USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Managing Congestion Control Heterogeneity on the Internet with *Approximate Performance Isolation*

Ayush Mishra, *ETH Zurich;* Archit Bhatnagar, *University of Michigan;*
Yixuan Zhang, *Tsinghua University;* Ben Leong, *National University of Singapore;*
Ya Gao, *Wuxi Institute of Technology;* Raj Joshi, *Harvard University*

https://www.usenix.org/conference/nsdi26/presentation/mishra

This paper is included in the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation.

May 4–6, 2026 • Renton, WA, USA

ISBN 978-1-939133-54-0

Open access to the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation is sponsored by

جامعة الملك عبدالله للعلوم والتقنية
King Abdullah University of Science and Technology

ARTIFACT EVALUATED
USENIX
AVAILABLE

Managing Congestion Control Heterogeneity on the Internet with *Approximate Performance Isolation*

Ayush Mishra[1] *Archit Bhatnagar*[2] *Yixuan Zhang*[3] *Ben Leong*[4] *Ya Gao*[5] *Raj Joshi*[6]
[1]*ETH Zurich* [2]*University of Michigan* [3]*Tsinghua University*
[4]*National University of Singapore* [5]*Wuxi Institute of Technology* [6]*Harvard University*

Abstract

The Internet hosts a diverse mix of congestion control algorithms (CCAs) optimized for specific throughput-delay trade-offs. However, traditional queuing disciplines and AQMs struggle to manage this heterogeneity and often lead to unfairness and suboptimal performance. In this paper, we explore isolation techniques that can allow competing CCAs to make their desired throughput-delay trade-offs independent of who they compete with. More specifically, we motivate *Approximate Performance Isolation* between competing flows by grouping flows with similar desired throughput-delay trade-offs in the same queue. We also present *Santa*, a new practical and scalable multi-queue AQM built on the principles of approximate performance isolation. *Santa* infers each flow's throughput-delay preferences by comparing their buffer occupancy, and shuffles aggressive ("naughty") and passive ("nice") flows into appropriate queues over time. We prototype *Santa* on a programmable switch to demonstrate that it is practical, scalable, and can approximate the isolation benefits of Fair Queuing (FQ) with a handful of of queues.

1 Introduction

Today's Internet needs to support a diverse range of applications with different performance requirements. For example, an online game is typically much more delay-sensitive than a file transfer, which can tolerate higher delays in exchange for greater bandwidth. We expect application developers to use congestion control algorithms (CCAs) that optimize for their desired throughput-delay trade-offs. Thus, the CCA deployed on a website is often highly correlated with the content it serves. BBR is currently the most popular CCA for websites serving video traffic [40]. More recent studies have even found that websites can run different CCAs for serving different kinds of assets on the same webpage [39].

However, since different CCAs are optimized to make different throughput-delay trade-offs, they often interact poorly when competing with each other under certain conditions. This issue can be particularly pronounced in the interactions between CUBIC and BBR, which happen to be the two most popular CCAs on the Internet [39]. BBR is designed to operate near the *Kleinrock point* to minimize queuing [13], while CUBIC is a buffer-filler by design to maximize throughput. However, when BBR and CUBIC flows compete in a deep buffer, not only are they unfair, but they also inflicts high delays on each other [54].

A natural question arises: *how do we allow CCAs with contrasting operating points, like BBR and CUBIC, to play well together*? In other words, how do we allow BBR to achieve low latency and CUBIC to achieve high throughput when sharing the same bottleneck link?

Existing queuing disciplines and AQMs are not designed to support a heterogeneous mix of CCAs and provide isolation between different flows with different desired throughput-delay trade-offs. While flow-level isolation can be achieved with Fair Queuing (FQ) [18] by providing each flow with its own queue, this is impractical at Internet-scale [33]. There have been proposals for AQMs like L4S [26] that attempt to incorporate a flow's preferences, but these often require explicit notification mechanisms and expect the end hosts to be honest about their preferences.

We show that CCAs with different desired throughput-delay trade-offs competing with each other can be a source of inefficiency in a network (§2.1), but these inefficiencies can be avoided if flows with different operating points are *isolated* from each other (§2.2). We hence explore practical and scalable ways to achieve this *performance isolation* between flows without relying on fair queuing. More specifically, we present a way to achieve *Approximate Performance Isolation* by placing flows with similar desired throughput-delay trade-offs in the same queue(§2.3)

As a proof of concept, we present *Santa*, a novel, practical, and scalable multi-queue AQM that is built on the principles of approximate performance isolation (§4). *Santa* assigns flows to different queues based on their desired throughput-delay trade-offs. It converges to an appropriate queue assignment by comparing the bandwidth share of a flow compared to the other flows in its current queue over the duration of a round. Flows that receive significantly higher or lower bandwidth than the fair share are called the *naughty* and *nice* flows respectively, and are shuffled between queues between rounds. By using the relative performance of flows, *Santa* is able to distill them across queues according to their desired throughput-delay trade-offs – thereby achieving approximate performance isolation.

We implement *Santa* on a programmable Intel Tofino switch to demonstrate that it is practical (§5) and show that we can achieve *approximate performance isolation* scalably with a small number of queues (§6). Despite its limitations, we use *Santa* to motivate *approximate performance isolation* as a practical design goal for Internet AQMs.

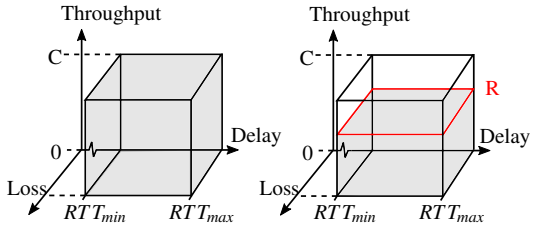


Figure 1: All networks can be thought of as a trade-off space. This space can be constrained by both natural (left) and arbitrary (right) constraints - such a rate-limit (R) applied by the network operator.

In summary, *Santa* allows different CCAs to make their respective trade-offs independently, without being unfairly influenced by competing flows. By providing each CCA with the flexibility to maintain its desired operating point—whether it be maximizing throughput, minimizing latency, or avoiding buffer overflows—*Santa* aims to enable CCAs to coexist in a more harmonious and efficient manner. In particular, we believe that by decoupling the performance of different CCAs from their inter-dependencies, *Santa* could mitigate issues of unfairness, promote stability, and better accommodate the growing diversity of CCAs deployed on the Internet.

2 Background and Motivation

While traditional congestion control algorithms (CCAs) were designed with the simple goal of utilizing the bottleneck bandwidth and preventing a congestion collapse [16, 28], most CCAs that run on the Internet today are more nuanced. The congestion control space is populated with numerous variants designed to achieve different trade-offs in the network [7, 9, 13, 22, 24, 45]. In this section, we will present an abstract view of the network as a trade-off space and illustrate how different CCAs explore this trade-off space differently.

If viewed as a black box, a bottleneck link in a network can impact the packets sent on it under the following constraints:

1. **Bandwidth** $[0, C]$**:** It can control how quickly packets are forwarded, bounded by the bottleneck link capacity C.
2. **Delay** $[RTT_{min}, RTT_{max}]$**:** On top of the per-packet service time($\frac{1}{bw}$), it can also impose additional delay on these packets before forwarding them. Typically, this delay cannot be less than the propagation delay RTT_{min} of the network and larger than the maximum queuing delay determined by the size of the buffers on the path ($RTT_{max} = RTT_{min} + \frac{B}{C}$).
3. **Drops** $[0, 1]$**:** Finally, the network can also decide not to forward a packet and drop it. We can model drops with a probability between 0 and 1, because the queuing discipline could be non-deterministic like RED [20].

These 3 constraints form a three-dimensional trade-off space, as illustrated in Figure 1. Each of these constraints can emerge naturally, or be applied explicitly by the network. For example, packet drops can happen both as a result of natural buffer overflows or explicit drops by the AQM [21, 43]. A network operator might also choose to limit flows to a rate R that is lower than the link capacity C (see Figure 1).

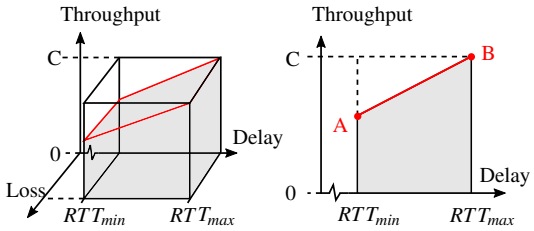


Figure 2: The difference between the operating points of a delay-sensitive (A) and throughput-hungry (B) flow for a hypothetical network.

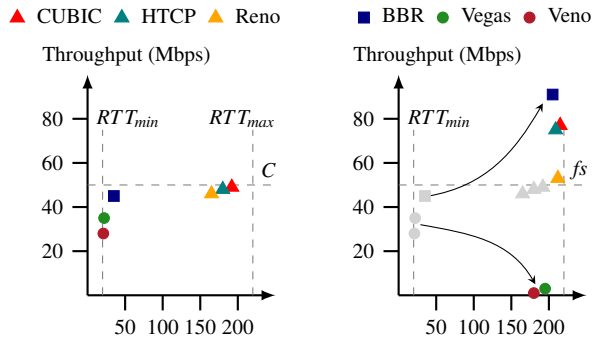
2.1 What does a CCA do?

If we take this abstract view of the bottleneck link, the role of a CCA is to operate at a point in this constrained space closest to its *desired ("natural") operating point*. We can imagine there being a difference between how a throughput-hungry and a delay-sensitive flow operates in the same network.

Consider a hypothetical network that constrains the throughput and delay as illustrated in Figure 2. For the sake of simplicity, we will focus on the throughput vs delay plane, where the network allows a flow to have high throughput, but not without a delay cost. In such a bottleneck link (without other competing flows), a delay-sensitive flow might be willing to give up some bandwidth and operate at point A to reduce the end-to-end delay. On the other hand, a throughput-hungry flow that can tolerate high delays would prefer to operate at point B.

To illustrate how this works, we took 6 different CCAs available in the Linux kernel and ran them *individually* through a 50 Mbps bottleneck link with a 10 BDP buffer and plotted their average throughput and delay to understand their desired ("natural") operating points (in the absence of competing flows). As we can see from Figure 3a, these operating points will vary depending on the CCA. CCAs like CUBIC [24] and Reno [27] will attempt to fill the bottleneck buffer to maximize throughput. Delay-sensitive CCAs like Vegas [9] typically like to maintain low delay, even at the cost of under-utilization. BBR [13] is somewhere in between, and aims to operate at the Kleinrock point [31].

However, these varying operating points can often make these CCAs incompatible with each other when they share a bottleneck link. To illustrate this, we ran the same 6 CCAs, but this time we allowed them to compete in a 300 Mbps link (fairshare 50 Mbps) with a 10-BDP bottleneck buffer (see Figure 3b). Under this new setting, the delay-sensitive flows are starved and experience the largest displacement from their desired operating point. The buffer-fillers fill the bottleneck buffer and seize a larger than fair share of the bottleneck bandwidth. None of this is surprising. Because of how BBR becomes aggressive in the presence of buffer-fillers [41, 55], it obtains the largest share of the bottleneck bandwidth-but not without suffering from higher delays itself.



(a) In isolation. **(b)** Under competition.

Figure 3: The throughput-delay trade-off space explored by different CCAs in the Linux kernel.

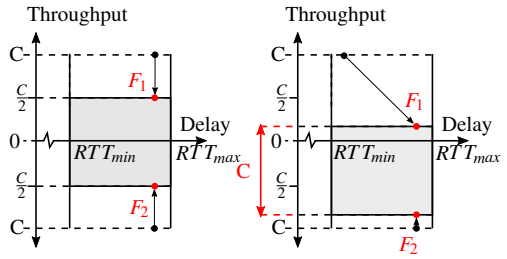


Figure 4: Since the bottleneck link capacity is limited, competing flows can add a moving constraint to each other's performance.

2.2 How different CCAs compete

A flow's desired operating point in the throughput-delay plane can directly impact how it competes with another flow. We illustrate this relation in Figure 4, where we consider two flows competing in a bottleneck link with bandwidth C. We mirror the throughput vs delay plane of the second flow and place it below the first flow. If both these flows desire the same throughput-delay tradeoffs, we can expect them to share the link equally (by symmetry) and collaboratively achieve their desired throughput-delay trade-offs (Figure 4, left). However, if the desired operating points of the two competing flows are dissimilar, we can expect the more throughput-hungry flow to *pull* the equilibrium away from the flows' natural operating points and create unfairness (Figure 4, right).

This unfairness between real-world CCAs has been documented by numerous measurement studies [41, 55, 58]. It is well known that while most CCAs tend to be fair to other flows that are also running the same CCA, they tend to not play very well with other CCAs. We replicate some of these well-known trends in the throughput-delay space in Figure 5.

Two flows of the same CCAs are able to compete fairly between themselves and collaboratively achieve their desired operating point, as shown in Figures 5a and 5b. However, when CUBIC competes with Vegas, CUBIC, being the more throughput-hungry flow, fills the buffer and starves the competing Vegas flow as shown in Figure 5c.

In some instances, when flows with different desired operating points compete, they can even mutually harm each other. For example, as shown in Figure 5d, when CUBIC and BBR share a bottleneck link, the throughput-hungry CUBIC flow has its throughput reduced while the delay-sensitive BBR flow suffers higher delay. Overall, we can make two key observations from observing how CCAs compete:

1. CCAs with similar desired operating points can collaboratively achieve their desired throughput-delay trade-offs and coexist amicably (Figures 5a and 5b).
2. CCAs with different operating points can often be incompatible, and even mutually harmful (Figures 5c and 5d).

2.3 Case for *Performance Isolation*

Given our observations in §2.2, we need a way for CCAs with different desired throughput-delay trade-offs to be treated differently and independently of the other competing CCAs. Unfortunately, classical AQMs are unable to do so, even if they can sometimes help mitigate the effects of a heterogeneous mix of CCAs competing. For example, Codel [43] can reduce the queuing delay a competing CUBIC flow inflicts on delay-sensitive Vegas flows. However, it still does not prevent Vegas from being starved. Also, since Codel behaves like a shallow buffer, it can result in under-utilization of the bottleneck bandwidth. In other settings, AQMs can even exacerbate existing performance issues. For example, if we apply RED [21] to the setting described in Figure 5d, BBR gains an even higher share of the bottleneck bandwidth, while still inflicting high queuing delays on itself because unlike CUBIC, it does not treat packet drops as a congestion signal.

What we desire is that each flow is able to achieve its desired throughput-delay tradeoffs regardless of who it is competing with. We call this *performance isolation*. A naïve way to achieve performance isolation could be to deploy Fair Queuing (FQ), where each flow is isolated in its own queue. Unfortunately, this is not practical given the large number of flows in real-world networks [33].

While there have been proposals for several approximate fair queuing solutions [14, 33, 46, 62], they fall short of the goals of performance isolation because they approximate the wrong feature of FQ. State-of-the-art approximate fair queuing solutions like AFQ [46], AHAB [33], SFQ [14], and HCSFQ [62] are designed to ensure each flow receives its fair share of the bottleneck bandwidth, with AHAB and HCFQ also being capable of providing predetermined flow *groups* a fixed bandwidth share. However, this is not the same as *performance isolation*, which is what we need: we want flows to achieve their desired throughput-delay trade-offs regardless of which CCAs they are competing with at the bottleneck. Under approximate FQ, it is still possible for flows to inflict delays, suffer from excessive packet loss, and therefore impact each other despite receiving their fairshare bandwidth.

Fortunately, we show that we do not need FQ or perfect performance isolation to allow different CCAs to coexist. Instead, we argue that all we need is a scalable way to achieve *approximate performance isolation* between flows.

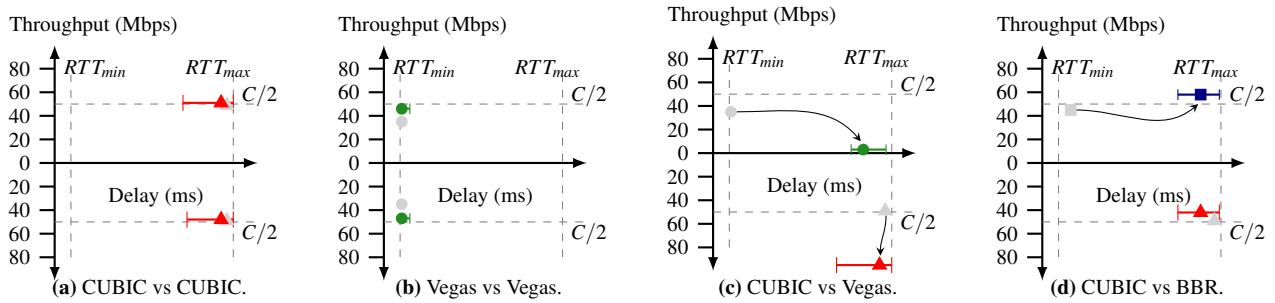


Figure 5: Real world example of unfairness arising when ▲CUBIC, ■BBR, and ●Vegas compete in a FIFO queue.

3 Approximate *Performance Isolation*

Fair Queuing (FQ) achieves performance isolation by assigning each flow its own queue, which allows each flow to maintain whatever buffer-occupancy it desires. *Approximate performance isolation* attempts to achieve performance similar to FQ with a much smaller number of queues than the number of flows. We note that *performance isolation* goes beyond the idea of Congestion Control Algorithm Independence (CCAI) [10]. While CCAI aims to guarantee that flows get a fixed throughput share regardless of who they compete with, the goals of *performance isolation* goes beyond just throughput guarantees. Under perfect performance isolation, not only would a flow get a fixed share of the bandwidth, it would also be able to do so while operating at the same point on the delay-throughput frontier regardless of who it competes with. While in theory performance isolation requires each flow to have its own queue, *approximate performance isolation* aims to meet these guarantees with a handful of queues.

To better understand approximate performance isolation, consider the example illustrated in Figure 6, where there are 5 flows, F_{1-5}, passing through a bottleneck, each with a different desired operating point in the throughput-delay trade-off space. For most efficient CCAs, we can expect these operating points to lie on a Pareto throughput-delay *frontier* [56].

In other words, flows with different preferences can naturally be ordered according to their preference for throughput or delay. A FQ scheme would provide each flow with a different queue, thereby ensuring performance isolation. However, we can reduce the number of queues by placing flows that want similar throughput-delay trade-offs together, as shown in Figure 6. Here, flows F_1 and F_2, and flows F_4 and F_5 are close to each other on the throughput-delay plane. Following our observations from §2.2, we can expect them to collaboratively operate at operating points close to their desired operating point. Since the displacement from the natural operating point is small, we achieve *approximate* performance isolation.

To see how well this works with real CCAs, we re-ran the experiment in Figure 3b with 6 flows with FQ (6 queues) comprising the following 6 CCAs: CUBIC, HTCP, Reno, BBR, Vegas, and Veno. From Figure 7a, we note that CUBIC, HTCP, Reno, BBR, Vegas, and Veno (in that order) provide a good spread across the Pareto frontier.

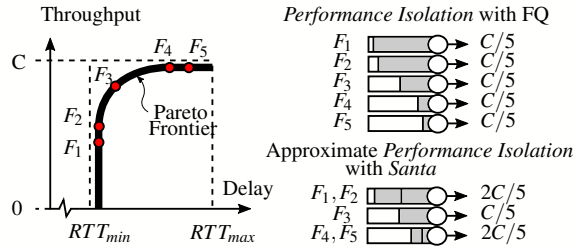


Figure 6: Different CCAs occupy different points on the throughput-delay frontier. We can use their preferences to group them in individual queues to approximate performance isolation.

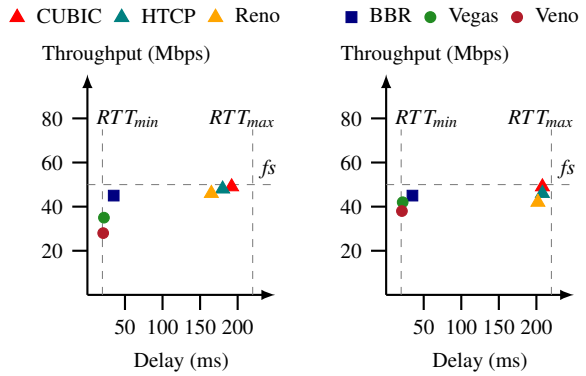


Figure 7: Comparison between perfect and approximate Performance Isolation.

Next, we repeated this experiment, but with only 3 FIFO queues. The three buffer fillers (CUBIC, HTCP, and Reno) shared one queue while the two delay-sensitive flows (Vegas and Veno) shared a separate queue. BBR was placed in the third queue on its own. We can see from Figure 7b that we can achieve approximate performance isolation with just three queues. In other words, the key insight is that we can achieve *approximate performance isolation* by grouping flows that are close together on the Pareto frontier into the same queue.

3.1 Inferring the desired operating point

To group flows based on their desired throughput-delay trade-offs, we need a way to determine where they lie on the throughput-delay frontier. While it is not possible to directly determine the desired throughput-delay trade-offs for a flow, we can infer the preference of a flow relative to other flows

sharing the same queue. This insight follows from the simple observation that when multiple flows compete, the flow that is the highest on the throughput-delay frontier is likely to also be the most *aggressive*, and therefore will naturally obtain the highest throughput share.

Defining Aggression. In order to sort flows by their operating points, we need to be able to measure *aggression*. A principled definition of aggression should capture the steady-state (packets in flight in relation to the BDP, sending rate compared to the fairshare) and transient (how readily a flow utilizes spare link capacity) behavior of a flow. *Santa* adopts a simpler, narrower definition. We measure the number of packets competing flows push into the buffer as a measure of their *relative* aggression. When two flows compete in the same queue, the flow that maintains a higher buffer occupancy is considered more aggressive than the other. We use this definition since it captures the flow's delay-tolerence in favor of higher throughput. We note that this definition of aggression is *relative* (depends on which other CCAs a flow competes with) and not absolute. *Santa* assumes that this relative aggression is transitive between flows. We discuss the consequences of making this assumption in §3.2.

Ignoring Mice Flows. Our analysis of CAIDA traces [12] (§6.4) revealed that 90% of the flows on the Internet are short-lived ("mice") flows. These flows will end before we can take any action on them. Hence, we will only provide performance isolation for flows that are not "mice" flows (or flows that last more than a single flight of packets). In any case, "mice" flows will likely care more about flow completion times (FCT), instead of the throughput-delay trade-offs.

To filter out the "mice" flows, when a new flow is observed, we route the first 10 packets to a high-priority mice queue. We set the threshold to 10 packets because it is the default TCP starting window size. After that, starting from a flow's second flight of packets, it gets assigned a queue.

Shuffling Flows Between Queues. In the same way that the flows can be ordered along the throughput-delay frontier, the set of queues that we use to group the flows is ordered. When a new non-mice flow is added, we assign it to one of these queues. Our key insight is a simple shuffling algorithm that considers the bandwidth share of a flow in its assigned queue, which we describe in Figure 8, is sufficient to group similar flows together. In particular, we can track the bytes transferred by each flow in each queue at the end of fixed and regular intervals, which we call a round. If we find that a flow in Q_i has a significantly larger bandwidth share than the other flows in the same queue, it will get "promoted" to a higher queue Q_{i+1}. On the contrary, if we notices that a flow is being starved for bandwidth in its current queue, it will be moved to a lower queue number Q_{i-1}. After a number of rounds, the flows will be naturally be grouped into different queues according to their level of aggression.

3.2 Implicit assumptions

While the shuffling mechanism in §3.1 seems straightforward, it does make some implicit assumptions about how we expect different flows and CCAs to interact as a function of their desired throughput-delay trade-offs. We discuss some of these assumptions in this section.

Conflating fairness with desired operating points. By attempting to infer a flow's desired operating point from its buffer occupancy relative to the other flows in the same queue, we risk conflating fairness with similarity in desired throughput-delay trade-offs. For most CCAs, this is not a huge issue because flows that attempt to operate at similar operating points will tend to be fair to each other.

However, this assumption may break when CUBIC and BBR flows compete in shallow buffers. Previous work has shown that in networks with BDP-sized buffers, CUBIC and BBR flows can be fair to each other [41]. In such a setting, our shuffling algorithm would place CUBIC and BBR in the same queue, because they will compete fairly with each other. However, this would be sub-optimal, since CUBIC and BBR have different desired operating points – and BBR flows would do strictly better if they were isolated in their own queues. This is a risk that cannot be fully avoided by our current shuffling algorithm. However, we can mitigate its impact by sizing the buffer either smaller or larger than the BDP. It is also possible for flows that want the exact same throughput delay trade-offs to be unfair to each other because they do not interact well. CCAs that have RTT unfairness can suffer from this. Our shuffling algorithm can however address this scenario by placing these flows in different queues.

Implied Transitivity of Aggression . Our shuffling strategy also has the implied assumption that the aggressiveness of a flow relative to other flows is transitive. That is, if flow A is more aggressive than flow B when they compete, and flow B is more aggressive than flow C when they compete, then flow A must be more aggressive than flow C when they compete. This would be true if a group of flows' ordering on the throughput-delay frontier (Figure 6) is the same if those flows were ordered based on how aggressive they were when they competed with each other.

However, for real-world CCAs, this is not always true. One example is when CUBIC and BBR compete. Even though BBR lies to the left of CUBIC on the throughput-delay frontier, it can still be more aggressive than CUBIC when the buffer is shallow. However, we argue that this does not matter, because as long as flows get shuffled based on their relative aggression to each other, flows with similar aggression will still be eventually grouped into the same queue, and we will achieve approximate performance isolation. In other words, while it is possible for flows not to be sorted by their relative order on the throughput-delay frontier across queues, they will still be grouped with other flows with similar desired operating points.

4 *Santa*'s Design

In this section, we describe *Santa*, our new AQM that achieves *approximate performance isolation*. With approximate performance isolation, we expect all the flows to operate at operating points that are close to their desired operating point.

In *Santa*, we maintain one high-priority mice queue and K *Santa* queues. Non-mice flows are randomly assigned to one of K *Santa* queues of equal priority, that are ordered from Q_1 to Q_k. The most aggressive flows are grouped into the highest queue (Q_k) and the least aggressive and most delay-sensitive flows in the lowest queue (Q_1).

At regular intervals, the AQM reviews its assignment policy by assessing each flow's average buffer occupancy compared to the other flows in the same queue. If a flow's average buffer occupancy exceeds the average per-flow buffer occupancy in that queue by some threshold, we take this as a hint that that flow belongs in a higher queue with other more throughput-hungry flows. Such flows are moved from Q_i to Q_{i+1}.

A similar rule applies for flows with buffer occupancy less than some threshold of the average per-flow buffer occupancy in the queue. In such instances, we infer that the flow is less aggressive and move it to a lower queue (Q_{i-1}) that would have a lower queuing delay by virtue of containing the less aggressive flows.

The implementation *Santa* involves 3 key design choices: (i) to which queue do we assign a new flow; (ii) how do we decide which flows should be shuffled; and (iii) how do we determine the bandwidth share to be allocated to each of the K queues. An overview of *Santa* is shown in Figure 8. We shall discuss these design choices in the following subsections.

4.1 Initial Queue Assignment

When we get a new flow, we must determine its initial queue assignment. We perform this assignment in two stages.

Our analysis of CAIDA traces [12] (see Figure 9) reveals that a surprisingly large proportion ($\approx$90%) of flows on the Internet are mice flows. Since it is not possible to know if a new flow is a mice flow or an elephant flow from the onset, *Santa* treats the first 10 packets of each flow as a mice flow and routes them through a special mice flow queue.

The mice flow queue has strict priority over all the other *Santa* queues to minimize the FCT of the mice flows. Based on our analysis of publicly available CAIDA traces [12], the first 10 packets of all flows make up less than 10% of the traffic volume. Therefore, we do not expect the strict priority mice flow queue to cause stalls under realistic scenarios.

Flows longer than 10 packets will be assigned to a *Santa* queue. If an empty queue is available, a new flow will be assigned to one; if all K queues contain live flows, a new non-mice flow will be assigned to one of the K *Santa* queues at random based on a weighted probability. In particular, likelihood of a flow being assigned to a *Santa* queue is proportional to the number of flows already assigned to that queue. We argue that this assignment strategy has two main benefits.

Maximizing the likelihood of the correct initial assignment. Ideally we want to assign a new flow to a queue that corresponds to its level of aggression. Unfortunately, we cannot determine the level of aggression until we assign a flow into a queue with other flows. However, if we assume that the distribution of CCAs is generally stable, then assigning a new flow to a queue (group of flows) with a probability proportional to the size of the group naturally maximizes the likelihood of assigning a new flow to the right group. For example, if 90% of flows are in Queue 1 and 10% of flows are in Queue 2, with no additional information our best guess is mapping a new flow to Queue 1 with probability 0.9 and to Queue 2 with probability 0.1.

Improving Stability. When a new flow is assigned to a queue with pre-existing flows, the new flow can disrupt the pre-existing flows, especially if the new flow is very aggressive. Hence, assigning a new flow to a queue with a large number of pre-existing flow has a beneficial side effect that it will likely have less impact that having it be assigned to another queue with fewer flow. In *Santa*, we also allocate more bandwidth to queues with more flows (§4.3). In other words, doing so will also mitigate the risk of overburdening a queue with limited bandwidth assigned to it.

4.2 Flow Shuffling

Recall that our goal is to group different flows with other flows that are nearby on the throughput-delay pareto frontier, by observing how aggressively a flow behaves compared to other flows in the same queue. Hence, once a flow is assigned to a *Santa* queue, we monitor its average queue occupancy in comparison to the other flows in the same queue. In our prototype, we determine average queue occupancy every 10 seconds, but the duration is a configurable parameter.

After we determine the average buffer occupancy of a flow over the last round, B_i, we compare this value to the average per-flow buffer occupancy of all the flows in that queue $\bar{B}$. If a flow is too aggressive ($B_i > r\bar{B}$), we move it to a higher queue (from Q_i to Q_{i+1}). On ther other hand if it is not able to compete with the other flows in its queue ($B_i < \frac{\bar{B}}{r}$), we move to a lower queue with less aggressive flows (Q_i to Q_{i-1}).

Impact of shuffling thresholds. Effectively, *Santa* will tolerate throughput unfairness in a queue by up to a factor of r^2. If we allocate the bandwidth proportional to the number of flows in each queue, each queue will maintain these bounds relative to the fairshare bandwidth. Therefore, even across queues, the worst case unfairness would be no larger than a factor of r^2. If we set r to something very small, it would cause frequent and unstable shuffles between queues; if we set r to something larger, we would have more infrequent shuffles, but we would need to accept more unfairness among the flows. *Santa* sets $r = 2$, but r is clearly a tuneable parameter depending how much unfairness we are willing to accept.

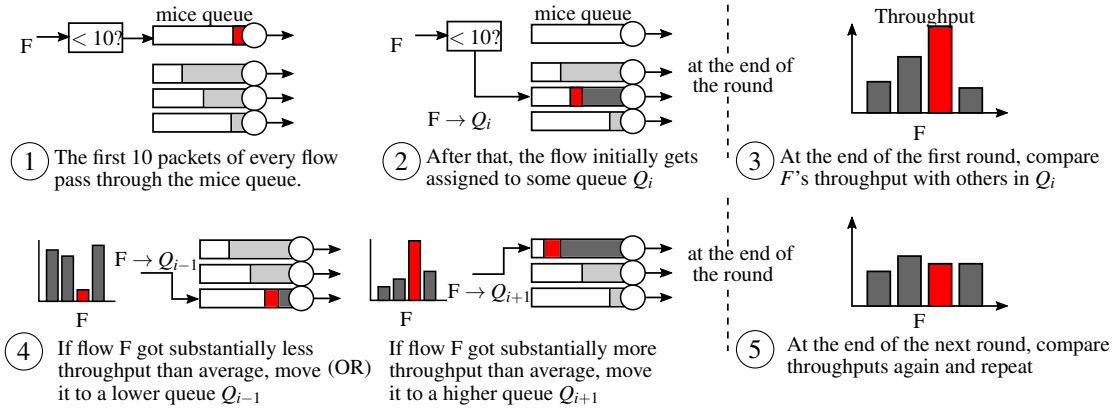


Figure 8: *Santa* compares a flow's throughput with the other flows in its queue and shuffles them if they are either too aggressive or too *nice*.

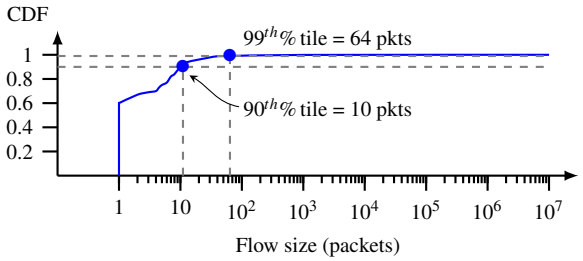


Figure 9: CDF of flow lengths for 60mins of CAIDA traces [12]

4.3 Bandwidth Allocation

In our prototype, each *Santa* queue is allocated bandwidth proportional to the number of flows assigned to it. This bandwidth assignment happens at the end of each round and remains fixed for the duration of the round.

However, this bandwidth allocation policy be easily be tweaked to provide the throughput-hungry flows with a larger share of the throughput than delay-sensitive flows, if so desired. In our implementation, our goal is to distribute bandwidth relatively fairly (within the bounds stated in §4.2) between the competing flows.

5 Prototype Implementation

We envision deploying *Santa* at any point on the Internet where the number of flows make it impractical to achieve performance isolation via Fair Queuing. To this end, and to investigate the practical constraints on implementing *Santa* on hardware switches, we prototype it on an Intel Tofino switch (bf-sde 9.11.2) with 950 lines of P4 code for the data plane (DP) and 1.2k lines of C++ code for the control plane (CP).

Figure 10 summarizes *Santa*'s prototype, which operates in three stages: (i) during each round, the DP uses the Queue Delay structure to record per-queue flow behavior; (ii) at the end of each round, the CP makes shuffling decisions (§4.2) and computes new queue bandwidth allocations (§4.3); (iii) the CP then updates the Q-Assign Table with new queue assignments and configures the switch's scheduler with the new bandwidth allocations. Additionally, the DP filters mice flows using a Count-Min sketch and assigns new, non-mice flows to queues via initial weighted random assignment (§4.1).

5.1 Queue Assignment at the Ingress

During each round, the DP processes packets from both new flows and those previously observed. The Q-Assign Table manages the latter with exact-match entries that map the 32-bit hash of each flow's 5-tuple to a specific queue. After every round, the control plane updates these entries according to the latest shuffling decisions.

New flows that incur a miss in the Q-Assign Table are processed by a Count-Min sketch (details in §6.4) to track their per-round packet count. All packets from a flow are directed to the mice queue until the flow exceeds 10 packets. Once this threshold is crossed, the flow is assigned to a *Santa* queue via weighted-random selection, with weights proportional to each queue's occupancy in the previous round (§5.3). We implement this selection using a range-match table, with ranges set according to the weights. The assignment for each new large flow is stored in a register (hashed by the five-tuple) to keep it consistent throughout the round, and at the end of the round, it is added to the Q-Assign Table.

5.2 Recording queue behavior at the Egress

We quantify each flow's queue behavior per round by its cumulative queue delay—the total sojourn time of all packets during that round—as a proxy for average queue occupancy. The Queue Delay data structure at the egress maintains this information using two levels of 64k-entry, 64-bit registers. Each entry stores a 32-bit flow fingerprint (for collision detection) and a 32-bit cumulative queue delay. With 256 ns granularity, the delay field can represent over 1k seconds per flow per round. In case of collisions, the corresponding entry in the second-level register is used instead. Both registers are partitioned into segments according to the number of configured queues (e.g., four 16k-entry segments for four queues), with each segment dedicated to flows in its respective queue. To allow the CP to read the Queue Delay data structure while the DP continues processing traffic, two copies of the structure are maintained, one for reading and one for writing, which are swapped atomically at every round boundary.

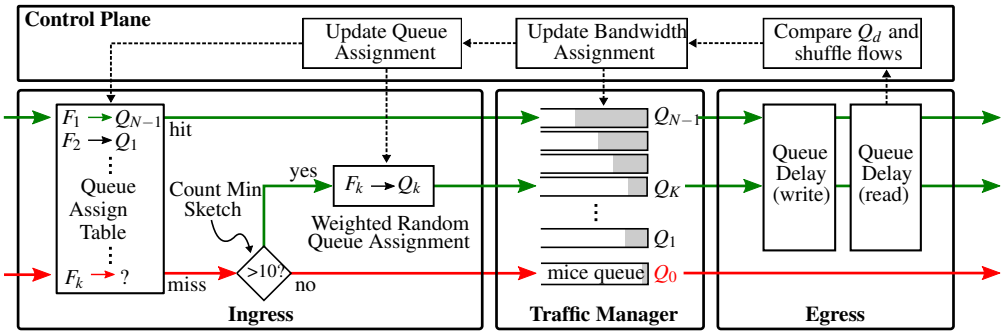


Figure 10: Overview of *Santa* prototype implementation.

5.3 Shuffling and bandwidth allocation

The CP uses information from the Queue Delay structure to make shuffling decisions and compute per-queue bandwidth allocations. It then uses Tofino driver APIs to update entries in the Q-Assign Table and the range-match table for initial weighted random queue assignment. Per-queue bandwidths are configured by programming the switch traffic scheduler's dynamic weighted round-robin mechanism.

6 Evaluation

In this section, we evaluate how well *Santa* can achieve approximate performance isolation and compare it to other common AQMs. Since *Santa* aims to approximate the performance isolation achieved by Fair queuing (FQ), we also compare it to state-of-the-art approximate FQ schemes AHAB [33], SFQ [14], and Cebinae [59].

We note that as performance isolation is not something that other AQMs have explicitly been designed for, the comparison with *Santa* would not be entirely fair. Nevertheless, the current version of *Santa* is also not fully optimized. Our implementation is a proof-of-concept prototype, and the goal is to observe how our approach compares with existing approximate FQ algorithms when it comes to performance isolation.

We also investigate how *Santa*'s performance is affected by the number of available queues, and how well *Santa* can straddle the trade-off space between a single FIFO queue and perfect FQ. Finally, we discuss the scalability of our current implementation of *Santa* in P4 on an Intel Tofino switch.

6.1 How well can *Santa* provide approximate performance isolation?

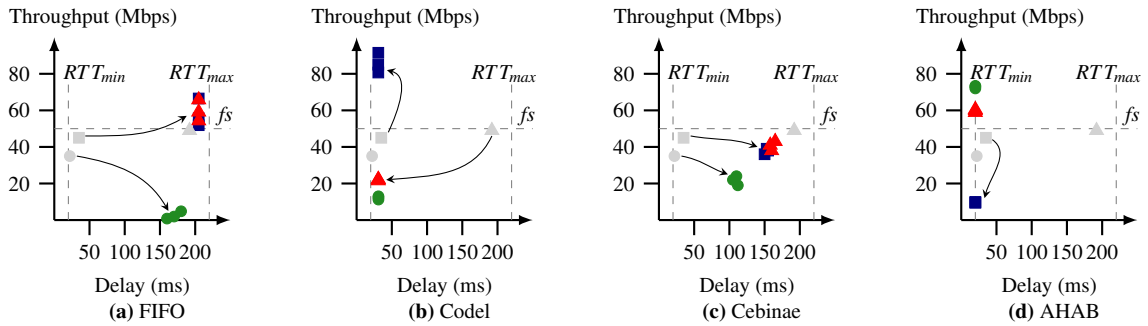
To evaluate how closely *Santa* can achieve performance isolation compared to other AQMs, we measure the throughput delay trade-offs for flows under *Santa*, and compare the observed throughput-delay trade-offs, with that for other AQMs (FIFO, Codel [43], Cebinae [59], AHAB [33], SFQ [14], FQ).

To do so, we launch 9 flows (3 each of CUBIC, BBR, and Vegas) through a fixed capacity 450 Mbps bottleneck link, with a 10 BDP buffer. All 9 flows are launched at the same time and have a minimum RTT of 20 ms. The flows are run concurrently for 1 minute. For FQ, each flow gets its own queue with the fairshare (50 Mbps) bottleneck bandwidth. We configure *Santa* to run with 4 queues: 3 shuffling queues and a mice queue. We plot the results in Figure 11.

Since we want to achieve *performance isolation*, we effectively want to minimize the displacement between the achieved operating point for a CCA from its desired operating point when competing with the other flows. Like before, FIFO (Figure 11a) does not perform well. Vegas flows are starved for bandwidth, and BBR flows suffer from high delays. Codel (Figure 11b) does not fare much better as well. While all flows maintain low delays, loss-sensitive CUBIC and Vegas flows have low throughputs, since Codel effectively behaves like a shallow buffer. BBR, which is mostly loss-agnostic, gains a disproportionately large share of the bottleneck bandwidth.

Existing state-of-the-art approximate FQ AQMs do not fare much better. Cebinae [59] aims for max-min fairness and approximates FQ by taxing bottlenecked flows based on their past bandwidth shares. While this works well for maintaining fairness between CUBIC and BBR flows, Cebinae often wrongly infers Vegas flows as non-bottlenecked flows, resulting in them receiving less than their fair share (see Figure 11c). Since Cebinae is only concerned about bandwidth fairness and does not try to isolate different flows, both BBR and Vegas suffer from large queuing delays. Other approximate fair queuing schemes like AHAB [33] (Figure 11d) and SFQ [14] (Figure 11e) suffer similarly. We can see from Figure 11f that *Santa* is able to achieve almost the same performance isolation as FQ (Figure 11g).

Packet Reordering. Since *Santa* shuffles flows across queues, it can be susceptible to packet reordering. These can manifest as significant performance hits if loss-based CCAs perceive this reordering as packet losses and slow down in response to them. In our experiments, we do observe occasional packet reordering between rounds in *Santa*. However, thanks to the standardization of RACK-TLP [15], the TCP stack is pretty robust to these packet reordering and seldom considers them to be legitimate packet losses. Moreover, these reordering events become more infrequent as the flows' queue assignments converge and they stop shuffling.



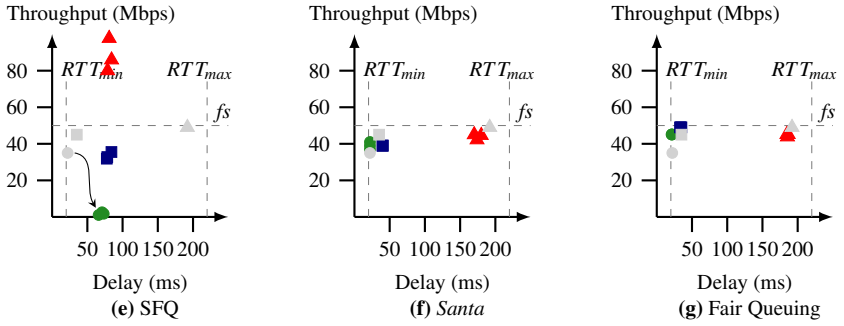


Figure 11: 9 long-running flows (3 each of ▲CUBIC, ■BBR, and ●Vegas) competing in a FIFO, Codel [43], Cebinae [59], AHAB [33], SFQ [14], *Santa*, and FQ bottleneck.

Convergence time for queue assignments. In practice, we observed that the convergence time increases with the number of queues. In the worst case, a flow can take up to K rounds to converge to a queue assignment, where K is the total number of *Santa* queues. In the experiment described in Figure 11f, the flows converge to an assignment within 2 to 3 rounds across multiple trials. For higher flow churn networks, this convergence time can be drastically reduced by setting shorter round intervals.

Scaling to larger number of flows. To understand the impact of a larger number of flows on performance isolation, we launched 90 flows, with 30 flows each of CUBIC, BBR, and Vegas for *Santa* with 3 queues. The total bottleneck bandwidth was set to 1.8 Gbps, which works out to be a fair share rate of 20 Mbps. The flows have a minimum RTT of 20 ms and were run for 1 minute. We plot the results in Figure 12a.

While all flows approximately receive their fair share of bandwidth and Vegas flows saw the lowest delays. BBR flows did not operate at their ideal Kleinrock point, but saw higher than expected delays. On investigation, we found that this was because the BBR flows were taking longer than expected to converge into their own separate queues, and spent most of the time competing with other CUBIC flows. However, over a longer time horizon, the average delay for the BBR flows would reduce as they spent more time in their own queues. This behavior is consistent even when we run *Santa* with more queues (Figure 12b).

We note here that we did not see this behavior earlier in Figure 11f. This is because we weren't able to set 10 BDP buffers for the 90-flow experiment, like we did for the experiment in Figure 11, due to buffer capacity constraints for a port

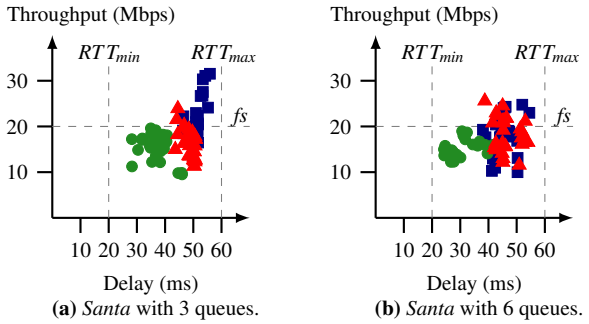


Figure 12: *Santa*'s performance with larger number of flows (90 flows, 30 each of ▲CUBIC, ■BBR, and ●Vegas).

on the switch. For the 90-flow experiment, the buffer size was about 2 BDP. CUBIC and BBR are known to be a lot fairer to each other at these smaller buffer sizes [41]. This suggests that buffer sizing is crucial for making *Santa* work optimally.

6.2 Impact of Number of *Santa* Queues

Given how *Santa* aims to approximate performance isolation, we can think of *Santa* occupying a trade-off space between FIFO and FQ in the throughput-delay plane. To demonstrate this, we ran 6 flows (2 each of CUBIC, BBR, and Vegas) for *Santa* with different number of queues. As we can see from Figure 13, *Santa* with a different number of *Santa* queues straddles the continuum between FIFO and FQ, when it comes to performance isolation. *Santa*-1 functions equivalent to a FIFO queue, and *Santa*-6 is approximately equivalent to FQ.

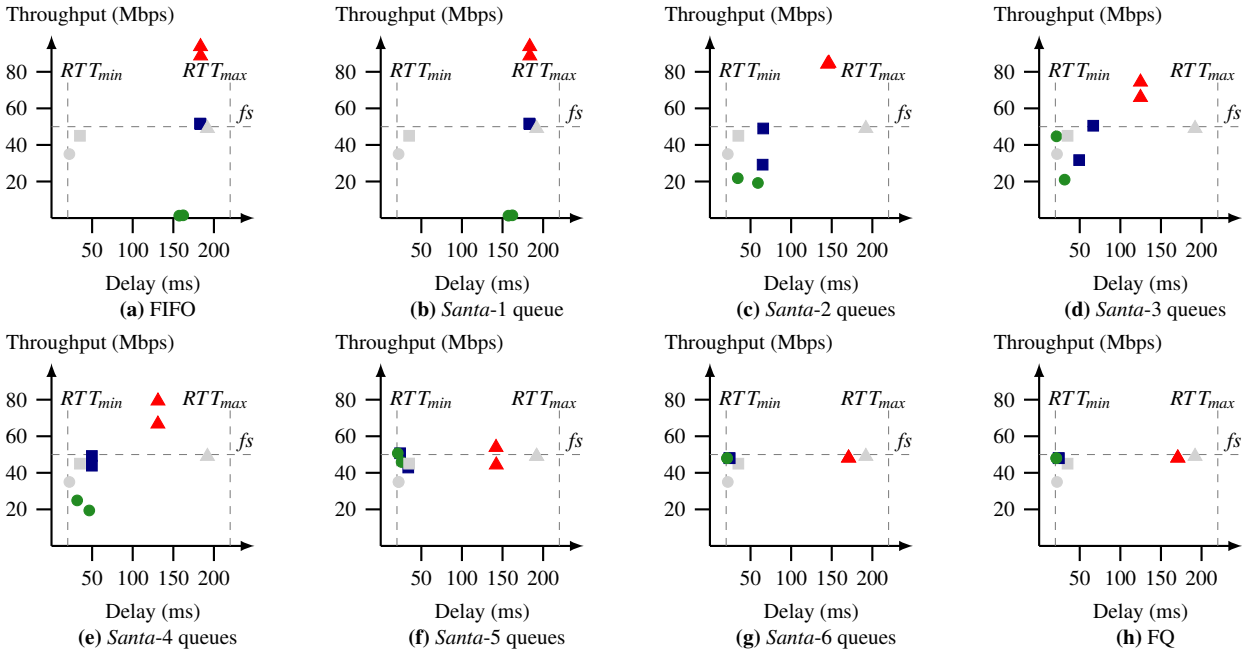


Figure 13: The *Santa* continuum. 6 flows (2 each of ▲CUBIC, ■BBR, and ●Vegas) competing in FIFO, FQ, and different instances of *Santa*.

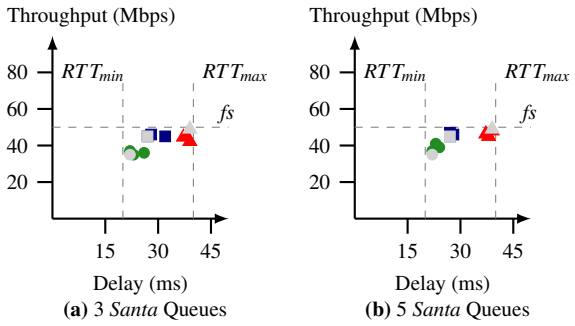


Figure 14: 9 Long-running flows (equal shares of ▲CUBIC, ■BBR, and ●Vegas) competing in a *Santa* bottleneck with a shallow 1 BDP buffer.

6.3 Buffer Sizing

While the experiments so far have been conducted in deep 10 BDP buffers, *Santa* works well in shallower 1 BDP buffers as well. To demonstrate this, we re-ran the experiments described in §6.1 but in shallower 1 BDP buffers. We plot the results of these experiments in Figure 14. *Santa* continues to provide good performance isolation between the competing classes of flows in both 3-queue and 5-queue configurations.

We note here that while all the experiments presented in this paper equally divide the available buffer between all the queues, this is a configurable parameter in *Santa*. One can imagine provisioning more buffer for queues that are expected to contain more aggressive flows to minimize packet loss - however we leave this as future work.

6.4 Scalability

We prototype *Santa* with scalability in mind. While maintaining per-flow state at Internet-scale is typically infeasible [33], we achieve this by leveraging probabilistic data structures (Count-Min Sketch [17]) for flow packet counts, frequent entry flushing, and decaying queue assignments. We evaluate the expected performance of these mechanisms on Internet-scale CAIDA traces [12] via simulations.

Table 1: Hardware resource consumption on the Intel Tofino for different CMS configs

Resource	santa_1col	santa_2col	santa_3col	santa_4col
SRAM	22.1%	25.5%	29%	32.4%
Hash Bits	6.3%	7.4%	7.7%	8.1%
Hash Dist. Unit	19.4%	23.6%	25%	26.4%
VLIW Ins.	4.9%	5.2%	5.7%	6.3%

Count-Min Sketch. The Count-Min Sketch (CMS) enables *Santa*'s scalability by filtering out roughly 90% of Internet flows which are mice flows (§4.1), so *Santa*'s mechanisms are applied only to non-mice flows. We implement a 4-row, 256-column CMS using four 8-bit dataplane registers of 256 entries each. Simulations on CAIDA traces (about 600k flows) show that this setup results in negligible hash collisions, even without flushing for up to 10 seconds. Table 1 shows the *total* dataplane resource footprint when using up to four CMS rows.

Queue Delay Structure. By filtering out ∼90% of flows, the CMS enables the Queue Delay structure to scale to Internet-sized workloads. Since we use 32 bits to store delays, the Queue Delay structure is flushed and refreshed every second to prevent overflow, especially with deep buffers. Given the CMS flushes at the end of a round, the number of collisions for the Queue Delay structure scale with round duration (see Figure 15). To enable this, we maintain two copies (read

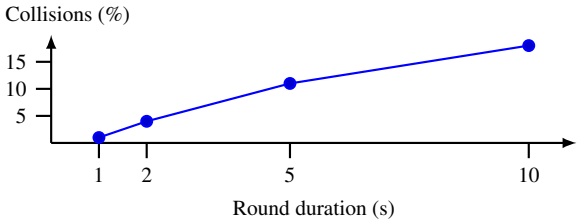


Figure 15: Q-Delay collisions (%) for different *Santa* round durations (assuming CMS flushes at the end of the round), on CAIDA traces using 4 queues.

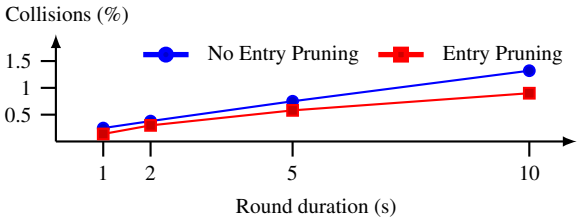


Figure 16: Q-Assign table's memory footprint for different *Santa* round durations on CAIDA traces.

and write) of the two-level registers, alternating them each round—writing to one while the CP reads and flushes the other. This periodic swapping decouples round size from collision rates in the Queue Delay structure.

Q-Assign Table. Without active removal, stale entries in the Q-Assign table can cause collisions and incorrect assignments at scale. To address this, we use an idle timeout of 10 seconds, automatically deleting any entry not accessed within that period. Simulations on CAIDA traces with a 90k-entry table and the same hash function as our P4 implementation show that this pruning significantly reduces collisions for longer *Santa* rounds (see Figure 16).

6.5 *Santa*'s Limitations

While we are confident that *approximate performance isolation* is a valuable goal for modern Internet AQMs, *Santa* represents just one way to achieve it. *Santa* does have its limitations.

More dynamic workloads. We have primarily analyzed long flows in this paper, but *Santa* might find it challenging to handle significant flow churn, even with short round durations. In our experience, tuning *Santa*'s shuffling threshold r becomes critical in ensuring flows are not shuffled too often, hurting their performance isolation. Tracking flow liveliness also remains an open challenge, since persistent connections that send intermittently (like video traffic) might not require the same bandwidth as a long-running download. In such cases, there might be ways to capture a flow's throughput demand better and shuffle it accordingly. We do not explore this aspect in this paper.

Naive buffer allocation. As discussed in §6.3, we employ a simple fair buffer allocation strategy between the *Santa* queues. This might not necessarily be optimal, depending on the classes of flows an operator supports. We expect *Santa* to be complimented by approaches like ABM [2] and L4S [26], that are centered around buffer management, and explicit notification from the end-hosts on their desired throughput-delay preferences.

Deployment Context. While *Santa* is scalable enough to handle a large number of flows with a handful of queues, we do not expect a single configuration of *Santa* to be equally effective in all deployment contexts. For example, the configuration of *Santa* evaluated in this paper (10 s round duration, shuffling threshold $r = 2$) is likely to not be very effective in networks where most flows last less than 10 seconds. While this could be mitigated in theory by setting shorter round intervals, we do not expect *Santa* to be very effective in environments that have have high flow churn and a lot of short flows (like datacenter networks).

7 Related Work

The growing CCA heterogeneity on the Internet and its impact on flow-level performance has been the focus of numerous recent studies, particularly those exploring its implications for fairness [51, 57] and buffer sizing [30, 34]. A major concern is that newer CCAs, like BBR, may lead to unfairness when competing with legacy CUBIC flows [35, 54]. The increase in CCA diversity and emerging variations [38], and its apparent inevitability [41], has also spurred discussions on the co-existence of flows on the Internet [53] and CCA standardization [25]. Others also highlight the challenges faced by delay-based CCAs in highly competitive and heterogeneous network environments [6, 23]. To address those issues, optimizations specifically designed for senders have been proposed to improve fairness in bandwidth sharing [7, 36, 42, 45].

While end-host CCA optimizations can improve fairness to some extent, they are fundamentally limited in scope. When a flow experiences unfair bandwidth allocation, its only available strategy at the transport layer is to increase its sending rate aggressively. However, this can lead to severe congestion in the network rather than achieving true fairness. To address this, network-assisted approaches have been explored to regulate overly aggressive flows. DCTCP [5], DCQCN [64], and L4S [26] use ECN signals, while HPCC [32] and PowerTCP [3] use in-network telemetry (INT) as in-network signals. DiffServ [8] assigns different priority levels to flows but depends on the marks given by the endpoints. The network signaling methods are hampered by the fact that internet users may not always comply with the recommended actions.

Active Queue Management. A more direct approach to fairness involves flow isolation, where each flow or user is allocated a separate queue to minimize interference. Fair queu-

ing (FQ [18]) ensures flow-level isolation, but switches today have a limited number of queues available. Thus there is a body of work that attempts to approximate fair queuing using a few queues: priority-based approximations (PIFO [49], SP-PIFO [4]) assign ranks to packets for scheduling and are less flexible; AIFO [61] uses a single queue and admission control; PIEO [48] uses programmable NICs to offload the scheduling; AFQ [46], PCQ [47], and HCSFQ [62] use multiple queues and specialized data structures to emulate FQ.

The emergence of programmable data planes has enabled practical implementation for AQMs at scale. Traditional AQMs mainly target only the loss-based CCAs by either performing early congestion signaling (RED [20], ARED [19]) or preventing the bufferbloat problem (CoDel [43], PIE [44]). Nimble [50] supports rate-limiting for fixed rates set by the control plane. Flowtamer [37] aims to alter TCP receive window to tame the aggression of the flows, but has scalability concerns and doesn't work with QUIC traffic. Cebinae [60] proposes a low-cost alternative on commodity programmable switches that approximates fair queuing on a large scale by taxing the heavy flows. P4air [51] attempts to provide isolation to different CCAs, but it requires maintaining extensive per-flow data, including queue length and timestamps. P4air proactively drops packets for each flow to gauge its response to packet loss, irrespective of congestion. Moreover, when the flow's group changes, P4air recirculates all packets of the flow, potentially impacting actual bandwidth.

Fair Queuing. To achieve fairness between flows, fair queuing (FQ [18]) isolates each flow by queuing it individually, thus reducing interference between them. In theory, this works perfectly for isolating flows, but switches today have a limited number of queues available. Thus there is a body of work that attempts to approximate fair queuing using a few queues: priority-based approximations (PIFO [49], SP-PIFO [4]) assign ranks to packets for scheduling and are less flexible; AIFO [61] uses a single queue and admission control; PIEO [48] uses programmable NICs to offload the scheduling; AFQ [46], PCQ [47], and HCSFQ [62] use multiple queues and specialized data structures to emulate fair queuing. However, these algorithms employ a uniform handling approach and do not differentiate between the different types of CCAs and their goals. The fundamental issue with these approaches is that they intend to approximate the incorrect aspect of fair queuing; they aim to achieve better bandwidth fairness instead of isolation.

Beyond bandwidth fairness. Almost all of the methods mentioned above focus on bandwidth equalization for fairness, but a more nuanced understanding of fairness is required beyond simply dividing bandwidth equally. Brown et al. challenge the effectiveness of TCP-friendliness in improving the CCA ecosystem [11]. They propose an alternative bandwidth allocation approach aligned with commercial agreements [10]. Zapletal et al. argued that users primarily care about flow completion time (FCT) rather than strict bandwidth fairness, suggesting that an imbalanced bandwidth allocation does not necessarily degrade user experiences [63]. In the past, arguments have also been made to view the congestion control design space in the Network Utility Maximization (NUM) paradigm [29]. However, a fundamental challenge in applying NUM to real-world networks is that the utility functions of CCAs (often depending on parameters such as delay and throughput) are typically unknown.

8 Discussion

In this paper, *Santa* presents a new *approximate performance isolation*-driven paradigm for AQMs. This presents several avenues for further discussion.

Handling bad actors. Protocols like TCP Brutal [1], which aggressively seize bandwidth with little regard for fairness, can significantly harm well-behaved flows. *Santa* can be extended to isolate such bad actors in a "hell queue," thereby limiting their impact on others. This is a concrete direction for network-layer mitigation [52] of selfish behavior by imposing certain penalties.

***Santa*'s impact on CCA design.** Currently, CCA innovation is often hampered by the need to remain competitive. Many CCAs switch to a CUBIC-like mode once they detect buffer fillers [7, 23]. Since *Santa* will allow CCAs with different throughput-delay preferences to co-exist, CCAs do not need to ensure that they are competitive with CUBIC. Instead, *Santa* allows a new CCA to optimize for its own desired throughput-delay target. The opens up the possiblity of new algorithms. For example, a new version of BBR that achieves even lower delay could potentially become practically deployable on the Internet.

Rethinking fairness. Traditional notions of fairness, most notably TCP-friendliness, have long been used to evaluate CCAs. However, in today's increasingly heterogeneous Internet, these definitions fall short of capturing the complexity of modern traffic dynamics. The recent proposals [10, 53] have been arguing for a shift from strict rate-based fairness toward cost-aware or behavior-aware definitions, which may better reflect the realities of diverse protocol behaviors and application requirements. Approximate performance isolation is arguably also a new notion of fairness.

9 Conclusion

Our current implementation of *Santa* is a proof-of-concept that shows it is possible to achieve *approximate performance isolation* using a handful of queues and a simple shuffling strategy. *Santa* explores a new design space for AQMs that can allow different CCAs to co-exist and achieve good performance tradeoffs. *Santa* is open-source and available on GitHub at `https://github.com/NUS-SNL/santa-nsdi-ae`.

10 Acknowledgments

We thank the anonymous NSDI reviewers and our shepherd, Ahmed Saeed, for their valuable feedback. We would also like to thank Xin Zhe Khooi for helping set up the Tofino switches for the authors to work on remotely. This work was supported by the Singapore Ministry of Education Grant MOE-T2EP20222-0006.

References

[1] Tcp-brutal: Congestion control algorithm that increases speed on packet loss, 2023. https://news.ycombinator.com/item?id=38164574.

[2] Vamsi Addanki, Maria Apostolaki, Manya Ghobadi, Stefan Schmid, and Laurent Vanbever. Abm: active buffer management in datacenters. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 36–52, New York, NY, USA, 2022. Association for Computing Machinery.

[3] Vamsi Addanki, Oliver Michel, and Stefan Schmid. PowerTCP: Pushing the performance limits of datacenter networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 51–70, Renton, WA, April 2022. USENIX Association.

[4] Albert Gran Alcoz, Alexander Dietmüller, and Laurent Vanbever. Sp-pifo: Approximating push-in first-out behaviors using strict-priority queues. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 59–76, 2020.

[5] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, page 63–74, New York, NY, USA, 2010. Association for Computing Machinery.

[6] Venkat Arun, Mohammad Alizadeh, and Hari Balakrishnan. Starvation in end-to-end congestion control. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 177–192, 2022.

[7] Venkat Arun and Hari Balakrishnan. Copa: Practical Delay-Based Congestion Control for the Internet. In *Proceedings of NSDI*, 2018.

[8] Yoram Bernet, Peter Ford, Raj Yavatkar, Fred Baker, Lixia Zhang, Michael Speer, Robert Braden, Bruce Davie, John Wroclawski, and Eyal Felstaine. A framework for integrated services operation over diffserv networks. Technical report, 2000.

[9] Lawrence S. Brakmo, Sean W. O'Malley, and Larry L. Peterson. TCP Vegas: New Techniques for Congestion Detection and Avoidance. In *Proceedings of SIGCOMM*, 1994.

[10] Lloyd Brown, Albert Gran Alcoz, Frank Cangialosi, Akshay Narayan, Mohammad Alizadeh, Hari Balakrishnan, Eric Friedman, Ethan Katz-Bassett, Arvind Krishnamurthy, Michael Schapira, et al. Principles for internet congestion management. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages 166–180, 2024.

[11] Lloyd Brown, Yash Kothari, Akshay Narayan, Arvind Krishnamurthy, Aurojit Panda, Justine Sherry, and Scott Shenker. How i learned to stop worrying about cca contention. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*, pages 229–237, 2023.

[12] CAIDA. The CAIDA UCSD statistical information for the CAIDA anonymized internet traces. https://www.caida.org/data/passive/passive_trace_statistics.xml, 2022. Accessed: [access date].

[13] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. BBR: Congestion-based Congestion Control. *CACM*, 60(2):58–66, 2017.

[14] Wei Chen, Ye Tian, Xin Yu, Bowen Zheng, and Xinming Zhang. Enhancing fairness for approximate weighted fair queueing with a single queue. *IEEE/ACM Transactions on Networking*, 2024.

[15] Yuchung Cheng, Neal Cardwell, Nandita Dukkipati, and Priyaranjan Jha. The rack-tlp loss detection algorithm for tcp. *RFC 8985*, 2021.

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

[17] Graham Cormode. Count-min sketch., 2009.

[18] Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and simulation of a fair queueing algorithm. *ACM SIGCOMM Computer Communication Review*, 19(4):1–12, 1989.

[19] Sally Floyd, Ramakrishna Gummadi, Scott Shenker, et al. Adaptive red: An algorithm for increasing the robustness of red's active queue management, 2001.

[20] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on networking*, 1(4):397–413, 1993.

[21] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking (ToN)*, 1(4):397–413, 1993.

[22] Cheng Peng Fu and S. C. Liew. TCP Veno: TCP Enhancement for Transmission over Wireless Access Networks. *IEEE JSAC*, 21(2):216–228, 2006.

[23] Prateesh Goyal, Akshay Narayan, Frank Cangialosi, Srinivas Narayana, Mohammad Alizadeh, and Hari Balakrishnan. Elasticity detection: A building block for internet congestion control. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 158–176, 2022.

[24] Sangtae Ha, Injong Rhee, and Lisong Xu. CUBIC: a new TCP-friendly high-speed TCP variant. *Operating Systems Review*, 42:64–74, 07 2008.

[25] ICCRG. Passive TCP Identification for Wired and Wireless Networks: A Long-Short Term Memory Approach. 2024. https://datatracker.ietf.org/doc/html/draft-briscoe-iccrg-prague-congestion-control.

[26] IETF. Low Latency, Low Loss, Scalable Throughput (L4S) Internet Service: Architecture. 2022. https://www.ietf.org/archive/id/draft-ietf-tsvwg-l4s-arch-16.html.

[27] Van Jacobson. Berkeley tcp evolution from 4.3-tahoe to 4.3-reno. *Proceedings of the 18ˆ< th> Internet Engineering Task Force, September 1990*, 1990.

[28] Van Jacobson and Michael J. Karels. Congestion avoidance and control. In *Proceedings of SIGCOMM*, Stanford, CA, Aug 1988.

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

[30] Elie Kfoury, Jorge Crichigno, and Elias Bou-Harb. P4bs: Leveraging passive measurements from p4 switches to dynamically modify a router's buffer size. *IEEE Transactions on Network and Service Management*, 2023.

[31] Leonard Kleinrock. Power and deterministic rules of thumb for probabilistic problems in computer communications. In *ICC 1979; International Conference on Communications, Volume 3*, 1979.

[32] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. Hpcc: high precision congestion control. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 44–58, New York, NY, USA, 2019. Association for Computing Machinery.

[33] Robert MacDavid, Xiaoqi Chen, and Jennifer Rexford. Scalable real-time bandwidth fairness in switches. *IEEE/ACM Transactions on Networking*, 32(2):1423–1434, 2024.

[34] Imtiaz Mahmud, George Papadimitriou, Cong Wang, Mariam Kiran, Anirban Mandal, and Ewa Deelman. Elephants sharing the highway: Studying tcp fairness in large transfers over high throughput links. In *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, pages 806–818, 2023.

[35] Robin Marx, Wim Lamotte, Jonas Reynders, Kevin Pittevils, and Peter Quax. Towards QUIC debuggability. In *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*, pages 1–7, 2018.

[36] Tong Meng, Neta Rozen Schiff, P Brighten Godfrey, and Michael Schapira. Pcc proteus: Scavenger transport and beyond. 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*, pages 615–631, 2020.

[37] Ayush Mishra, Harsh Gondaliya, Lingesh Kumaar, Archit Bhatnagar, Raj Joshi, and Ben Leong. Poster: Expanding the design space for in-network congestion control on the internet. In *Proceedings of the ACM SIGCOMM 2024 Conference: Posters and Demos*, ACM SIGCOMM Posters and Demos '24, page 42–44, New York, NY, USA, 2024. Association for Computing Machinery.

[38] Ayush Mishra, Sherman Lim, and Ben Leong. Understanding speciation in quic congestion control. In *Proceedings of the 22nd ACM Internet Measurement Conference*, IMC '22, page 560–566, New York, NY, USA, 2022. Association for Computing Machinery.

[39] Ayush Mishra, Lakshay Rastogi, Raj Joshi, and Ben Leong. Keeping an eye on congestion control in the wild with nebby. In *Proceedings of SIGCOMM*, 2024.

[40] Ayush Mishra, Xiangpeng Sun, Atishya Jain, Sameer Pande, Raj Joshi, and Ben Leong. The Great Internet TCP Congestion Control Census. In *Proceedings of SIGMETRICS*, 2019.

[41] Ayush Mishra, Wee Han Tiu, and Ben Leong. Are we heading towards a BBR-dominant internet? In *Proceedings of IMC*, 2022.

[42] Vikram Nathan, Vibhaalakshmi Sivaraman, Ravichandra Addanki, Mehrdad Khani, Prateesh Goyal, and Mohammad Alizadeh. End-to-end transport for video qoe

fairness. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 408–423. 2019.

[43] Kathleen Nichols, Van Jacobson, Andrew McGregor, and Jana Iyengar. Controlled delay active queue management. Technical report, 2018.

[44] Rong Pan, Preethi Natarajan, Chiara Piglione, Mythili Suryanarayana Prabhu, Vijay Subramanian, Fred Baker, and Bill VerSteeg. Pie: A lightweight control scheme to address the bufferbloat problem. In *2013 IEEE 14th international conference on high performance switching and routing (HPSR)*, pages 148–155. IEEE, 2013.

[45] Dario Rossi, Claudio Testa, Silvio Valenti, and Luca Muscariello. Ledbat: the new bittorrent congestion control protocol. In *2010 Proceedings of 19th International Conference on Computer Communications and Networks*, pages 1–6. IEEE, 2010.

[46] Naveen Kr Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. Approximating fair queueing on reconfigurable switches. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 1–16, 2018.

[47] Naveen Kr Sharma, Chenxingyu Zhao, Ming Liu, Pravein G Kannan, Changhoon Kim, Arvind Krishnamurthy, and Anirudh Sivaraman. Programmable calendar queues for high-speed packet scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 685–699, 2020.

[48] Vishal Shrivastav. Fast, scalable, and programmable packet scheduler in hardware. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 367–379. 2019.

[49] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. Programmable packet scheduling at line rate. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 44–57, 2016.

[50] Vineeth Sagar Thapeta, Komal Shinde, Mojtaba Malekpourshahraki, Darius Grassi, Balajee Vamanan, and Brent E Stephens. Nimble: Scalable tcp-friendly programmable in-network rate-limiting. In *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)*, pages 27–40, 2021.

[51] Belma Turkovic and Fernando Kuipers. P4air: Increasing fairness among competing congestion control algorithms. In *2020 IEEE 28th International Conference on Network Protocols (ICNP)*, pages 1–12, 2020.

[52] Wayne Wang, Diwen Xue, Piyush Kumar, Ayush Mishra, Roya Ensafi, et al. Is custom congestion control a bad idea for circumvention tools? *Free and Open Communications on the Internet*, 2025.

[53] Ranysha Ware, Matthew K Mukerjee, Srinivasan Seshan, and Justine Sherry. Beyond jain's fairness index: Setting the bar for the deployment of congestion control algorithms. In *Proceedings of Hotnets*, pages 17–24, 2019.

[54] Ranysha Ware, Matthew K Mukerjee, Srinivasan Seshan, and Justine Sherry. Modeling bbr's interactions with loss-based congestion control. In *Proceedings of the IMC '19'*, pages 137–143, 2019.

[55] Ranysha Ware, Matthew K. Mukerjee, Srinivasan Seshan, and Justine Sherry. Modeling BBR's interactions with loss-based congestion control. In *Proceedings of IMC*, 2019.

[56] Keith Winstein, Anirudh Sivaraman, and Hari Balakrishnan. Stochastic forecasts achieve high throughput and low delay over cellular networks. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, 2013.

[57] Xinyu Wu, Zhuang Wang, Weitao Wang, and TS Eugene Ng. Augmented queue: A scalable in-network abstraction for data center network sharing. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 305–318, 2023.

[58] Francis Y. Yan, Jestin Ma, Greg D. Hill, Deepti Raghavan, Riad S. Wahby, Philip Levis, and Keith Winstein. Pantheon: the training ground for internet congestion-control research. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018.

[59] Liangcheng Yu, John Sonchack, and Vincent Liu. Cebinae: scalable in-network fairness augmentation. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 219–232, 2022.

[60] Liangcheng Yu, John Sonchack, and Vincent Liu. Cebinae: Scalable in-network fairness augmentation. Proceedings of SIGCOMM, page 219–232, New York, NY, USA, 2022.

[61] Zhuolong Yu, Chuheng Hu, Jingfeng Wu, Xiao Sun, Vladimir Braverman, Mosharaf Chowdhury, Zhenhua Liu, and Xin Jin. Programmable packet scheduling with a single queue. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 179–193, 2021.

[62] Zhuolong Yu, Jingfeng Wu, Vladimir Braverman, Ion Stoica, and Xin Jin. Twenty years after: Hierarchical {Core-Stateless} fair queueing. In *Proceedings of NSDI '21*, pages 29–45, 2021.

[63] Adrian Zapletal and Fernando Kuipers. Slowdown as a metric for congestion control fairness. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*, pages 205–212, 2023.

[64] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale rdma deployments. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 523–536, New York, NY, USA, 2015. Association for Computing Machinery.