allocation of {1,1,6,1,1}, the strawman solution will correctly limit the aggressive flow's bandwidth because it has already achieved its fair share. Unfortunately, the other flows do not have a mechanism to claim their own fair share and will languish even though they do not have a bottleneck link. Networks can enter such unfair allocations in many different ways, including from previously fair allocations. For example, a similar scenario to Figure 2a can arise when a flow operating in isolation captures the majority of a link's bandwidth, then four new flows join.

The second issue is that modern congestion control algorithms have become diverse, not just in their approaches to rate allocation, but also in the congestion control signals they leverage. Algorithms that ignore loss and focus solely on delay, for instance, may not be responsive to a simple token-bucket filter, or they may treat the loss more seriously than necessary.

Cebinae bandwidth redistribution. Cebinae differs from the above strawman in at least three ways.

First, rather than freezing the bottlenecked flow(s) to their last known rate, Cebinae instead attempts to redistribute a small fraction of the flows' bandwidth; in effect, a tax on the largest flow(s) on each link. Although this tax imposes some overhead on the worst-case throughput of the network, it also ensures that flows that take more than their fair share of the network will not be able to dominate the network forever. In addition, the tax rate τ is a configurable parameter that allows operators to explicitly trade off the convergence speed with the overhead of the approach. In our experiments, we found that tax rates as low as 1% were robust across a wide range of protocols and configurations.

Second, to promote stability and amortize the taxation among flows, Cebinae applies the tax to the maximal-rate flow(s) as well as any flow within δ of the maximum, where δ is another configurable parameter that represents the amount of allowed unfairness, as a fraction of the link's maximal rate.

Finally, to better serve modern congestion control needs, Cebinae adds delay and ECN as additional signals of congestion that trigger before packet loss. It does so with the help of a modified and approximated leaky-bucket filter.

Examples of the Cebinae approach, in context. In order to develop an intuition for how and why Cebinae's binary taxation approach works, we can consider a few concrete examples.

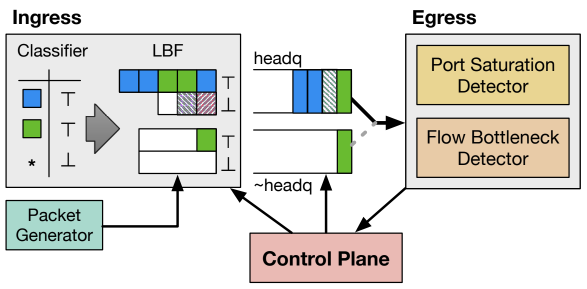
(1) Fair flows on a single bottleneck link: The first and most straightforward is a correct, max-min allocation. For example, take the topology of Figure 2a with a modification where all five flows are homogeneous (same demand, same algorithm, same RTT, etc.). Starting from an empty network, all flows will increase their sending rates equally until the link saturates. At that point, all of the flows should exhibit roughly equal rates. Cebinae will detect that the network is saturated and tax all flows by τ. Eventually, Cebinae will detect that the network is no longer saturated and remove all limits, allowing the flows to reclaim the taxed capacity. Utilization will fluctuate around full capacity but will never decrease by more than τ.

Note that, in principle, Cebinae could detect this case and decline to levy any tax; however, new flows may suffer if the existing flows are more aggressive (at least until Cebinae notices the new unfairness). Cebinae instead chooses to ensure that there is always room for new flows to grow.

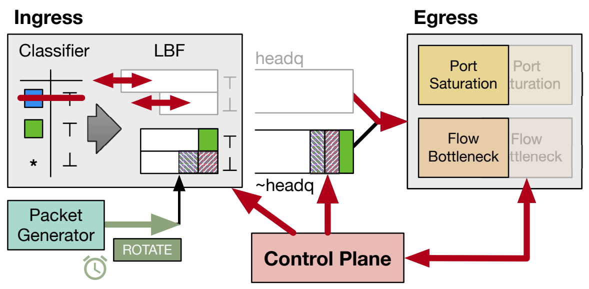
(2) Unfair flows on a single bottleneck link: Now consider the same topology with the scenario introduced in the strawman discussion. Again starting from empty, the aggressive flow will initially acquire six units of capacity, while all other flows only get one. When the link reaches saturation, Cebinae will rate-limit the aggressive flow to a rate of $6(1-\tau)$. The other flows are then allowed to reclaim the taxed capacity. After they do so, the link will again saturate, and Cebinae will further limit the aggressive flow to $6(1-\tau)^2$. Assuming that the more aggressive flow can always claim capacity up to its original rate, the network will converge to max-min fairness in $\frac{\ln(2/3)}{\ln(1-\tau)}$ timesteps, at which point it will proceed to fluctuate in a manner similar to Example (1).

This example demonstrates some of the principal benefits of Cebinae. In particular, that Cebinae enables flows to capture capacity in an underutilized link as quickly as possible—as quickly as they would without fairness augmentation. After they capture the capacity, Cebinae's bandwidth redistribution drives the allocation toward a more fair configuration, regardless of the congestion control algorithms involved.

(3) Unfair flows on multiple bottleneck links: Definition 2 ensures that Cebinae's approach to single-device fairness extends to a network of devices and bottlenecks.



(a) Normal operation



(b) During control-plane modification

Figure 3: An overview of Cebinae's per-router architecture, both (3a) during normal operation and (3b) during a control-plane-/packet-generator-aided reconfiguration. In 3a, the bottlenecked flows (⊤) have exceeded their headq allocation, but the other flows (⊥) have not; headq has higher priority than ¬headq. In 3b, bold red markings indicate the changes made by the control plane using serializable transactions.

Param.	Description
δ_p	The threshold for the port saturation status.
δ_f	The threshold for the flow bottleneck status.
τ	The Cebinae tax rate.
P	The number of dT periods before we recompute utilization and rate limits.
L	Control-plane reconfiguration deadline.
dT	The time allocated to each physical bucket (2^n).
vdT	The time in each virtual bucket (2^m, $m < n$).

Table 1: Configurable parameters of Cebinae. We discuss expected values of these parameters in Section 4.4.

For example, consider the case in Figure 2b, with three flows of greatly differing ability to acquire and hold bandwidth. The initial allocation of this network would have the largest flow, A, take ∼18 units of capacity, while B and C take ∼1.8 and ∼0.18 units, respectively. In this allocation, A is bottlenecked on ℓ_3. Again assuming that A retains its competitive advantage, a tax rate of $\tau = 1\%$ would reduce it to $\sim 18 * 0.99 = 17.84$, leaving B and C free to increase to $\sim 1.8 + 0.18 * {}^{10}/_{11} = 1.97$ and $\sim 0.18 + 0.18 * {}^{1}/_{11} = 0.197$, respectively.

The above process will continue until B and C can occupy an aggregate of 10 units of capacity. At that point, flow A will fluctuate around its optimal based on the actions of its bottleneck link (ℓ_3), and ℓ_2 will start to consider B as constrained. The system has still not converged, however, as C has no bottleneck link. Instead, ℓ_2 needs to tax and redistribute capacity from B to C in a manner similar to Example (2). Eventually, A will be bottlenecked by ℓ_3, B by ℓ_2, and C by ℓ_5, achieving global max-min fairness.

Comparison to fair queuing. As Example (3) illustrates, Cebinae does not guarantee perfect fairness at every instance in time as proceeds from its taxes are not redistributed equally. Instead, Cebinae focuses on mitigating unfairness by preventing flows that take more than their fair share from continuing to take more than their fair share. With stable demands, the system will reach steady state in bounded time, though the steady state may involve oscillations between a few fixed configurations, e.g., the oscillations around full capacity in Example (1). The advantage of this approach is simpler implementation and improved scalability, especially in the presence of numerous and/or bursty flows.

4 THE DESIGN OF CEBINAE

Note that, as with prior work, the above analysis assumes protocols that respond to capacity limitations [19, 43]. A blind UDP flow, for instance, may unnecessarily waste network bandwidth before being delayed and dropped by a downstream Cebinae router. Addressing this requires network-level admission control, which is orthogonal to our core mechanism.

The design of Cebinae is guided by the following principles.

Be agnostic to congestion control algorithms. Cebinae should not assume anything about the specifics of the congestion control algorithms that are using it, except that they make some effort to control congestion. This includes but is not limited to assumptions about congestion signals, buffer utilization patterns, and burstiness.

Minimize hardware overheads. Cebinae also seeks to maximize usable buffer, minimize the requirements on queues and priority levels, and incur no recirculation. For a system that needs to inject delay with FIFO queues without recirculation, two priority levels is the provable minimum, and Cebinae achieves it.

Never make unfairness worse. Finally, Cebinae should ensure that no flow is taxed that does not deserve it. This means bottleneck detection and classification should not be prone to false positives, e.g., because of hash collisions. False negatives, on the other hand, are tolerable as those flows will simply compete with others exactly as they do today.

Figure 3a illustrates Cebinae's high level architecture. As previously mentioned, Cebinae does not require changes to end hosts or explicit coordination between different network devices. Instead, Cebinae routers can be considered in isolation, with each router containing three components:

- An egress-pipeline flow-rate cache that tracks port saturation and bottleneck flow statuses at a fine granularity.
- An ingress-pipeline flow scheduler that injects delay/loss into bottlenecked flows to cap and redistribute their bandwidth.
- A low-latency control plane agent that records flow rates and dynamically adjusts bottleneck flow membership and sending rate limits.

To implement the approach described in the previous section, the control plane—at a fine granularity—polls the egress pipeline for

```
last_headq = headq
headq = DP.get_headq()
if headq == last_headq: continue
busy_sleep(dT - L)
configure_rates_and_swap_queue_priorities()
if ++recomputation_counter % P != 0: continue

last_port_bytes = port_bytes
port_bytes = DP.get_port_bytes()
flow_bytes = DP.get_flow_bytes()
for p in PORTS:
    byte_count = port_bytes[p] - last_port_bytes[p]
    if byte_count / CAPACITY[p] < 1 - $\delta_p$:
        DP.set_unbottlenecked(p)
        continue

    max_flow_bytes = max(flow_bytes[p])
    bottleneck_bytes = 0
    for f in flows:
        if flow_bytes[p][f] >= max_flow_bytes * (1 - $\delta_f$):
            DP.set_bottlenecked(f)
            bottleneck_bytes += flow_bytes[p][f]
        else:
            DP.unset_bottlenecked(f)
    bottleneck_bytes *= (1 - $\tau$)

    DP.set_⊤_rate(p, headq, bottleneck_bytes/dT)
    DP.set_⊥_rate(p, headq,
                (CAPACITY[p] - bottleneck_bytes)/dT)
```

Figure 4: Pseudocode for Cebinae's control plane agent. Some optimizations and details omitted for readability. Note that all data-plane reads are serializable, as are data-plane writes.

```
// vdT_mask = ¬((1<<log2(vdT)) - 1) [usually all 1s]
f = pkt.egress_port << ⊤.contains(pkt.flow)

if pkt.type == ROTATE:
    bytes[f] = max(bytes[f] - pkt.last_rate * dT, 0)
    base_round_time += dT
    headq = ¬headq
    drop pkt

elif pkt.type == NORMAL and egr_saturated(pkt):
    if current_time >= (round_time + vdT):
        round_time = current_time & vdT_mask
    relative_round = (round_time - base_round_time) / vdT

    if relative_round < dT/vdT:  // in headq
        aggregate_size =
            rate[headq][f] * relative_round * vdT
    elif relative_round < 2*dT/vdT: // in ¬headq
        aggregate_size = rate[headq][f] * dT +
          (relative_round-dT/vdT) * rate[¬headq][f] * vdT
    // else: should never happen

    bytes[f] = max(bytes[f], aggregate_size) + pkt.size
    past_head = bytes[f] - rate[headq][f] * dT
    past_tail = past_head - rate[¬headq][f] * dT
    // optionally mark ECN bits

    if past_head <= 0:
        enqueue(headq)
    elif past_tail <= 0:
        enqueue(¬headq)
    else:
        drop pkt
```

Figure 5: Pseudocode for the data-plane implementation of Cebinae's leaky-bucket filter. Keywords, constants, builtins, and stateful variables are color-coded.

information on link and flow utilization, then manages the ingress flow scheduler accordingly. To ensure both speed and transactional isolation, the system is built on the Mantis [60] switch-reaction framework. While Cebinae operates canonically on the basis of flows, the mechanism can be generalized to alternative methods of traffic attribution [10, 11, 37, 45] through modifications to the unit identifiers in its rate accounting, membership management, and rate limiters. In this section, we describe the details of Cebinae's implementation in the context of a bottleneck flow's typical life-cycle. Throughout, we will refer to Table 1 for relevant parameters.

4.1 Detecting Port Saturation

In the flow-bottleneck detection procedure described in Section 3.2, the first decision depends on whether the target link is saturated (as unsaturated links do not act as a bottleneck for any flow). To accurately determine saturation, Cebinae tracks utilization at the egress pipeline, where it maintains a simple transmit byte counter for each port, stored as elements in a register array. Note that the Mantis framework requires a shadow copy of each element to guarantee consistency with other data-plane reads/writes.

The Cebinae control plane agent periodically samples the register array and, without resetting the counters, computes the observed difference from the previous iteration to find the utilization during the last interval, dT. If the utilization is above $(1 - \delta_p) \cdot capacity$, the link is considered saturated.

4.2 Detecting Bottlenecked Flows

If there is a positive determination of port saturation, Cebinae then determines which flows are bottlenecked by the current link. From Definition 2 and Section 3.2, this is simply the flow(s) on the link with the *maximum* observed rate.

Cebinae detects these bottleneck flows with a *heavy-hitter cache* in the egress pipeline. The goal is to accurately track both (*a*) the size of the largest flow and (*b*) the IDs of any flows of similar size without false positives. Cebinae can leverage any of the recently proposed techniques that satisfy both; our prototype adapts the HashPipe mechanism [49].

Compared to prior heavy-hitter caches [7, 49], Cebinae's data structure minimizes memory management overhead. Prior caches manage memory *actively*, with eviction logic in the data plane that tries to avoid replacing active heavy hitters. This eviction logic is expensive, for example it often requires packet recirculation [7]. Cebinae eliminates such overheads by *passively* managing cache memory. After every interval, *all* entries are evicted to the control plane, giving every active flow another chance to claim an entry. Intuitively, active heavy hitters are the most likely to (re)claim their entry, simply because they send the most packets.

Concretely, Cebinae's heavy-hitter cache uses multiple stages of hash-mapped flow tables. A packet arriving at a stage is hashed to an entry, and it either increments its byte counter (if the entry is unused or is for the packet's flow) or proceeds to the next stage (if the entry is already used by another flow). If there is no room for a packet in any of the stages, it is simply not counted. The entire data structure is polled and reset by the control plane (ideally in a serializable fashion [60]) after every interval dT. Cebinae finds the maximum byte counter, c_{max}, of any flow and declares flows as 'bottlenecked' if their byte counts c_i satisfy $c_i \geq c_{max} \cdot (1 - \delta_f)$. Partial pseudocode for this process is in Figure 4.

4.3 Coarse-grained Rate Enforcement

When a port is saturated and the set of locally bottlenecked flows identified, Cebinae proceeds to cap and tax the bottlenecked flows in the ingress pipeline. Cebinae's mechanism for enforcing fairness draws inspiration from calendar-queue-based leaky bucket filters [48]. We refer readers to [48] for details, but at a high level, calendar queues are capable of injecting loss, latency, and ECN bits (based on virtual queue lengths) to passing flows by scheduling packets in current or future physical queues and periodically rotating queue priorities so that the drained queues can be reused. Cebinae, however, makes several innovations to ensure practicality and scalability.

In particular, instead of trying to enforce static, per-flow fairness with a deep set of future calendar queues, Cebinae implements dynamic rate enforcement by tracking only two flow groups—bottlenecked (abbreviated as $\top$) and unbottlenecked (abbreviated as $\bot$)—and using only two total queues/priorities (headq and ¬headq). These changes lead to material and fundamental differences in the approach's design, implementation, and scalability. They also introduce significant challenges to membership and rate changes, but we discuss solutions to those below.

Figure 5 includes the pseudocode for Cebinae's rate enforcement. For each packet, it takes the rate allocation of the target flow group ($\top/\bot$) and computes the expected send time of the packet [lines 23–25]. Cebinae translates this send time to a time bucket and associated physical queue. If the send time is past the current queue, Cebinae delays the packet (put in a lower-priority queue) [line 31]; if it is past the currently available queues, Cebinae drops the packet [line 33]. The control plane periodically rotates the priorities of old, empty queues to continually generate lower-priority queues for future traffic.

Reducing the required priority levels to just two. As previously mentioned, Cebinae's restriction on queue utilization introduces several challenges. For example, with only two queues, the baseline implementation may not have sufficient time to drain an old queue before it is needed again. This situation can result in the switch dropping all incoming packets until the rotation is completed. Cebinae addresses this with two techniques.

The first is a mechanism for *virtual pacing* within a physical queue. We observe that the primary bottleneck for queue rotations is the worst-case queue drain time. In particular, with the baseline implementation and sufficient hardware buffers, if flows were to send their full rate allocations ($\sum_{r \in Rates} r * dT = BW * dT$) at the end of the round, it would take dT, i.e., the full round to drain the queue completely. Instead, Cebinae limits these 'catch-up' bursts by creating virtual rounds within each physical round. Virtual rounds in the same physical round share a priority and do not affect the worst-case burst allowance of the system; however, they ensure that at the end of a round, the previous queue will drain within vdT. With the extra time, Cebinae can perform operations on the previous queue before it is needed again.

The second mechanism is a strict-real-time technique for queue rotation/modifications. Queue rotations are triggered by precisely tuned hardware packet generators that are found on modern programmable switches. Updates of `base_round_time` are synchronized to the timing of these packets. In fact, Cebinae uses the first ROTATE packet to set its initial value (by stepping backward by vdT+L) to bootstrap the time origin of the state machine. Even the control plane is synchronized to these packets—it detects their `headq` flips and executes `asm pause` instructions until it can be sure that `headq` is again stale and drained.

To adhere to these timings, the control plane has a configurable L seconds to complete the queue priority swap and any other necessary operations. If the control plane cannot complete all operations in L time, it will truncate its tasks (e.g., not moving a flow $\top \leftrightarrow \bot$) to make its deadline; it can make the change in the subsequent round. Figure 6 depicts the precise timeline of all operations.

This real-time approach is markedly different from the event-driven model of other calendar-queue systems, and it is what enables our 2-queue approach. We note that it also imposes tighter efficiency bounds and eliminates the need for packet digests and recirculation overhead.

Reducing the required flow tracking groups to just two. Cebinae also reduces the granularity of flow tracking from needing to track the utilization of all flows to only tracking the utilization of two flow groups. This simplification saves SRAM, eliminates the possibility of hash-collision unfairness, and also substantially reduces the full-utilization *BpR* requirement of Equation (1) (both by reducing the denominator of the expression and leveraging statistical multiplexing effects on *buffer_req* [4]). Flows will still compete within their respective groups just as they do today; rather, the goal is to nurture $\bot$ flows in the absence of the bottlenecked flows.

We note that the data plane still needs to identify flows' bottleneck status; however, the information is only for $\top$ flows and is static (i.e., implementable in match-action tables, where hash collisions with $\bot$ flows are not an issue).

Supporting dynamic rate changes. To implement the Cebinae tax, a key feature is the ability to continuously adjust the rate allocations. These adjustments, however, need to respect the real-time guarantees developed above. To see why this is challenging, consider a rate change 80%/20%⟶20%/80% ($\top/\bot$). A naïve implementation might allow a $\top$ flow to consume 80% of the link bandwidth before the rate change and a $\bot$ flow to consume 80% after the change.

Cebinae addresses this problem by only allowing changes on the boundaries between physical queues. Specifically, Cebinae fixes each logical queue's rates as soon as it becomes available for scheduling—the queue's rates can only change when it is the fully drained `headq`. Cebinae integrates these heterogeneous rates [lines 15–20].

Supporting dynamic membership changes. In addition to supporting dynamic rate changes, Cebinae also needs to support dynamic membership changes as flows become (un-)bottlenecked by the current link. We must take care when implementing these membership changes to avoid spurious packet reordering, especially for flows that we wish to grow.

As an example, consider a flow, f, that is transitioning from [$\top \rightarrow \bot$]. Further, assume that the $\top$ flow group has exhausted its allocation in `headq` (and is placing packets in `¬headq`), while the $\bot$

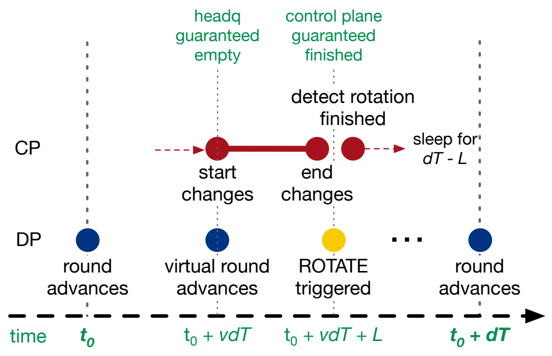


Figure 6: The timeline of Cebinae's control plane (CP) and data plane (DP) actions during each time bucket, *dT*. All priority and configuration changes are completed during the period marked by the solid red line.

flow group still has capacity in headq. After the transition, newer packets of f may be placed in a higher priority queue than existing packets from f.

Cebinae prevents this scenario by noting that, for a brief period (i.e., $t_0 + vdT$ to $t_0 + vdT + L$ in Figure 6), only one hardware queue contains packets. Thus, membership changes during this period will not result in any reordering. We note that this window does not align with the transition periods for rate allocations, but aggregate rates will still be respected (and worst-case drain time maintained). Further, any observed discrepancies in rate allocation can be corrected in the next dT round (by over/under-allocating bandwidth).

Supporting phase changes. The above strategy may not be efficient enough for a wholesale change of membership/rates involved in a phase change between port saturation and non-saturation. Cebinae, therefore, implements phase changes with a separate mechanism (omitted from Figure 5). During the saturated phase, the Cebinae data plane will track a third per-port byte counter (`total_bytes[]`) that is computed in the same way as `bytes[]`. The counter will not be used to filter packets until the port becomes unsaturated, when the control plane (during its configuration period) will flip a boolean flag and begin applying the `total_bytes[]` filter to all incoming traffic immediately. In this way, the membership/rate change is atomic, prevents reordering, and maintains the real-time queue drain guarantee.

Transitioning from unsaturated⟶saturated is also executed atomically during a control-plane configuration period; however, in this case, the first packet of each flow group will set `bytes[f] = total_bytes[port] * (rate[f] / BW[port])`.

4.4 Configuring Cebinae

We now discuss the parameters of Table 1. Several of these could benefit from an understanding of the network traffic characteristics (e.g., skewness among flows) or an adaptive reconfiguration based on Cebinae's real-time measurements, but we note that conservative values for all of them result in a correct implementation that eventually mitigates the unfairness caused by $\top$ flows (though perhaps slowly).

Utilization thresholds ($\delta_{p/f}$). In Cebinae, links are marked saturated if they are within δ_p of their capacity over the measurement period. Flows are marked as bottlenecked if they are within δ_f of the maximum flow's rate. In both cases, $\delta_{p/f}$ determines Cebinae's aggressiveness, with small values leading to only a few taxed flows in the most heavily congested networks and large values leading to faster redistribution. We anticipate that smaller values (e.g., 1%) will be more common.

Tax rate (τ). τ determines how much bandwidth Cebinae attempts to redistribute in each protocol iteration. The primary effect is to adjust the rate of convergence to fairness, with higher τ leading to faster convergence but more potential instability. Like the δs, we anticipate relatively small τs, which still promote fairness but minimize overheads. Again we find that 1% is a robust value for a wide range of configurations.

Recomputation period (P). Utilization and taxes are computed over P, a discrete time window. P is defined in terms of dT and should be an integer multiple thereof. To be resilient to burstiness, P should be large enough to capture typical-RTT-timescale effects, which provide a lower bound for utilization tracking. Larger values of P slow the convergence time of the system but are, again, safe.

Control plane deadline (L). The parameter L is constrained by two factors. The first is the need to rotate queues every dT, necessitating $L \leq dT - vdT$. The second is its role as a deadline for control plane actions. For the latter, we can compute the theoretical maximum by examining the number of registers and table entries that must be read/written in every iteration. In practice, we can reduce this lower bound by considering only typical amounts of membership change with outliers handled in subsequent rounds.

We can reduce it even further by taking advantage of the underlying two-phase updates [60] and P, the recomputation period. Specifically, if the *actual* control-plane computation time, $\hat{L} \leq (P-1) \cdot dT \simeq RTT$, then all changes can be made in the shadow copy before they are needed, reducing the effective L parameter to zero. Priority swaps should still be applied every dT.

Virtual bucket duration (vdT). For vdT, its primary effects are to limit 'catch-ups.' It also has an effect on the lower bound of dT through Equation (2). Lower is better, and the ideal setting is the minimum precision of the data plane clock.

Physical bucket duration (dT). Finally, dT is an important parameter that determines the bytes allowed into the two physical queues/priorities. In general, lower values enforce more aggressive control because Cebinae's relaxations to fair queuing mean that loss and delay are only injected at dT boundaries (although, again, higher values are safe and approximate the behavior of today's networks).

A lower bound is given by a modified version of Equation (1). In particular, while Equation (1) ensures that flows in AFQ have sufficient virtual buffer space to satisfy their protocol-specific needs, Cebinae goes one step further. Deriving from the design goals of Section 4, we wish to ensure that for a flow group with sufficient allocation, *all* of the switch buffer is available at *all* times, regardless of connection sending patterns. This guarantee should continue to hold even during the period in which new packets can only be placed in ¬headq (i.e., $[t_0, t_0 + vdT + L]$). Said differently, even if a

				Throughput [Mbps]			Goodput [Mbps]			JFI		
Btl. BW	**RTTs [ms]**	**Buf. [MTU]**	**CCAs**	**FIFO**	**FQ**	**Cebinae**	**FIFO**	**FQ**	**Cebinae**	**FIFO**	**FQ**	**Cebinae**
100 Mbps	{20.8, 28}	250	{NewReno:2, NewReno:8}	98.95	95.62	95.92	95.35	92.16	92.44	0.740	0.982	0.999
100 Mbps	{20.4, 40}	350	{Cubic:8, Cubic:2}	98.96	98.95	98.00	95.37	95.37	94.45	0.539	1.000	0.980
100 Mbps	{20.4, 60}	500	{Vegas:2, Vegas:8}	98.88	98.83	98.88	95.29	95.24	95.29	0.873	1.000	0.993
100 Mbps	{200}	1700	{NewReno:16, Cubic:1}	98.28	90.99	94.53	94.38	87.61	91.02	0.446	0.995	0.925
100 Mbps	{100}	850	{NewReno:16, Cubic:1}	98.72	91.45	95.58	95.11	88.10	92.08	0.857	0.998	0.960
100 Mbps	{50}	420	{NewReno:16, Cubic:1}	98.90	93.86	95.37	95.30	90.45	91.90	0.936	0.999	0.993
100 Mbps	{50}	420	{Vegas:16, Cubic:1}	98.90	98.90	95.47	95.30	95.30	91.99	0.096	1.000	0.988
100 Mbps	{100}	850	{Vegas:16, NewReno:1}	98.71	97.77	95.67	95.07	94.19	92.16	0.093	0.999	0.985
100 Mbps	{100}	850	{Vegas:128, NewReno:1}	98.88	98.74	97.45	95.26	95.10	93.88	0.189	0.966	0.976
100 Mbps	{60}	500	{Vegas:8, NewReno:8, Cubic: 2}	98.87	98.02	96.52	95.27	94.45	93.00	0.510	0.991	0.973
1 Gbps	{5}	420	{NewReno:32, Cubic:8}	989.8	989.8	985.4	954.0	954.0	949.7	0.844	0.988	0.955
1 Gbps	{10}	850	{Vegas:128, Cubic:1}	989.8	989.8	968.0	954.0	954.0	932.9	0.048	0.966	0.953
1 Gbps	{10}	850	{Vegas:1024, Cubic:2}	989.8	989.8	949.2	953.6	953.6	914.1	0.275	0.833	0.846
1 Gbps	{50}	4200	{NewReno: 128, BBR: 1}	988.7	923.6	981.6	952.7	890.0	945.8	0.992	0.975	0.990
1 Gbps	{50}	4200	{NewReno: 128, BBR: 2}	988.9	953.9	979.9	952.8	919.2	944.2	0.951	0.963	0.981
1 Gbps	{50}	21000	{NewReno: 128, BBR: 2}	988.8	953.9	963.8	952.7	919.2	928.7	0.773	0.963	0.936
1 Gbps	{100}	8350	{NewReno: 128, BBR: 2}	986.9	938.2	956.3	950.7	903.9	921.1	0.884	0.968	0.967
1 Gbps	{10}	850	{Vegas:64, NewReno:1}	989.8	989.8	976.2	953.8	954.0	940.7	0.042	0.967	0.976
1 Gbps	{100}	8500	{Vegas:4, NewReno:128}	986.9	917.6	957.3	950.8	884.1	922.2	0.946	0.970	0.971
1 Gbps	{100, 64}	8500	{Vegas:4, NewReno:128}	988.4	941.1	959.8	952.4	906.8	924.7	0.956	0.970	0.964
1 Gbps	{100}	8500	{Vegas:8, NewReno:128}	987.0	936.1	964.4	950.8	901.8	929.0	0.921	0.968	0.969
1 Gbps	{10}	850	{Vegas:128, BBR:1}	989.8	989.8	987.3	954.0	954.0	951.5	0.886	0.965	0.985
1 Gbps	{100}	8500	{Bic:2, Cubic:32}	985.1	960.3	952.6	944.9	924.9	911.3	0.799	0.999	0.946
10 Gbps	{50, 44}	41667	{NewReno:128, Cubic:16}	9876	9705	9780	9514	9352	9420	0.917	0.969	0.968
10 Gbps	{28, 28}	25000	{NewReno:128, Cubic:128}	9891	9856	9787	9532	9498	9432	0.863	0.942	0.952

Table 2: Results for a range of different network configurations that include varying bandwidth, RTT, and congestion control algorithms. The cases cover intra- and inter-CCA unfairness, RTT unfairness, loss-based protocols (NewReno, Cubic), and delay-based or hybrid algorithms (Vegas, BBR).

large burst of packets arrives just before $t_0 + vdT + L$, the switch should be able to admit as many packets as its physical buffer can hold. The resulting constraint is:

$$\begin{aligned}(t_0 + dT - (t_0 + vdT + L)) \cdot BW &\geq buffer \\ (dT - (vdT + L)) \cdot BW &\geq buffer \qquad (2)\end{aligned}$$

Accounting for the aforementioned optimizations to vdT and L, the bound on dT approaches $dT \geq \frac{buffer}{BW}$.

Summary. When configuring a Cebinae router, operators should carefully tune δ_p, δ_f, and τ to trade off efficiency and degree of unfairness mitigation, with a preference for conservative values. The remainder of the parameters can be set from network characteristics. With a single-round control plane: P should be set to capture the maximum RTT of the network; vdT should be set to the precision of the dataplane clock (e.g., 1 nanosecond); L can be decided based on the typical flow-membership churn; and $dT \geq \frac{buffer}{BW} + vdT + L$. When the control plane can span multiple rounds, $dT \geq \frac{buffer}{BW} + vdT$.

5 EVALUATION

We implement a hardware prototype of Cebinae on a testbed that consists of a Wedge100BF-32X switch (emulating two switches for a dumbbell topology) and a set of servers with Mellanox ConnectX-4 NICs. To enable more extensive evaluation, precise tracing, and fair comparison with its alternatives under a wide range of background conditions, we also implement Cebinae as ns-3.35 [3] traffic control layer modules attached to L2 NetDevices (encoding the Cebinae state machine and data-plane, control-plane operations described in Figures 4 to 6). In total, the core implementation consists of 2,000 lines of P4 and Lucid [50] for the data plane, 1,500 lines of C++ for the control plane, and around 1,500 lines of C++ for the NS-3 module. The code and scripts are open-sourced for reproducibility.

Methodology and overview. We evaluate Cebinae over a wide range of network conditions (RTT, speed, and buffer size) and combinations of congestion control algorithms. As comparison points, we replicate all experiments with FIFO drop-tail queues and fair queues. For the latter, we utilize NS-3's FQ-CoDel variant [1, 2], which combines both Deficit Round Robin (DRR) (for fair queuing over a large number of queues) and CoDel (an Active Queue Management scheme). We change the default 1024 queues in FQ-Codel to $2^{32} - 1 = 4,294,967,295$ to ensure an ideal per-flow queue.

Similar to prior work on congestion-control unfairness [25, 44, 46, 54, 57], we choose a representative set of popular congestion control algorithms (CCAs) used in the Internet today: NewReno [43], Cubic [24], Vegas [52], BBRv1 [15]. NewReno represents the classic approach to loss-based congestion control, Cubic is the current default algorithm on Linux and Windows Server (Bic [59] being its older version), TCP Vegas represents a classic combined use of latency/loss, and BBR is a next-generation protocol recently proposed by Google that eschews loss to directly compute the bandwidth and delay of the network.

5.1 Cebinae Is Agnostic to the CCA

Table 2 shows the results for a sweep of network conditions. In each case, we measure fairness for a set of long-lived, heterogeneous flows competing over a single bottleneck with infinite demand [54, 56]. While Cebinae is effective across a range of configurations, as a test of its robustness, we fix the configuration to a relatively conservative set of parameters: $\delta_p = 1\%$, $\tau = 1\%$, $\delta_f = 1\%$. We examine several metrics: average bottleneck link throughput, average application goodput (representing efficiency), and Jain's

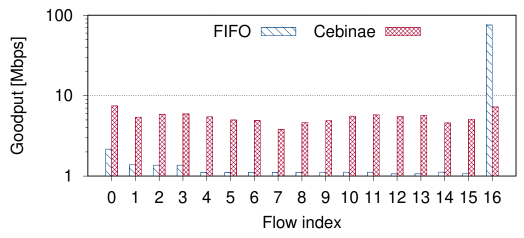


Figure 7: The goodput for 16 TCP Vegas flows (0–15) and a NewReno flow (16) competing over a 100 Mbps bottleneck with and without Cebinae. Cebinae moves the skewed unfair allocation towards fairness with little efficiency impact.

Fairness Index (JFI) [27] over the entire evaluation period (100 s). We delve into some of these results below.

We note that different congestion control algorithm configurations, while having the same application data rate demand per flow, could lead to different levels and types of unfairness, Cebinae can reduce the degree of unfairness regardless. Similarly, we also find Cebinae robust to different configurations of the *same* congestion control algorithm because fundamentally, Cebinae treats them as a black box and is agnostic of their concrete implementation.

5.2 Cebinae Mitigates Unfairness

From Table 2, we can see that across a wide range of congestion-control configurations and latencies, Cebinae improves the network's fairness. Cebinae achieves this fairness augmentation with little-to-no efficiency impact as Cebinae enforces penalties only when the link is saturated and only towards $\top$ flows. The remaining impacts become negligible with less aggressive tax rates or $\bot$ flows that can quickly reclaim available bandwidth headroom. In the following sections, we dive deeper into various aspects of Cebinae's performance.

Preventing aggressiveness. One egregious case is when 16 Vegas flows (delay-based) compete with 1 NewReno flow (loss-based) over a 100 Mbps link, each flow having equal RTTs and demands. With FIFO, the single NewReno flow takes over around 80% bandwidth, as shown in Figure 7, resulting in persistent unfairness and an average JFI as low as 0.093. Cebinae, instead, improves the fairness by redistributing the bandwidth of the NewReno flow and allowing the rest of the flows to grow their share; the resulting average JFI is an order of magnitude higher at 0.984. Figure 8a illustrates a similar case when 128 NewReno flows compete with 2 BBR flows over a 1 Gbps link with equal RTTs and demands. Cebinae taxes the excessive bandwidth claimed by the BBR flows and improves JFI from 0.774 to 0.936.

Mitigating starvation. Another case is when 128 NewReno flows compete with 4 Vegas flows over a 1 Gbps link with the same per-flow demand and RTTs of 100 ms and 64 ms, respectively. While the initial JFI provides seemingly high values (0.956), this is a result of the fact that the majority of NewReno flows (with their dominating flow count) get a relatively fair allocation. This masks the unfair allocations towards the 4 Vegas flows, as depicted by the detailed goodput distribution in Figure 8b. Cebinae mitigates the starvation

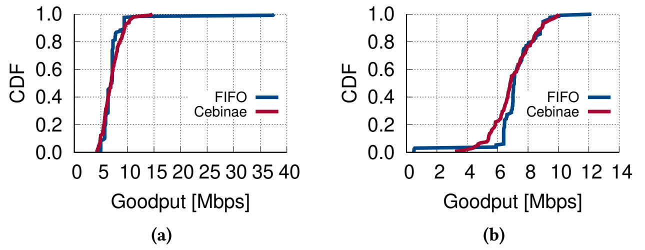


Figure 8: (a) 128 NewReno v.s. 2 BBR flows over 1 Gbps link where Cebinae prevents biased rate allocation towards aggressive flows. (2) 128 NewReno v.s. 4 Vegas flows 1 Gbps link where Cebinae mitigates the starvation effects.

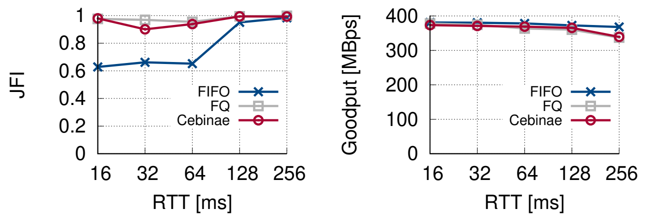


Figure 9: Cebinae mitigates unfairness upon various degree of RTT asymmetries for Cubic flows over a 400 Mbps link.

effect towards the 4 Vegas flows and further improves JFI from 0.956 to 0.964, again, with little efficiency impact.

Reducing RTT unfairness. Cebinae can also mitigate unfairness due to RTT differences, as shown in Table 2. To that end, we measure the degree of fairness when 4 Cubic flows compete with another 4 Cubic flows sharing the same 400 Mbps bottleneck link and 3 MB switch buffer (similar to the setting in [24]). The first group has a fixed RTT of 256 ms and the other group of flows has RTTs ranging from 16 ms to 256 ms, resulting in an asymmetry ratio of up to 16×. Figure 9 shows that Cebinae can mitigate the unfairness due to RTT variances with minimal efficiency degradation.

5.3 Cebinae Pushes Towards Max-min Fairness

The prior settings have shown Cebinae's effectiveness in mitigating unfairness. To be clear, even with a fixed set of flows and infinite demand, the port saturation status, flow bottleneck membership, and rate accounting in Cebinae change dynamically as a result of underlying bursty packet arrivals and bandwidth reshuffling (e.g., when the $\top$ flows react aggressively to penalties and $\bot$ flows ramp up, leading to $\top$ membership migrations), as in Figure 1.

Figure 10 illustrates the JFI time series for a scenario when a set of Vegas flows reach a stable state, then a NewReno flow and a Cubic flow join at around 5 s and 25 s, respectively. Without Cebinae, the system enters an unfair state. Cebinae, however, pushes the network towards a fairer direction and mitigates the unfairness.

We also examine Cebinae's effects under a multi-bottleneck setting. Concretely, 8 NewReno flows traverse 3 switches contending with 2 Bic, 8 Vegas, and 4 Cubic flows at 3 separate 100 Mbps bottlenecked links in a 'Parking Lot' topology [8, 29]. All flows have the same application data rate. Figure 11 depicts the average goodputs

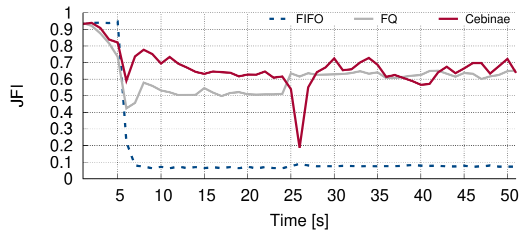


Figure 10: JFI time series (measured per second) for goodput starting from time 0 (stable state of 32 Vegas flows). A NewReno and a Cubic flow arrives at around 5 s and 25 s respectively.

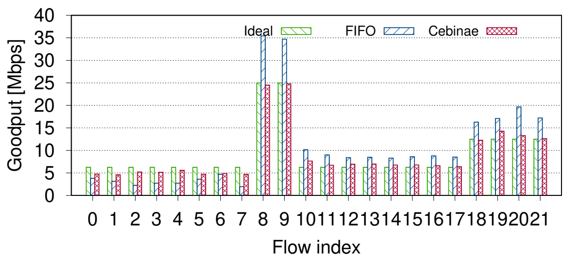


Figure 11: 8 NewReno flows (0–7) contend with 2 Bic (8–9), 8 Vegas (10–17), and 4 Cubic flows (18–21) over 3 bottleneck links. Cebinae promotes the JFI of the network $0.852 \rightarrow 0.978$.

$\{r_i\}$ across the entire evaluation period (100 s) with and without Cebinae compared with the ideal global max-min fair allocation values $\{\hat{r_i}\}$. Initially, the NewReno flows get unfair goodputs due to their larger number of hops, RTTs, and the more aggressive protocols along its path, Cebinae can move the network towards max-min fairness by taxing the aggressive flows and releasing headroom for other flows. Here, JFI quantifies the distance to the ideal allocation per max-min fairness criterion [26, 30]: JFI = $\frac{(\sum x_i)^2}{\sum x_i^2}$ where $x_i = \frac{r_i}{\hat{r_i}}$.

5.4 Cebinae Is Robust to Its Parameters

We explore the sensitivity of Cebinae to its parameters. In general, we found that Cebinae is remarkably robust. While the specific relationship is a function of the workload pattern, network conditions, and the behaviors of edge CCA algorithms as a whole, we find the parameters' trade-offs to be straightforward to reason about due to their explicit physical semantics (Section 4.4).

In particular, we examine the JFI and application goodput of several configurations in a scenario involving 16 NewReno flows competing against 1 Cubic flow. As the parameters jointly determine the aggressiveness of Cebinae operations (the transaction frequency, the set of target flows to regulate, and the degree of penalties), we vary δ_p, δ_f, and τ together and examine their effects.

Figure 12 shows that Cebinae generally improves fairness compared to FIFO, and it provides comparable fairness to ideal fair queuing. As expected, the application goodput decreases as a function of the aggressiveness of the parameters. In particular, it drops sharply especially as the threshold crosses the fair share of the flows. In the extreme case where all thresholds are set to 100% (an unrealistic parameter set), flows will always be marked as bottlenecked regardless of their status and their rate will be limited to 0.

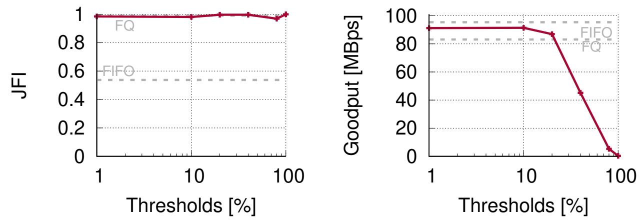


Figure 12: JFI and application goodput as a function of the thresholds: δ_p, δ_f, and τ.

Cache stages	Pipeline stages	PHV	SRAM	TCAM	VLIW instrs.	Queues
1	11	937b	2448KB	15KB	89	64
2	11	1042b	4096KB	34KB	93	64

Table 3: Cebinae data plane resource usage on a 32-port Tofino switch.

Overall, this confirms our intuition that conservative parameters are generally preferable as they mitigate persistent unfairness via taxing the dominating $\top$ flows while exerting minimal disturbance to efficiency.

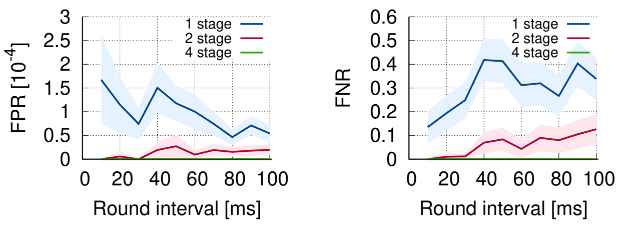
5.5 Cebinae Resource Usage Scales

Finally, Table 3 lists the total resource usage of the Cebinae data plane with both one-stage and two-stage egress flow-cache configurations (4096 slots per port per stage). Cebinae's resource consumption is less than 25% for all types of compute and memory resources, which is less than HashPipe [49] for equivalent performance (in addition to not requiring recirculation).

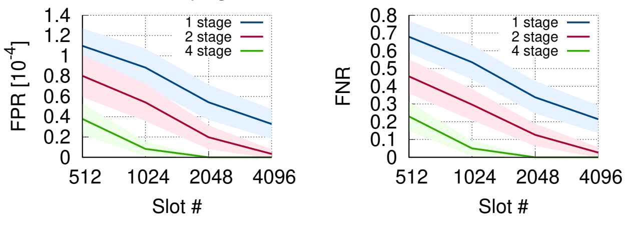
An important attribute of Cebinae is scalability with respect to the number of flows. In contrast to AFQ, PCQ, or ideal FQ, the number of physical queues required remains constant as the number of flows increases. SRAM utilization is the only metric that increases with concurrent flows during each round cycle. The amount of SRAM determines the accuracy of Cebinae's detection data structure of top-1 flows. To evaluate this effect, we replay CAIDA [12] traces collected from a 10 Gbps ISP backbone link to evaluate Cebinae's detection accuracy of $\top$ flows. As shown in Figure 13a, for environments with >400,000 flows/min, the default configuration (2-stage, 2048 slots) provides negligible false positive rates ($< 0.005\%$) of $\top$ flow detection and relatively low false negatives ($< 10\%$) across common round intervals. Using more stages or slots will further reduces the false negative rate (FNR) and false positive rate (FPR) to 0, as shown in Figure 13b. This is on the order of 1000× more flows than AFQ or PCQ can support on equivalent switch hardware.

6 RELATED WORK

Congestion control fairness has a long and rich history in the literature, with many protocols, measurements/metrics, and mechanisms [5, 6, 9, 13, 20–22, 25, 32, 35, 36, 40, 44, 46, 51, 53, 54, 57]. Cebinae focuses on deployments where flows are congestion controlled but with a potentially heterogeneous set of algorithms. Its goal is to prevent persistent unfairness and continuously approximate the eventual max-min fair allocation. Cebinae's taxing and



(a) Varying round interval with 2048 slots



(b) Varying slot number with 100 ms interval

Figure 13: FPR and FNR of $\top$ flow detection for a range of slot numbers and round intervals under CAIDA traces on a 10 Gbps ISP backbone link (100 trials per data point).

redistribution approach is conceptually similar to the bandwidth shuffling mechanism of XCP's fairness controller [29]. However, Cebinae presumes co-existence of legacy end-host algorithms (interoperability) and does not require modifications of existing stacks nor the additional header space to communicate the sender states and rate decisions arbitrated by the switches.

Fundamentally, Cebinae falls into the category of in-network fairness enforcement that do not require end host cooperation. In this category, the most popular style of approach is fair queuing and its many variants. As described in Section 2, existing proposals for fair queuing require specialized hardware or face scalability issues related to their need to track/schedule individual flows in a manner that is fair on packet granularities. Compared to these approaches, Cebinae represents a significant simplification—one that sacrifices packet-granularity fairness at every instance of time but retains unfairness mitigation properties.

Cebinae is also closely related to preferential dropping techniques like RED-PD [38] and AFD [41], which try to achieve similar results to fair queuing, but by dropping flows' excess packets rather than guaranteeing delivery of their fair share. RED-PD, in particular, exploits the skewed flow rate distribution in the Internet and makes a similar observation as Cebinae that a simpler, heavy-hitter-only rate limit can be adapted to provide the same converged properties as fair queuing. Aside from demonstrating an implementation on commodity programmable hardware, Cebinae makes two additional contributions. The first is to propose a complete method of finding the target maximum rate, rather than assuming that it is given. The second is to consider support for non-loss-based protocols.

Finally, we note that Cebinae benefits from and builds directly on top of a slew of recently proposed mechanisms in programmable switches. These include calendar queues [48] (which inspired Cebinae's data-path rate-limiters), heavy hitter detection [49] (which is necessary for bottleneck identification), as well as fast/consistent control- and data-plane interaction [60] (which are used in Cebinae's periodic queue rotations). Cebinae can, therefore, benefit from future advancements in hardware, techniques, or reactive algorithms [33, 34, 50, 55, 61] in any of the above components.

7 FUTURE WORK

We note that Cebinae is amenable to a range of potential extensions and optimizations. We lay out a few such directions.

Providing provable convergence properties. While Cebinae presents a mechanism for scalable in-network fairness enforcement, and it is able to prevent persistent unfairness and enter a steady state in a bounded number of time steps, we leave a more formal description and modelling of Cebinae's convergence properties under heterogeneous rate control and traffic behaviors to future work. To that end, we note that Cebinae will not *guarantee* convergence to max-min fairness, especially in the presence of adversarial congestion control protocols. Instead, Cebinae targets a practical method for improved fairness in typical network configurations.

We postulate that an extension of Cebinae that tracks each bottleneck flow separately would provide the opportunity for much stronger guarantees than what Cebinae currently provides. In particular, we expect it to guarantee equivalent network-level convergence to fair queuing under the assumption of 'eventual stability,' i.e., that bottleneck flows eventually claim their allocated rates, but we leave a proof to future work. The current version of Cebinae does not pursue these guarantees in favor of better statistical multiplexing of bottleneck flows and the higher utilization it provides.

Fine-grained adaptation to current network conditions. Building on the above, another way to improve the robustness and performance of Cebinae is to incorporate heuristics into the reaction strategy, e.g., to limit unnecessary oscillations or to selectively avoid penalties that will cause out-sized short-term fluctuations in bottleneck flow goodput.

Other metrics. Finally, we leave an exploration of practical, in-network mechanisms for other metrics of fairness for future work.

8 CONCLUSION

In today's networks, protocol designers often sacrifice fairness in service of performance and efficiency (either consciously or subconsciously). Especially in public networks where users are free to bring their own congestion control protocols, network operators need practical mechanisms for enforcing conformance beyond static allocation and over-provisioning. Cebinae provides exactly such a mechanism, and materially improves upon the practicality and scalability of prior approaches. Our evaluation results demonstrate that Cebinae can enforce fairness across a wide range of congestion control protocols.

This work does not raise any ethical issues.

ACKNOWLEDGMENTS

We gratefully acknowledge Naveen Kr. Sharma, Jennifer Rexford, Neal Cardwell, Vladimir Gurevich, our shepherd Sergey Gorinsky, and the anonymous SIGCOMM reviewers for all of their help and thoughtful comments. This work was funded in part by Google, Meta, VMWare, and NSF grant CNS-1845749.

REFERENCES

[1] 2022. The Flow Queue CoDel Packet Scheduler and Active Queue Management Algorithm. https://datatracker.ietf.org/doc/html/rfc8290. (January 2022).

[2] 2022. FQ-CoDel. `https://www.nsnam.org/docs/release/3.35/models/html/fq-codel.html`. (January 2022).

[3] 2022. Network Simulator 3. https://www.nsnam.org. (January 2022).

[4] Guido Appenzeller, Isaac Keslassy, and Nick McKeown. 2004. Sizing Router Buffers. *SIGCOMM Comput. Commun. Rev.* 34, 4 (aug 2004), 281–292.

[5] Venkat Arun and Hari Balakrishnan. 2018. Copa: Practical {Delay-Based} Congestion Control for the Internet. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. 329–342.

[6] Andrea Baiocchi, Angelo P Castellani, and Francesco Vacirca. 2007. YeAH-TCP: yet another highspeed TCP. In *Proc. PFLDnet*, Vol. 7. 37–42.

[7] Ran Ben-Basat, Xiaoqi Chen, Gil Einziger, and Ori Rottenstreich. 2018. Efficient measurement on programmable switches using probabilistic recirculation. In *2018 IEEE 26th International Conference on Network Protocols (ICNP)*. IEEE, 313–323.

[8] Jean-Yves Le Boudec. 2021. Rate adaptation, Congestion Control and Fairness: A Tutorial. https://leboudec.github.io/leboudec/resources/tutorial.html. (November 2021).

[9] Lawrence S Brakmo, Sean W O'Malley, and Larry L Peterson. 1994. TCP Vegas: New techniques for congestion detection and avoidance. In *Proceedings of the conference on Communications architectures, protocols and applications*. 24–35.

[10] Bob Briscoe. 2007. Flow rate fairness: Dismantling a religion. *ACM SIGCOMM Computer Communication Review* 37, 2 (2007), 63–74.

[11] Lloyd Brown, Ganesh Ananthanarayanan, Ethan Katz-Bassett, Arvind Krishnamurthy, Sylvia Ratnasamy, Michael Schapira, and Scott Shenker. 2020. On the future of congestion control for the public internet. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*. 30–37.

[12] Caida. 2022. The CAIDA UCSD Statistical information for the CAIDA Anonymized Internet Traces. `https://www.caida.org/data/passive/passive_trace_statistics.xml`. (2022).

[13] Carlo Caini and Rosario Firrincieli. 2004. TCP Hybla: a TCP enhancement for heterogeneous networks. *International journal of satellite communications and networking* 22, 5 (2004), 547–566.

[14] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2016. BBR: Congestion-Based Congestion Control. *ACM Queue* 14, September-October (2016), 20 – 53. `http://queue.acm.org/detail.cfm?id=3022184`

[15] Neal Cardwell, Yuchung Cheng, S Hassas Yeganeh, and Van Jacobson. 2017. BBR congestion control. *Working Draft, IETF Secretariat, Internet-Draft draft-cardwell-iccrg-bbr-congestion-control-00* (2017).

[16] Dah-Ming Chiu and Raj Jain. 1989. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Computer Networks and ISDN systems* 17, 1 (1989), 1–14.

[17] A. Demers, S. Keshav, and S. Shenker. 1989. Analysis and Simulation of a Fair Queueing Algorithm. *SIGCOMM Comput. Commun. Rev.* 19, 4 (aug 1989), 1–12. `https://doi.org/10.1145/75247.75248`

[18] Nandita Dukkipati, Masayoshi Kobayashi, Rui Zhang-Shen, and Nick McKeown. 2005. Processor Sharing Flows in the Internet. In *Proceedings of the 13th International Conference on Quality of Service (IWQoS'05)*. Springer-Verlag, Berlin, Heidelberg, 271–285.

[19] S Floyd. 2008. RFC 5348 TCP-Friendly Rate Control (TFRC) Protocol Specification. *RFC 5348 Proposed Standard* (2008).

[20] Cheng Peng Fu and Soung C Liew. 2003. TCP Veno: TCP enhancement for transmission over wireless access networks. *IEEE Journal on selected areas in communications* 21, 2 (2003), 216–228.

[21] Manfred Georg, Christoph Jechlitschek, and Sergey Gorinsky. 2007. Improving individual flow performance with multiple queue fair queuing. In *2007 Fifteenth IEEE International Workshop on Quality of Service*. IEEE, 141–144.

[22] Sergey Gorinsky and Christoph Jechlitschek. 2007. Fair efficiency, or low average delay without starvation. In *2007 16th International Conference on Computer Communications and Networks*. IEEE, 424–429.

[23] Sergey Gorinsky and Harrick Vin. 2008. Effairness: Dealing with Time in Congestion Control Evaluation. In *Fourth International Conference on Networking and Services (icns 2008)*. IEEE, 40–45.

[24] Sangtae Ha, Injong Rhee, and Lisong Xu. 2008. CUBIC: A New TCP-Friendly High-Speed TCP Variant. *SIGOPS Oper. Syst. Rev.* 42, 5 (jul 2008), 64–74. `https://doi.org/10.1145/1400097.1400105`

[25] Mario Hock, Roland Bless, and Martina Zitterbart. 2017. Experimental evaluation of BBR congestion control. In *2017 IEEE 25th International Conference on Network Protocols (ICNP)*. IEEE, 1–10.

[26] Raj Jain, Arjan Durresi, and Gojko Babic. 1999. Throughput fairness index: An explanation. In *ATM Forum contribution*, Vol. 99.

[27] Rajendra K Jain, Dah-Ming W Chiu, William R Hawe, et al. 1984. A quantitative measure of fairness and discrimination. *Eastern Research Laboratory, Digital Equipment Corporation, Hudson, MA* 21 (1984).

[28] Lavanya Jose, Stephen Ibanez, Mohammad Alizadeh, and Nick McKeown. 2019. A distributed algorithm to calculate max-min fair rates without per-flow state. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 3, 2 (2019), 1–42.

[29] 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)*. Association for Computing Machinery, New York, NY, USA, 89–102.

[30] F. Kaudel. 1998. ATM Forum Performance Testing Specification Draft. https://www.broadband-forum.org/technical/download/af-test-tm-0131.000.pdf. (December 1998).

[31] Frank P Kelly, Aman K Maulloo, and David Kim Hong Tan. 1998. Rate control for communication networks: shadow prices, proportional fairness and stability. *Journal of the Operational Research society* 49, 3 (1998), 237–252.

[32] Tom Kelly. 2003. Scalable TCP: Improving performance in highspeed wide area networks. *ACM SIGCOMM computer communication Review* 33, 2 (2003), 83–91.

[33] Xin Zhe Khooi, Levente Csikor, Jialin Li, Min Suk Kang, and Dinil Mon Divakara. 2021. Revisiting Heavy-Hitter Detection on Commodity Programmable Switches. In *2021 IEEE 7th International Conference on Network Softwarization (NetSoft)*. IEEE, 79–87.

[34] Yiran Lei, Liangcheng Yu, Vincent Liu, and Mingwei Xu. 2022. PrintQueue: Performance Diagnosis via Queue Measurement in the Data Plane. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication (SIGCOMM '22)*. Association for Computing Machinery, Amsterdam, Netherlands. `https://doi.org/10.1145/3544216.3544257`

[35] Douglas Leith and Robert Shorten. 2004. H-TCP: TCP for high-speed and long-distance networks. In *Proceedings of PFLDnet*, Vol. 2004.

[36] Shao Liu, Tamer Başar, and Ravi Srikant. 2008. TCP-Illinois: A loss-and delay-based congestion control algorithm for high-speed networks. *Performance Evaluation* 65, 6-7 (2008), 417–440.

[37] Jonathan Mace, Peter Bodik, Rodrigo Fonseca, and Madanlal Musuvathi. 2015. Retro: Targeted resource management in multi-tenant distributed systems. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. 589–603.

[38] Ratul Mahajan, Sally Floyd, and David Wetherall. 2001. Controlling high-bandwidth flows at the congested router. In *Proceedings Ninth International Conference on Network Protocols. ICNP 2001*. 192–201. `https://doi.org/10.1109/ICNP.2001.992899`

[39] John Nagle. 1987. On Packet Switches with Infinite Storage. *IEEE Transactions on Communications* 35, 4 (1987), 435–438. `https://doi.org/10.1109/TCOM.1987.1096782`

[40] Jitendra Padhye, Victor Firoiu, Don Towsley, and Jim Kurose. 1998. Modeling TCP throughput: A simple model and its empirical validation. In *Proceedings of the ACM SIGCOMM'98 conference on Applications, technologies, architectures, and protocols for computer communication*. 303–314.

[41] Rong Pan, Lee Breslau, Balaji Prabhakar, and Scott Shenker. 2003. Approximate Fairness through Differential Dropping. *SIGCOMM Comput. Commun. Rev.* 33, 2 (apr 2003), 23–39.

[42] 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.

[43] Larry L Peterson and Bruce S Davie. 2007. *Computer networks: a systems approach*. Elsevier.

[44] Adithya Abraham Philip, Ranysha Ware, Rukshani Athapathu, Justine Sherry, and Vyas Sekar. 2021. *Revisiting TCP Congestion Control Throughput Models & Fairness Properties at Scale*. Association for Computing Machinery, New York, NY, USA, 96–103.

[45] Lucian Popa, Gautam Kumar, Mosharaf Chowdhury, Arvind Krishnamurthy, Sylvia Ratnasamy, and Ion Stoica. 2012. FairCloud: Sharing the network in cloud computing. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*. 187–198.

[46] Dominik Scholz, Benedikt Jaeger, Lukas Schwaighofer, Daniel Raumer, Fabien Geyer, and Georg Carle. 2018. Towards a Deeper Understanding of TCP BBR Congestion Control. In *2018 IFIP Networking Conference (IFIP Networking) and Workshops*. 1–9. `https://doi.org/10.23919/IFIPNetworking.2018.8696830`

[47] Naveen Kr. Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. 2018. Approximating Fair Queueing on Reconfigurable Switches. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 1–16. `https://www.usenix.org/conference/nsdi18/presentation/sharma`

[48] Naveen Kr. Sharma, Chenxingyu Zhao, Ming Liu, Pravein G Kannan, Changhoon Kim, Arvind Krishnamurthy, and Anirudh Sivaraman. 2020. Programmable Calendar Queues for High-speed Packet Scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 685–699. `https://www.usenix.org/conference/nsdi20/presentation/sharma`

[49] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S. Muthukrishnan, and Jennifer Rexford. 2017. Heavy-Hitter Detection Entirely in the Data Plane. In *Proceedings of the Symposium on SDN Research (SOSR '17)*. Association for Computing Machinery, New York, NY, USA, 164–176.

[50] John Sonchack, Devon Loehr, Jennifer Rexford, and David Walker. 2021. Lucid: A language for control in the data plane. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 731–747.

[51] Ion Stoica, Scott Shenker, and Hui Zhang. 1998. Core-Stateless Fair Queueing: Achieving Approximately Fair Bandwidth Allocations in High Speed Networks. *SIGCOMM Comput. Commun. Rev.* 28, 4 (oct 1998), 118–130.

[52] Ao Tang, Jiantao Wang, Sanjay Hegde, and Steven H Low. 2005. Equilibrium and fairness of networks shared by TCP Reno and Vegas/FAST. *Telecommunication Systems* 30, 4 (2005), 417–439.

[53] Ao Tang, Jiantao Wang, Steven H Low, and Mung Chiang. 2007. Equilibrium of heterogeneous congestion control: Existence and uniqueness. *IEEE/ACM Transactions on Networking* 15, 4 (2007), 824–837.

[54] Belma Turkovic, Fernando A Kuipers, and Steve Uhlig. 2019. Fifty shades of congestion control: A performance and interactions evaluation. *arXiv preprint arXiv:1903.03852* (2019).

[55] Shie-Yuan Wang, Hsien-Wen Hu, and Yi-Bing Lin. 2020. Design and implementation of tcp-friendly meters in p4 switches. *IEEE/ACM Transactions on Networking* 28, 4 (2020), 1885–1898.

[56] Ranysha Ware, Matthew K. Mukerjee, Srinivasan Seshan, and Justine Sherry. 2019. Beyond Jain's Fairness Index: Setting the Bar For The Deployment of Congestion Control Algorithms. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks (HotNets '19)*. Association for Computing Machinery, New York, NY, USA, 17–24.

[57] Ranysha Ware, Matthew K Mukerjee, Srinivasan Seshan, and Justine Sherry. 2019. Modeling BBR's interactions with loss-based congestion control. In *Proceedings of the internet measurement conference*. 137–143.

[58] Jörg Widmer, Robert Denda, and Martin Mauve. 2001. A survey on TCP-friendly congestion control. *IEEE network* 15, 3 (2001), 28–37.

[59] Lisong Xu, Khaled Harfoush, and Injong Rhee. 2004. Binary increase congestion control (BIC) for fast long-distance networks. In *IEEE INFOCOM 2004*, Vol. 4. IEEE, 2514–2524.

[60] Liangcheng Yu, John Sonchack, and Vincent Liu. 2020. Mantis: Reactive programmable switches. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 296–309.

[61] Liangcheng Yu, John Sonchack, and Vincent Liu. 2022. OrbWeaver: Using IDLE Cycles in Programmable Networks for Opportunistic Coordination. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 1195–1212. `https://www.usenix.org/conference/nsdi22/presentation/yu`