Efficient Policy-Rich Rate Enforcement with Phantom Queues

Ammar Tahir
UIUC, Microsoft Research
ammart2@illinois.edu

Prateesh Goyal
Microsoft Research
g.pratish@gmail.com

Ilias Marinos
Microsoft Research
ilias.marinos@microsoft.com

Mike Evans
Microsoft
michaelevans@microsoft.com

Radhika Mittal
UIUC
radhikam@illinois.edu

Abstract

ISPs routinely rate-limit user traffic. In addition to correctly enforcing the desired rates, rate-limiting mechanisms must be able to support rich rate-sharing policies within each traffic aggregate (e.g. per-flow fairness, weighted fairness, and prioritization). This must be done at scale to support the vast magnitude of users efficiently. There are two primary rate-limiting mechanisms – traffic shaping (that buffers packets in queues to enforce the desired rates and policies) and traffic policing (that filters packets as per the desired rates without buffering them). Policers are lightweight and scalable but don't support rich policy enforcement and often provide poor rate enforcement (being notoriously hard to configure). Shapers, on the other hand, achieve desired rates and policies, but at the cost of high system resource (memory and CPU) utilization impacting scalability. This paper explores whether we can get the best of both worlds. We present our system BC-PQP, which augments a policer with (i) multiple phantom queues that simulate buffer occupancy using counters and enable rich policy enforcement, and (ii) a novel burst-control mechanism that enables auto-configuration of the queues for correct rate enforcement. Our system achieves the rate and policy enforcement properties close to that of a shaper with 7× higher efficiency.

CCS Concepts

• **Networks** → *Network management*; Transport protocols; **Middle boxes / network appliances**.

Keywords

Rate Enforcement; Congestion Control; Network Management

ACM Reference Format:
Ammar Tahir, Prateesh Goyal, Ilias Marinos, Mike Evans, and Radhika Mittal. 2024. Efficient Policy-Rich Rate Enforcement with Phantom Queues. In *ACM SIGCOMM 2024 Conference (ACM SIGCOMM '24), August 4–8, 2024, Sydney, NSW, Australia.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3651890.3672267

1 Introduction

Rate limiting is prevalent among network operators and Internet Service Providers (ISPs) [14, 21, 28, 33]. ISPs routinely rate-limit their customers' traffic based on their plans and subscriptions. Cellular service providers also commonly rate limit bandwidth-hungry video streaming traffic for each user in the cellular core, before the traffic hits their radio access network (RAN), so as to not overwhelm the limited RAN resources [1, 28, 33, 48]. Programs like T-Mobile's "Binge on" [48] and Verizon's "Netflix & Max" [52] provide unlimited access to specific video streaming services but limit the subscribers' network traffic outside of those services.

The rate-limiting mechanism must correctly enforce the desired cumulative rate for each traffic aggregate (e.g. set of flows belonging to a given user). In addition to that, it must satisfy two important requirements. First, it should be able to support different rate-sharing policies among flows within each aggregate. For example, enforcing per-flow fairness within an aggregate allows flows using different congestion control algorithms (BBR [17], New Reno [53], Cubic [25], Vegas [13], etc) to compete fairly with one another [36, 41, 42]. It is also often desirable to enforce weighted fair sharing or prioritization within a given user's traffic as per their preferences (e.g. prioritizing video streams or web traffic over bulk downloads). [1] Per-flow fairness is also desired when cellular operators rate limit video streaming sessions, so as to ensure that audio chunks are not head-of-the-line blocked by video chunks (based on our conversations with a large US-based telecom company, this is a highly desirable feature that is difficult to implement for reasons we discuss below).

The second requirement is that the rate and policy enforcement mechanism must be efficient. This requirement stems from the scale at which such systems operate, with a typical ISP supporting thousands of customers.

Rate limiting can be done using two different mechanisms (that are typically implemented in a software middlebox): traffic shaping and traffic policing. Traffic shaping for a given aggregate involves buffering packets in one or more queues, which can be served based on desired policies (e.g. prioritization, round-robin for fairness, weighted round-robin, etc). Traffic shapers are thus able to enforce a rich set of policies. However, as we detail in §2, doing so is costly as it requires buffering packets in memory and pointer chasing at the time of dequeues – this cost materializes as increased utilization of system resources (memory and CPU cycles), which impacts scalability.

Policers, in contrast, are much more lightweight and therefore more scalable. They do not require storing packets, and instead immediately determine whether an incoming packet should be dropped or transmitted depending on whether the incoming traffic's rate exceeds the enforced rate. This is typically implemented using a token-bucket filter, where tokens are added to a fixed-size bucket at

[1] Commercial SD-WAN solutions [4, 5] already provide interfaces for enterprise customers to express such preferences to their ISPs, and there are several research proposals to enable this more broadly [12, 18, 24, 30, 54, 55].

This work is licensed under a Creative Commons Attribution International 4.0 License.

ACM SIGCOMM '24, August 4–8, 2024, Sydney, NSW, Australia

© 2024 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-0614-1/24/08
https://doi.org/10.1145/3651890.3672267

the specified rate (by incrementing a counter) – a packet is allowed to pass through only if there are enough tokens in the bucket (worth the packet size).

By the virtue of being more efficient, policers have emerged as the more popular rate-limiting choice [21]. However, the scalability provided by this choice has come along with notable downsides: (1) Typical policers, by design, do not provide any mechanism for enforcing desired rate-sharing policies within each rate-limited traffic aggregate. (2) Policers are notoriously hard to configure, often leading to poor rate enforcement (with a trade-off between meeting the desired average rate limit vs reducing burstiness and packet drop rates) [21, 28, 50]. Shapers can adequately address these downsides of a policer, but at the cost of lower system efficiency (and scalability).

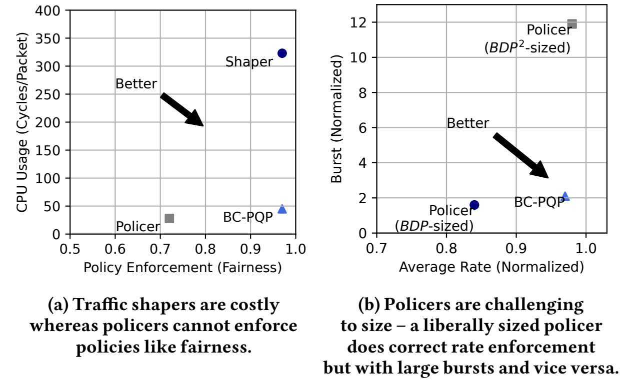
The question we explore in this paper is whether we can get the best of both worlds: *can we have the system efficiency and scalability of a policer, along with the network-level properties (ability to enforce desired rates and policies) of a shaper?*

We answer this question in the affirmative by implementing policers using *phantom queues*. Phantom queues *simulate* the occupancy of a buffer without actually buffering packets, and have been used before for active queue management [8, 31, 32]. We apply a similar concept for policing. A phantom queue-based policer immediately transmits a packet upon arrival if there is enough capacity (worth the packet size) in the phantom queue, and drops it otherwise. Every time it transmits a packet, it enqueues a phantom packet of the same size in the phantom queue – these phantom packets are simply realized as byte counters. It dequeues the phantom queue at the desired rate by decrementing the byte counters.

A policer implemented in the above manner using a single phantom queue mimics the behavior of a token-bucket filter. To enforce different rate-sharing policies, we extend such a policing system for each traffic aggregate to use *multiple* phantom queues – we classify incoming packets into one of these phantom queues (based on flow identifiers in the packet header fields), and immediately transmit or discard the packet based on the queue's occupancy as described above. We dequeue the phantom packets in each of these phantom queues (i.e. decrement the corresponding byte counters) based on the desired policy, e.g. prioritization, round-robin, etc, analogous to a shaper system. We refer to such a phantom queue-based policer as PQP. We show (both analytically and empirically) how PQP can correctly enforce the desired aggregate rate, as well as achieve the desired rate-sharing policies on average, as long as the phantom queues are sufficiently sized.

While a sufficiently sized PQP correctly enforces the desired rates on average (over multiple round-trip times), the instantaneous rates over smaller timescales can burst to much higher values, with the burst increasing with queue size. The minimum queue size required for enforcing correct average rates with phantom queues is very large, to begin with: $O(BDP^2)$ (in comparison to $O(BDP)$ sized buffers required for shaper queues with real packets). The burstiness caused by such a large queue is further aggravated in PQP – with N active phantom queues, the worst-case burst can be N times larger!

We therefore need a mechanism to control the burst while still ensuring correct average rate enforcement. For this, we design a novel burst control mechanism for phantom queues, where we start with sizing each phantom queue to a sufficiently large value. However, if the enqueue rate of the queue exceeds a certain threshold, we vacuously fill up the queue with *magic* phantom packets (that do not correspond to real packets). Filling up the queue in this manner prevents the flow from bursting and induces early drops. At the same time, keeping the queue large (but occupied by the magic packets that drain at the desired dequeuing rate) complies with the queue size requirement for correct average rate enforcement. We refer to this extension of PQP as BC-PQP (for burst-controlled PQP). The rate threshold for vacuously filling up a phantom queue in a BC-PQP system is governed by the rate at which the queue is served (as per the rate-sharing policy). This enables auto-tuning of the queue configuration, as the set of active flows (and consequently the rate assigned to a given phantom queue) changes.



(a) Traffic shapers are costly whereas policers cannot enforce policies like fairness.

(b) Policers are challenging to size – a liberally sized policer does correct rate enforcement but with large bursts and vice versa.

Figure 1: Drawbacks of traffic shapers and policers

We implement our system on a testbed comprising three Linux servers (a sender, a middlebox implementing BC-PQP, and a receiver). The middlebox transparently rate-limits the traffic sent by the sender using a kernel-bypass stack based on Intel's DPDK. Our evaluation (using self-generated traffic as well as real-world applications) shows how BC-PQP achieves the rate and policy enforcement properties close to that of a shaper while being 7 × more efficient (with the efficiency within 1.5-2 × of a standard policer). Through dynamic burst control, BC-PQP further achieves up to 2.5× lower drop rates and up to 18× smaller burst (tail throughput deviation from the desired value) than a policer. BC-PQP is able to enforce a variety of rate-sharing policies including per-flow fairness, weighted fairness, prioritization, and nested combinations of these policies.

2 Background and Motivation

Today, there are two prevalent mechanisms to do rate enforcement: traffic shapers and traffic policers. We describe both of them below.

2.1 Traffic Shapers

Traffic shapers, conventionally implemented on network routers and dedicated hardware appliances, are now often implemented in software middleboxes as virtualized network functions for flexibility. [2] They can support a large set of QoS (quality of service) mechanisms such as prioritization [38], weighted fair queueing (WFQ) [44, 46], etc.

Rate enforcement with traffic shapers. Traffic shapers maintain a separate buffer for each traffic aggregate. An incoming packet gets

[2]Based on our conversation with a large US-based telecom company.

	Traffic Shaper	Traffic Policer
Enqueue	29	28
Dequeue	293	-

Table 1: Breakdown of average number of CPU cycles spent to process a packet with shaper and policer

enqueued into the buffer corresponding to its traffic class (e.g. based on the end-user). If the buffer is full, the packet is dropped. Each such buffer is dequeued at the required rate r (that is the rate that we wish to enforce on that traffic aggregate).

Policy enforcement with traffic shapers. Traffic shapers further divide the buffer for each traffic aggregate into a set of N queues, and dequeue packets from these queues as per the desired policy at a cumulative rate r. For example, to enforce weighted fairness, a deficit round-robin scheduler is often used, which attempts to dequeue $w_i MSS$ bytes from a queue i (if the queue is not empty), before moving on to the next one. Since a packet can be dequeued from the shaper only after MSS/r time, a dequeue call is scheduled periodically every MSS/r. When doing such rate enforcement at scale, typically a timer wheel [51] is used to schedule these dequeue calls efficiently for different shapers.

Inefficiency of shapers: While shapers can achieve very accurate rates and policy enforcement, they can be computationally inefficient to implement. For starters, they require a large amount of memory e.g. for a single traffic shaper with 16 drop-tail queues of size 48 MSS-sized packets, the memory that needs to be reserved is at least 1 MB. When doing rate enforcement at scale for thousands of shapers, memory bottlenecks start to arise.

In terms of CPU cycles, fewer cycles are spent when processing a packet at the time of enqueue. This is because on modern x86 CPUs, that feature Intel's Data Direct IO (DDIO) technology, incoming packets are DMAed to the CPU's Last Level Cache (LLC) and the CPU can classify them without incurring cache misses. However, the dequeue operation is an order magnitude more expensive. Since the packets cannot be dequeued immediately they are eventually evicted to the main memory (DRAM). The CPU constantly polls all available shapers (i.e., queues) and instructs the NIC to DMA packets out when allowed. While this could be a relatively efficient operation if the shaper maintains a single FIFO queue and the packets are buffered to contiguous memory, it is quite expensive when enforcing policies like DRR or prioritization with multiple queues: in such cases, packets are not necessarily dequeued in the order that they were received and the CPU needs to lookup for each packet individually from different locations in memory before instructing the NIC to transmit them. Hence this operation can cause frequent CPU stalls manifesting as increased cycles spent per packet.

Table 1 breaks down the average number of CPU cycles spent to process a packet at enqueue and dequeue with a shaper maintaining 16 queues served using a round-robin policy, when compared to a policer. Throughout the paper, we use CPU efficiency as a proxy for scalability. If a rate-limiting mechanism consumes a higher number of CPU cycles per packet, it will require a proportionally larger number of cores (and servers) to meet the scalability requirements.

Figure 2a further compares shapers with policers in terms of their efficiency and degree of policy enforcement. While a shaper is more effective at policy enforcement (it achieves higher fairness, in this case), it spends far more CPU cycles per packet compared to a policer. We discuss these benefits and limitations of a policer in more details next.

2.2 Traffic Policer

Unlike traffic shapers that regulate traffic by buffering and delaying packets, policers enforce rate limiting by dropping packets when a certain rate is exceeded. By avoiding packet buffering, policers are quite lightweight and scale better than traffic shapers on conventional hardware. Traffic policing is done using token bucket filters (TBF) [21]. Policers maintain a TBF for each traffic aggregate. In a TBF, tokens are added to a bucket of size B at the desired rate r. For each packet of size s that arrives, if there are at least size s worth of tokens in the bucket, the packet consumes those and is immediately forwarded. Otherwise, it is dropped. This way, the policer does not need to store any packets and hence eliminates the overhead of memory management-related bottlenecks.

While providing an excellent option in terms of system-level efficiency, traffic policers suffer from two key limitations:

1. Poor rate enforcement. Traffic policers are notoriously hard to configure [21]. An inappropriately small bucket size (B) can result in an average rate lower than the desired one. Whereas, an appropriately large bucket size can cause a large burst in the instantaneous rates, which can be orders of magnitude higher than the desired rate. Figure 2b illustrates the tradeoff between the steady-state rate and peak rate due to bursts allowed by a policer. This can be quite problematic: bursty behavior can result in packet drops and unfairness which can severely impact the users' quality of experience. As per our analysis in §3.5 (and as per what prior studies have reported [21, 50]), such a trade-off is fundamental for any TBF-based policer with a statically configured bucket size.

2. Lack of policy enforcement. By design, traditional traffic policers (that use TBFs) can only support simple rate enforcement on a traffic aggregate, without providing any means for controlling how this aggregate rate is further subdivided between different flows or applications within the aggregate.

Recent work, called FairPolicer, has explored the idea of augmenting TBF-based policers with per-flow fairness across N flows [41, 42]. It achieves this by effectively dividing the bucket B equally across the N flows, and distributing the tokens equally between buckets of active flows. However, it is not immediately clear how to extend the point solution provided by FairPolicer to support more general rate-sharing policies (e.g. weighted or hierarchical fairness). Moreover, due to a statically configured bucket size, it suffers from large bursts and poor rate and policy enforcement under many scenarios. Our evaluation in §6 provides a detailed comparison with FairPolicer.

2.3 Our Goals

Based on the applications and use cases we have discussed so far, we need a rate enforcement mechanism that:

- Does rate enforcement correctly without large bursts.
- Allows arbitrary rate-sharing policies within the aggregate.
- Is scalable, efficient, and lightweight.

Shapers satisfy the first two goals but fail on the third goal. Policers satisfy the third goal but fail on the first two.

In the next few sections, we present our system that augments policer with phantom queues to meet all of the above goals. As shown

in Figure 1, our system has efficiency comparable to a policer, and rate and policy enforcement capabilities comparable to a traffic shaper.

3 Policers with Phantom Queues

We augment a policer with *phantom queues* to realize different rate-sharing policies. Prior work has used the concept of phantom (or virtual) queues for active queue management [8, 31, 32] – these queues simulate the occupancy of the link with lower utilization using packet counters (without actually buffering the packets), enabling early signaling (via ECN or packet drops) when the simulated buffer is full. We apply a similar concept for policing as follows.

3.1 Policing with a Single Phantom Queue

We can realize such a policing system using a phantom queue by considering a (simulated) buffer of size B, served at rate r. When a packet of size s arrives, we first check if there is sufficient capacity in the phantom queue's simulated buffer. If the remaining capacity in the phantom queue is at least s, we immediately transmit the (real) packet and enqueue a "phantom" packet of size s in the phantom queue on its behalf. If the phantom queue is full (or its remaining capacity is less than s), we drop the (real) packet. We dequeue the phantom packets in the phantom queue at rate r. Notice how *we do not buffer any real packets* – we either transmit or drop the real packets right away upon arrival. The phantom packets in the phantom queue are simply maintained as byte counters that get incremented and decremented upon phantom enqueue and dequeue events respectively. Moreover, unlike a shaper, where we need to regularly dequeue packets based on rate r, phantom dequeues can be batched and done only when the phantom queue becomes full. Such a policing system, implemented using a single phantom queue, essentially works in the same way as a token bucket filter of size B with rate r, as described in section 2.2.

3.2 Policing with Multiple Phantom Queues

Once we realize a policer as a phantom queue, we can extend it to a system of N phantom queues (analogous to a shaper with N queues) to realize different rate-sharing policies. Figure 2 compares such a system with an analogous traffic shaper. When a packet of size s arrives, we classify it into one of the N queues (say Q_i with a buffer size of B_i) based on packet header fields (e.g. flow ID, a hash of source-destination addresses, etc). If the remaining buffer capacity in Q_i is at least s, we transmit the real packet and enqueue the corresponding phantom packet in Q_i by incrementing its byte counter by s. If the remaining buffer capacity in Q_i is less than s (after accounting for any pending phantom dequeues), we drop the packet.

We dequeue the phantom packets from the phantom queues (by decrementing the corresponding byte counters) as per the desired policy. For example, to enforce per-flow fairness, we maintain a phantom queue for each flow (or approximate it by hashing the flow identifiers in the packet header fields into one of the N queues), and dequeue phantom packets from the occupied phantom queues in a round-robin manner at a cumulative rate of r. This phantom system (maintained via counters) is exactly analogous to a shaper system that enforces fairness via per-flow queues storing real packets served in a round-robin manner at a cumulative rate of r. We can similarly emulate other policies – weighted fairness (doing weighted round-robin between occupied phantom queues with differing weights), prioritization (dequeuing from lower priority phantom queue only when the higher priority queue is unoccupied), or hierarchical combinations of these (e.g. dividing the queues into two classes, with the first class of queues having 2× the weight of the second class, and enforcing per-flow fairness across the queues within each class).

We refer to such a policing system with multiple phantom queues as PQP. We further use the term "analogous shaper system" to refer to a hypothetical shaper system that applies the same enqueuing and dequeuing policies on real packets as PQP does on phantom packets.

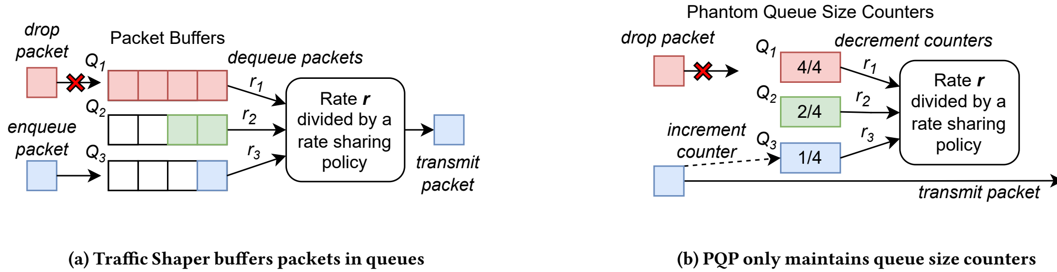
3.3 Scope and Properties of PQP

Notice how PQP directly enforces the desired policies on *phantom* packets (that are maintained as counters). These policies *indirectly* influence real packets by changing the phantom queue occupancy, thereby determining whether the real packets must be transmitted or dropped. This discrepancy between real and phantom behavior imposes certain restrictions on the kind of policies PQP can realize.

Restriction #1: No drop after enqueue. The first restriction stems from the fact that PQP decides whether a packet should be transmitted or dropped upon its arrival. If the corresponding phantom queue occupancy allows the packet to be transmitted, that is done right-away, and its phantom copy is enqueued (with the assumption that it will eventually be dequeued). By design, such a system cannot emulate policies where the fate of the packet (whether it should be dropped or transmitted) changes *after* the packet has been enqueued. An example of such a policy is priority dropping – where a queue enqueues packets with differing priorities, dropping the lowest priority packet when it is full.

We, therefore, restrict PQP to emulate a set of N *drop-tail* queues, where each queue Q_i has a fixed size B_i – if the occupancy of Q_i allows the packet to be transmitted (and its phantom copy to be enqueued) upon arrival, then the corresponding phantom packet is guaranteed to be eventually dequeued (with the dequeue time governed by the policy as described in §3.2). This restriction complies with how most policy-rich shaper systems are implemented [2, 44]. Note that we use the term *drop-tail* rather generously – the only requirement being that a (phantom) packet cannot be dropped after it has been enqueued. We need not necessarily wait for Q_i to become full before we drop a packet upon its arrival; we can apply active queue management policies (as we do in §4) or even apply access control-based filters that drop packets upon arrival based on other criteria.

Restriction #2: Rate-sharing Policies. The second restriction stems from the fact that real and phantom packets in PQP are dequeued at different times. So while PQP enforces the desired policies (that an analogous shaper system applies on real packets) on phantom packets, the specific timings do not translate to real packets. As a result, PQP cannot enforce policies pertaining to packet timings or scheduling order – a packet arriving at PQP at time t will either be dropped or transmitted at time t. For example, a shaper served at rate r can ensure that high-priority packets never experience any queuing delay from low priority if all downstream hops have a capacity greater than r. In contrast, PQP can transmit a burst of (real) low-priority packets before transmitting high-priority packets that arrive later (while the phantom low-priority packets wait behind



(a) Traffic Shaper buffers packets in queues (b) PQP only maintains queue size counters

Figure 2: Phantom Queue Policer (PQP) serviced at rate r using a rate sharing policy, and an equivalent traffic shaper.

phantom high-priority ones) – this can cause the high-priority packets to wait behind the burst of low-priority ones at a downstream hop whose link capacity, while greater than r, is lower than the burst rate.

While we cannot control fine-grained packet timings with PQP, we can enforce different *rate-sharing* policies on an average (over longer timescales), in terms of how the cumulative rate r is divided between individual queues. For example, a per-flow fairness policy (implemented as round-robin dequeue from per-flow phantom queues) will serve Q_i roughly at rate $r_i = max(r/N')$, where N' is the number of non-empty queues. A weighted fairness policy will serve Q_i at rate $r_i = \frac{w_i r}{\sum_{Q_j \text{ not empty}} w_j}$, where w_i is the weight of Q_i. A prioritization policy will serve a lower priority queue at the rate of r minus the rate at which the higher priority queues are served (as driven by their packet arrival rates). [3]

PQP, by design, guarantees the following properties, that allow it to enforce such rate-sharing policies on average:

Property 1. *Assuming the set of packets that arrive at a PQP system is exactly the same as the set of packets that arrive at the analogous shaper system, if a packet gets dequeued at time t_d in the shaper system, its phantom copy will also be dequeued at the same time t_d in the PQP system.*

Property 2. *If a (real) packet is transmitted by a PQP, then its phantom copy is eventually dequeued by the PQP.*

Property 3. *If a PQP transmits a (real) packet at time t_e, its phantom copy will be enqueued in phantom queue Q_i at time t_e and will be dequeued at time $t_d = t_e + D(i, t_e)$, where $D(i, t_e)$ is the phantom queuing delay i.e. the time needed to drain the phantom queue build up until time t_e at Q_i.*

We can combine these properties to see how PQP can effectively enforce rate-sharing policies. As per Property 1, if an analogous shaper system divides the rate r between N queues such that Q_i is served at rate r_i (e.g. as dictated by weighted round-robin scheduling, priority scheduling, or their hierarchical combination), then the corresponding PQP system will serve the *phantom* packets in Q_i at rate r_i. As per Properties 2 and 3, if the phantom packets in Q_i are dequeued at rate r_i, then, on an average (over a longer timescale), the corresponding real packets also get served at rate r_i. How much the instantaneous rates of real packets deviate from their ideal phantom counterparts is dictated by the phantom queuing delay, which in turn is governed by the phantom queue size (that controls the amount

[3] The precise rates at which each queue is served would depend on the rate at which packets get enqueued in each queue, which dictates the max-min weighted fair share rates as well as the spare capacity (in case of prioritization).

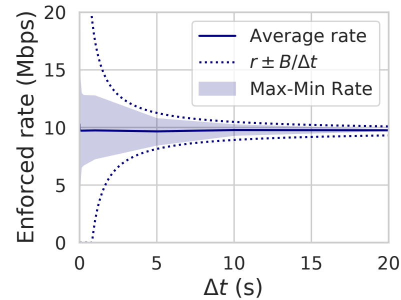


Figure 3: Enforced rate as a function of Δt (for a Reno flow with $B = 1000KB$ at 10 Mbps).

of burst allowed the PQP system) – we analyze this more formally in §3.4 and devise a mechanism to effectively limit the burst in §4. Further, note that Property 1 holds under the assumption that the set of input packets is the same in the PQP system and the analogous shaper system. However, timing deviations in when a packet actually gets transmitted impact the feedback loop of congestion control algorithms, thereby affecting the packet arrival rates. Our evaluation in §6 shows how the rate and policy enforcement with PQP, in spite of this effect, closely matches the analogous shaper system.

3.4 Bounds on Rate and Policy Enforcement

Consider a phantom queue Q of size B that is serviced at rate r. Let the length of the phantom queue (the number of bytes in the queue's simulated buffer) at time t be given by $L(Q,t)$. This queue length governs the phantom queuing delay of a packet transmitted at time t.

Theorem 1: *Over any time interval $\Delta t = t_2 - t_1$, as long as phantom queue Q occupancy does not go to zero i.e. $L(Q,t) > 0, \forall t \in (t_1, t_2)$, then the rate enforced over duration Δt is bounded by $(r \pm B/\Delta t)^+$.*

Proof: Given, $L(Q,t) > 0$ over $t_1 < t < t_2$, Q continues to drain phantom packets at rate r. Over time Δt, it drains $r\Delta t$ bytes. Therefore, the amount of data, $A(t_1, t_2)$, that Q accepts during duration (t_1, t_2) can be given as:

$A(t_1,t_2) = (L(Q,t_2) - L(Q,t_1) + r\Delta t)^+$

Here, $(v)^+ = \max(0,v)$.

Since $0 < L(Q,t) \leq B$, we can find upper and lower limits on the number of accepted packets as follows:

Upper Limit: $L(Q,t_1) = 0$ and $L(Q,t_2) = B$

$$A_{max}(t_1,t_2)_+ = r\Delta t + B$$

Lower Limit: $L(Q,t_1)=B$ and $L(Q,t_2)=0$

$$A_{min}(t_1,t_2)=(r\Delta t-B)^+$$

Thus, number of accepted packets over duration Δt is given as:

$$A(t_1,t_2)=(r\Delta t\pm B)^+$$

Dividing the above equation by Δt gives the actual enforced rate, r' over duration Δt. As Δt grows, the actual enforced rate comes closer to the phantom queue draining rate of r:

$$r'=\lim_{\Delta t\to\infty}\frac{A(t1,t2)}{\Delta t}=\lim_{\Delta t\to\infty}(r\pm\frac{B}{\Delta t})^+=r$$

This can be seen in Figure 3 for a Reno flow throttled to 10 Mbps. The dotted lines represent the theoretical error in rate from theorem 1, whereas the shaded region is the actual error in enforced rate. As we increase Δt, the error in enforced rate becomes smaller and smaller.

The above result is provably achieved only as long as the phantom queue remains non-empty over the duration Δt. If the phantom queue is empty over some time duration, the enforced rate will accordingly be lower than r. This can happen if the packet arrival rate (traffic demand) is itself lower than r. It can also happen if the traffic demand is higher than r, but the queue is not sufficiently sized, causing it to empty out under a typical flow's congestion control behavior – we discuss this further in §3.5.

Now consider a set of N phantom queues, serviced at a cumulative rate r, where r is divided across individual phantom queues Q_i, each serviced at rate r_i as per the desired policy (as discussed in §3.3). If each queue is sized by B_i, we can use the above theorem to show the following bounds on such a system: *If any phantom queue Q_i, whose occupancy does not go to zero over a duration Δt, has a phantom dequeue rate of r_i, it has an enforced rate of $r_i' = (r_i \pm \frac{B_i}{\Delta t})^+$ over duration of Δt.* Moreover, if we sum this for all queues, we get bounds on the overall rate enforced for the aggregate as: $r' = (r \pm \frac{\sum_i^n B_i}{\Delta t})^+$. So, if each phantom queue is sized to be B, the overall rate enforced is $r'=(r\pm N\frac{B}{\Delta t})^+$

Takeaways. We have the following two key takeaways from these theorems: (i) The average rate that PQP enforces on real packets will match the desired rates (enforced on phantom packets) over long enough timescales, as long as the phantom queue remains occupied. (ii) The discrepancies in these two rates over a smaller timescale is bounded by the size of the phantom queues. Very large queue sizes can cause instantaneous enforced rates to be much higher than the desired phantom rates (i.e. cause large bursts). Very small queue sizes, on the other hand, will result in lower than desired instantaneous (and average) rates as this may lead to phantom queue going empty at times. We discuss how phantom queues should be sized next.

3.5 Sizing the Phantom Queues

Guidelines on how to size the phantom queue depend on factors like rate r, RTT, and congestion control protocol used by the flow. We now analyze how phantom queues should be sized for correct average rate enforcement.

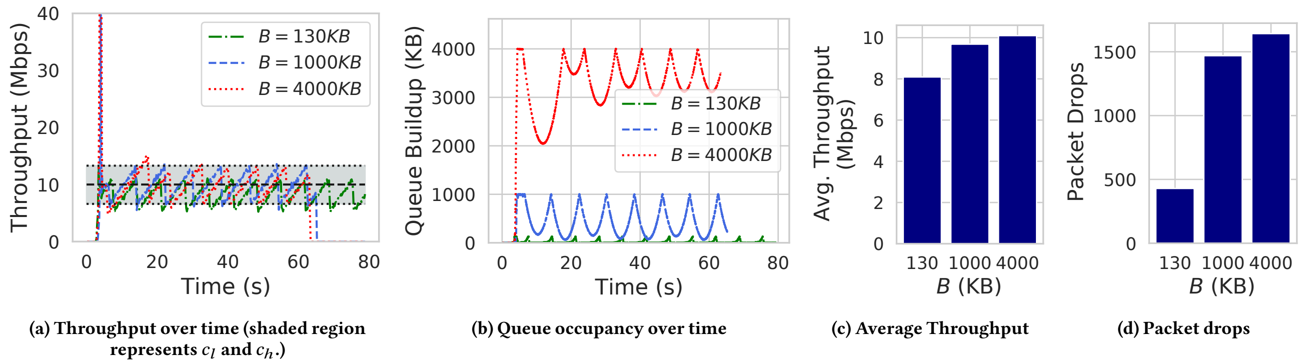
Our bounds on enforced rates in §3.4 were conditioned on the queue remaining occupied over the given time duration. Therefore, in order to achieve these bounds, the phantom queue must be sized such that the congestion control protocol of a backlogged sender (that generates data at a rate higher than the policed rate of r) is able to keep it occupied [4]. This is analogous to how we reason about sizing shaper queues (that manage real packets) [9]. However, the outcome (i.e. the required queue size) is very different for phantom queues, due to the discrepancy between timings in when the phantom packet is dequeued and the real packet is transmitted, and how that affects the congestion control loop.

We consider congestion control protocols frequently used in production today: Cubic (default for most users [19]), New Reno (used by Netflix[47]), and BBR (used by Google and YouTube [3, 10]). The phantom queue size B should be large enough to support any of these protocols. Reno has the largest queue size requirements amongst these protocols (we use the term Reno to refer to both Reno and New Reno protocols, that share the same core logic, other than fast recovery). This means that if we size the phantom queues as per Reno's requirements, we can ensure correct rate enforcement for other protocols too.

Need $O(BDP^2)$ sized phantom queues. The rule-of-thumb for shaper queues (with real packets) requires $O(BDP)$ size to ensure that they remain occupied by a backlogged Reno sender [9], where BDP is the bandwidth-delay product of the network. In contrast, we find that in order to keep a *phantom* queue occupied with a backlogged Reno flow, we need to size it at $O(BDP^2)$. Specifically, we find that for correct rate enforcement for a Reno flow, the phantom queue size should be at least $\frac{BDP^2}{18}\times MSS$ bytes, where $BDP=r\times RTT$, with r being the desired rate (at which the phantom queue is dequeued) and RTT is the flow's round-trip time. This comes from our analysis (detailed in Appendix A) that shows that in order to maintain an average enforced rate of r, the instantaneous rate of the Reno flow should vary between $\frac{2r}{3}$ and $\frac{4r}{3}$ in the steady AIMD (additive increase multiplicative decrease) phase, and a phantom queue with buffer size at least $\frac{BDP^2}{18}\times MSS$ bytes is needed to support this rate variation. It should also be noted that, in the steady state, the upper limit on the size of the buffer is not important. Once the phantom queue becomes full, it automatically makes room for more packets at the rate of r.

Why not BDP-sized queues? The reason for the larger buffer size requirement with the phantom queue (when compared to the rule-of-thumb for regular queues with real packets) stems from the fact that phantom queue does not have any queuing delay for real packets. In a real queue, when the flow's congestion window (*cwnd*) exceeds *BDP*, additional packets are queued and dequeued at rate r, thus incurring a queuing delay additional to the base RTT. Only when the acknowledgments for all packets has reached the sender, *cwnd* is incremented by 1. By the time this additional packet due to the *cwnd* update arrives, all the packets for the previous *cwnd* had already been transmitted, thus the standing queue increases by only 1 packet after every *cwnd* update. In the phantom queue, however, acknowledgment for all *cwnd* amount of packets reaches within the base *RTT* time (irrespective of however long it takes the phantom queue to drain). With the shorter feedback loop that excludes any queuing delay, by the time packets for the next *cwnd* arrive, $(cwnd-BDP)^+$ phantom packets from the previous round are also present. So, where in a physical queue, queue build-up increases by 1 packet after each *cwnd* update, in phantom queues

[4]Senders that generate data at a rate lower than r are app-limited, and not affected by policing.



(a) Throughput over time (shaded region represents c_l and c_h.)

(b) Queue occupancy over time

(c) Average Throughput

(d) Packet drops

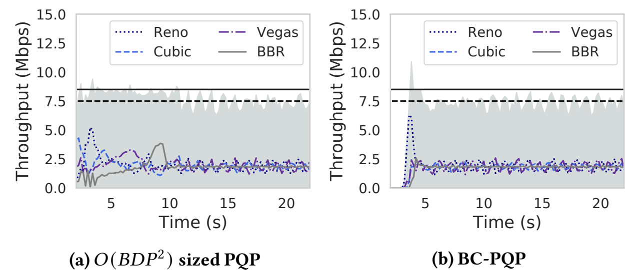
Figure 4: A Reno flow's behavior with phantom queues of different sizes (B).

it increases by $(cwnd-BDP)^+$ packets. A phantom queue therefore must be sized such that it can hold all of these additional packets.

Drawbacks of $O(BDP^2)$ sized queues. If queues are sized by the $O(BDP^2)$ rule, they result in good rate enforcement in a steady state for all congestion control protocols. However, it can cause many other problems. To begin with, it will cause a very large burst in rate during the flow's slow start phase. For example, consider a phantom queue sized for enforcing a rate of 15 Mbps. Suppose the queue is sized assuming the maximum RTT of 100 ms at 1.4MBs, using the $O(BDP^2)$ rule. If a flow with 10 ms RTT passes through this phantom queue, it can burst up to a rate of 143 Mbps over a 100 ms period during its slow start phase, assuming a starting *cwnd* of 10 MSS packets (default in Linux). This also results in a high drop rate – the slow start phase ends with such a high *cwnd* value, that it takes multiple rounds of packet losses (and *cwnd* halving) before the *cwnd* comes down to an average value comparable to *BDP* for correct rate enforcement.

Empirical results. Figure 4 shows the impact of how we size the phantom queue buffer (B) on a Reno flow. We have a Reno flow with *RTT* of 100 ms and we want to enforce rate r of 10 Mbps. For such a flow, B needs to be at least 1000 KB. When B is set to a smaller size of 130 KB, queue occupancy often ends up going to 0, as shown in Figure 4b. This results in the Reno *cwnd* not being able to reach the required peak, thus causing the enforced rate to be lower than r (Figure 4c). When B is large enough (1000KB or 4000KB), we have correct rate enforcement in the steady state, but we have a very large bursts (Figure 4a) and higher drop rates (Figure 4d). Also, as long as the queue remains occupied, its size does not matter in the steady state, e.g. a 4000 KB sized phantom queue does as good a rate enforcement as a 1000 KB one.

The issue of sizing gets worse when we have multiple queues instead of one, where each queue must be sized by the $O(BDP^2)$ rule to ensure correct rate enforcement. The burst caused by this would be much larger and it can further lead to poor policy enforcement if we have a secondary bottleneck after the phantom queue. Figure 5a shows a scenario where we use phantom queues to enforce fair sharing of 7.5 Mbps between 4 flows. We have a secondary bottleneck of 8.5 Mbps after phantom queues[5]. Since phantom queues allow such a large burst to go through, the packets are really



(a) $O(BDP^2)$ sized PQP

(b) BC-PQP

Figure 5: $r=7.5$ Mbps shared across 4 flows with different CC protocols with a secondary bottleneck of 8.5 Mbps afterward.

bottlenecked at the secondary bottleneck which results in poor policy enforcement i.e. fairness in this case.

We need the phantom queue buffers to be large enough for correct rate enforcement in the steady state, but we would still like to avoid the large transient burst. How do we achieve such a behavior? We address this in the next section.

4 Burst Controlled PQP

We saw in the previous section that once a queue becomes full, irrespective of how big it is (albeit it is larger than Reno's requirements), it does correct average rate enforcement. In other words, there is no upper limit on how the queue should be sized for it to do correct rate enforcement in the steady state of a flow. Thus, instead of asking a more complicated question of how to dynamically size the queues, the answer to which depends on various factors like flow's congestion control protocol, RTT, enforced rate r, and demands of other flows, we ask how we can put flows in the steady state without letting them burst.

We have seen in the previous section that to enforce a rate of r, we need to allow some rate variation e.g. between $\frac{2}{3}r$ and $\frac{4}{3}r$ for Reno. However, any burst larger than this is undesirable. We now develop an active phantom queue management scheme that allows us to minimize this burst while still doing correct rate and policy enforcement.

Our idea is based on the observation that once an appropriately sized phantom queue becomes full, a saturating flow (with demand greater than the desired rate r) tries to keep it full in the steady state (e.g. the AIMD state for TCP Reno), and that results correct average rate enforcement. However, during the starting (slow-start) phase,

[5]This can occur when service providers rate-limit the flows before they hit the RAN which may have bandwidth comparable to the enforced rate.

the flow can burst up to a very large rate while it is filling up an empty phantom queue, exiting the slow-start phase only when the queue is full. Our key insight is that *we do not need to wait until the queue becomes full* to exit the starting phase. Instead, we can 'magically' fill the queue when the flow's sending rate exceeds a certain upper threshold (e.g. $\frac{4}{3}r$, which is the upper bound on Reno's rate in a steady state). Similarly, we can drain these 'magic packets' once the flow is finishing up i.e. its sending rate falls below a lower threshold (e.g. $\frac{2}{3}r$, which is the lower bound on Reno's rate in steady state).

Our algorithm achieves this in the following way. We maintain the following additional parameters to configure a PQP system with N queues: (i) an upper threshold multiplier θ^{+}, (ii) a lower threshold multiplier θ^{-}, and (iii) a time period length T.

On enqueue of any packet into phantom queue Q_i, we estimate the dequeue rate r_i^* for this phantom queue and calculate the expected number of bytes that may be dequeued from Q_i over time period T as $X_i = r_i^* T$. r_i^* can be calculated simply based on what queues are active and the rate sharing policy [6]. For example, in the case of fairness, r_i^* is simply r divided by the number of active queues. For prioritization, $r_i^* = r$ if Q_i is the highest priority queue that is active, and 0 otherwise.

Based on this, we compute the upper and lower thresholds on how many bytes each phantom queue is allowed to dequeue before we fill it up with magic packets. Specifically, if the number of packets accepted by a phantom queue Q_i over the current time window of length T is greater than $X_i^+ = \theta^+ X_i$, we fill up the queue with magic packet by magically incrementing its byte counter by $M_i = B - L(Q_i, t)$ (where t is the current time). We keep track of magic packets added for each queue and when the accepted bytes over time T is less than $X_i^- = \theta^- X_i$, we remove all M_i 'magic packets' from this phantom queue [7]. We refer to a PQP system that adopts such an algorithm as burst-controlled PQP (BC-PQP).

With this, any phantom queue Q_i bursts at most X_i^+ bytes, where X_i^+ is proportional to BDP if T is set to a value comparable to RTT. Across all flows in an aggregate, burst is at most $N\theta^+ X$ for any arbitrary rate-sharing policy, where N is the number of phantom queues, and $X = \sum_{i=1}^{N} X_i$. However, it is much smaller on average for policies like fair sharing and prioritization. In the worst case for fairness, we may have all n flows become active over time period T and burst to the maximum possible value of X_i^+. For the first flow, this is $\theta^+ X$, for the second $\theta^+ X/2$, then $\theta^+ X/3$, and so on. This is a harmonic series, which sums to $\theta^+ X(\ln n + 0.5772)$ [11]. For 64 queues, this number is approximately $4.72 \times \theta^+ X$. Similarly, for prioritization, a queue Q_i has $X_i = 0$ if any of the other higher-priority queues are active. As analyzed in §3.4, the cost of this relatively smaller burst is further amortized, the longer the queues remain occupied.

We configure T according to the tail RTT e.g. 100 ms (to get a reasonable estimate of packet enqueue and dequeue rates). We further configure θ^- and θ^+ to be small multipliers (0.5 and 1.5 respectively based on requirements for New Reno). These configurations ensure that we do correct rate enforcement while avoiding unnecessary bursts. Figure 5b shows how flows see a very small and controlled burst with BC-PQP, which results in fair sharing of 7.5 Mbps across 4 different flows in the scenario where we have a secondary bottleneck of 8.5Mbps placed after the policer system.

We end this section by discussing some design insights below:

1. How to set the phantom queue buffer size B_i in a BC-PQP? While the size of the phantom queue does not matter for normal behaving flow as we fill it up as soon as the upper-threshold on enqueue rate is reached, we suggest max-sizing phantom queues to about 2× the required $O(BDP^2)$ limit, so as to avoid a malicious flow, that constantly sends at a rate greater than r but smaller than the upper threshold, from sending at a higher rate than r indefinitely.

2. Why do we remove magic packets once the flow becomes inactive? BC-PQP removes the magic phantom packets (i.e. appropriately decrements the byte counter) when the dequeue rate recedes the lower threshold, e.g. when the flow becomes inactive. Doing so allows BC-PQP to immediately allocate the spare rate to other flows (as opposed to continually dequeuing the magic packets). In normal PQP with very large queues, when a flow becomes inactive, it takes a long time before its phantom queue is drained, which results in transient under-enforcement of rate even though other flows are active.

3. How does BC-PQP enable dynamic auto-configuration of phantom queues? BC-PQP estimates the expected dequeue rate, $r_i{}^*$ of each queue Q_i independently – adapting this rate as per the rate sharing policies, as the set of active queues changes. This allows BC-PQP to automatically adapt the phantom queue configurations (upper and lower thresholds for adding and removing magic packets) with changing traffic patterns, instead of relying on a static configuration that is hard to tune. This also allows BC-PQP to easily generalize to arbitrary rate-sharing policies.

5 Implementation

We develop a middlebox for transparently enforcing traffic rates using a kernel-bypass stack based on Intel's DPDK. We implement BC-PQP and other relevant baselines (including shaper, policer, and fairpolicer) in this middlebox for our evaluations in §6. We run microbenchmarks on Azure public cloud and use three standard F8sv2 Virtual Machines running Ubuntu 22.04, dedicated for the sender, receiver, and the middlebox respectively. For traffic, we create flows of different sizes using TCP sockets and configure the congestion control algorithm at a per-flow granularity. We use Linux kernel implementation of all congestion control protocols. We use Linux netem to artificially inflate latency and approximate realistic WAN RTTs. The sender traffic is routed through the middlebox responsible for enforcing the network rate before it reaches the receiver.

We also test phantom queues with real applications. For this, we implement phantom queues and other baselines in Mahimahi. Inside the Mahimahi shell, we run a browser to run different applications i.e. video streaming services like YouTube and Netflix, and web browsing.

6 Evaluation

We compare BC-PQP with several baselines (traffic shapers, traffic policers, and FairPolicer [42]), evaluating the following:

- BC-PQP's ability to do correct rate enforcement (§6.2).
- BC-PQP's system efficiency (§6.3).
- BC-PQP's ability to enforce different rate sharing policies (§6.4).

[6]Maintaining a list of active queues also has an efficiency benefit, since during phantom dequeue, we do not have to iterate over all queues but only the active ones.

[7]It is possible that we may not have enough packets in queue size to reclaim all magic packets, however, this is a transient behavior for a backlogged flow and does not affect rate/policy enforcement much.

• Improvement in application quality-of-experience (QoE) with BC-PQP for real-world applications – video streaming and web browsing (§6.5).

6.1 Experiment Setup

We do rate enforcement for multiple flow aggregates (representing a given user), each consisting of multiple flows. Our setup consists of three machines: a sender machine, a rate enforcer machine, and a receiver machine. The traffic from the sender machine is routed through the rate enforcer machine. Our goal is to enforce the specified rate of r for each aggregate. Unless otherwise specified, we use the rate-sharing policy of round-robin (per-flow fairness) within each aggregate.

Rate per aggregate. We fix the enforced rate r for each flow aggregate in each experiment, varying this value across experiment runs ranging it from 1.5Mbps to 200 Mbps. We specifically experiment with per-aggregate rates of 1.5Mbps, 7.5Mbps, 25Mbps, 50Mbps, 100Mbps, and 200Mbps.

Number of aggregates. For each experiment, we have 100 flow aggregates, except for experiments where we set the per-aggregate rate to 100Mbps or 200Mbps (in these experiments, we reduce the number of flow aggregates to 60 and 30 respectively to avoid bottlenecks in netem at the sender).

Traffic mix within each aggregate. The flow sizes in each aggregate range from a few 10s of KBs to 100s of MBs. For congestion control protocols, we pick from New Reno, Cubic, BBR, and Vegas (covering popularly used protocols ranging from purely loss-based to delay-based). We use Linux kernel implementation of all these protocols. We also use netem to inject different delays to different flows ranging from 2 ms to 50 ms. In each experiment, we have a mix of aggregates, in half of the aggregates all flows use the same congestion control protocol (∈ {Cubic, Reno, BBR, Vegas}) and have the same RTT (∈ {2 ms, 5 ms, 10 ms, 25 ms, 50 ms}), while in the other half, we have flows with different congestion control protocols as well as different RTTs. Moreover, in each of these groups, some aggregates only have backlogged flows (large flows with flow completion time in minutes), whereas others only have short on-and-off flows (10s of KBs), whereas a third subgroup has both backlogged and short on-and-off flows.

Computing relevant metrics. The rate enforcer machine forwards traffic to the receiver machine, where per-flow throughput is measured over 250ms windows. We use these per-flow rates over these 250ms windows to compute different flow-specific metrics when evaluating policy enforcement within each aggregate. We further sum the throughput of each flow within each aggregate over each 250ms window and normalize this aggregate throughput by enforced rate r to compute the aggregate enforced rate over each 250ms window.

Baselines. We compare BC-PQP with the following baselines:

• *Shaper*: We implement our shaper inspired by an industry-standard algorithm. Similar to Carousel [39], we use timerwheels [51] to schedule dequeues from the shapers. Carousel puts packets to dequeue directly on the timerwheel and therefore is unable to support rate sharing policies within a shaper. We put shapers on the timer wheel and dequeue packets from the shapers based on the rate-sharing policy. Each shaper has multiple queues each sized according to maximum BDP.

• *FairPolicer (FP)*: As mentioned in §2, FP [42] provides a point solution for realizing per-flow fairness with statically configured TBF-based policers. We use the best possible configuration for FP. We size the bucket B for FP to be the maximum of any flow's requirement i.e. we pick the maximum RTT and compute B needed for correct rate enforcement for New Reno and Cubic, and pick the maximum value. For small values of RTT and rate, Cubic requires a larger bucket size, whereas in other cases New Reno requires a larger bucket size (using $O(BDP^2)$).

• *Policer*: A token bucket traffic policer sized according to maximum BDP.

• *Policer+*: A token bucket traffic policer sized similar to FP.

For BC-PQP, we do not need to set an explicit size of the bucket thus we pick a very high value of at least $10 \times O(BDP^2)$. For other parameters of BC-PQP, we set θ^+, θ^-, and T to be 1.5, 0.5, and 100ms respectively.

6.2 Rate Enforcement

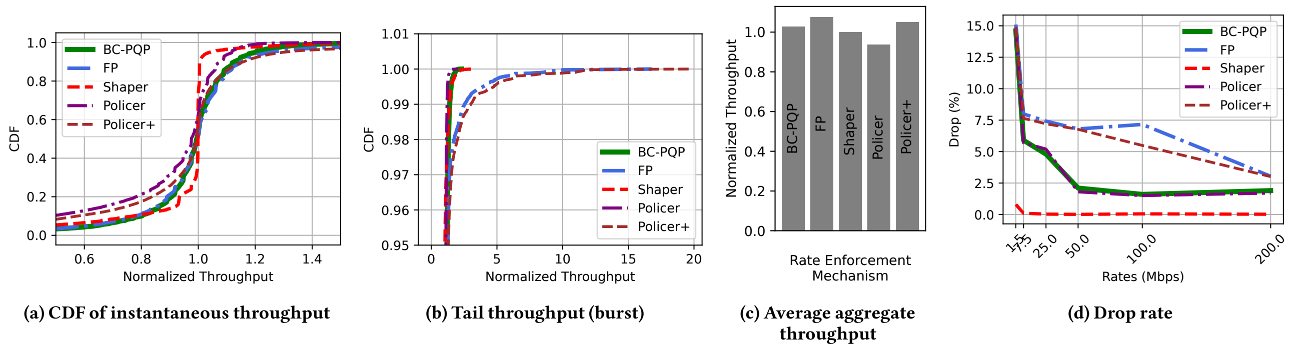
We begin with presenting results on how BC-PQP's aggregate rate enforcement behavior compares with the baselines. Figure 6 summarizes the rate enforcement performance of BC-PQP and different baselines. We can draw the following insights from these results:

(1) The distribution of normalized throughput of each aggregate for all the rates is shown in Figure 6a. It can be seen that the shaper does accurate rate enforcement over short time scales. The instantaneous rate for other baselines also stays within small bounds (roughly $0.8r$ to $1.2r$) for most parts. However, Policer+ and FP cause a much larger burst and have a long tail for aggregate throughput (as shown in Figure 6b). BC-PQP, in contrast, retains a small deviation in aggregate throughput even at the tails, performing close to a shaper. As shown previously in Figure 5 in §4, BC-PQP's small burst ensures that the rate and policy enforcement are correct even in the presence of a secondary bottleneck with bandwidth comparable to r.

(2) Policer (with BDP-sized buffer) has a small burst, but at the cost of the average throughput being lower than the desired rate r. This can be observed in Figure 6a, where the line for Policer is slightly shifted left. Figure 6c further reports the average of all non-zero aggregate throughput measurements. The average throughput is higher for FP and Policer+ because bursty throughput points skew the average up.

(3) Due to the lack of buffering, all policing-based schemes induce a higher number of packet drops than Shaper. The number of packet drops reduces as the BDP (either of rate or RTT) increases, this is especially apparent from trends for BDP-sized policer and BC-PQP. However, as discussed previously a large-sized policer (and also FP) results in a higher number of packet drops especially when flows end up with an over-estimated congestion windows at the end of the slow start phase. BC-PQP on the other hand avoids such drops by avoiding large bursts and has drop rates comparable with BDP-sized policer. Shaper's small drop rates come at the cost of inducing higher queuing delays. We find that the high queuing delay of Shaper can at times hurt application performance more than the relatively higher packet drop rates of BC-PQP (§6.5) – so the trade-off between them is unclear.

Overall, BC-PQP results in correct aggregate rate enforcement with 10× smaller burst compared with FP or a correctly sized policer.



(a) CDF of instantaneous throughput **(b) Tail throughput (burst)** **(c) Average aggregate throughput** **(d) Drop rate**

Figure 6: Aggregate rate enforced by BC-PQP and other baselines, 6a and 6b show distribution of aggregate throughput measured over 250 ms windows normalized by enforced rate, 6c shows normalized average aggregate throughput and 6d shows drop rate at different enforced rates

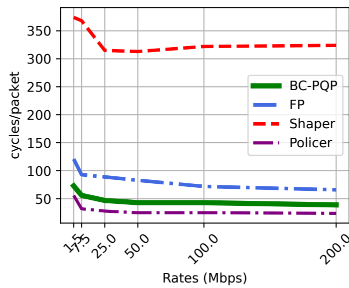


Figure 7: CPU cycles spent per packet

6.3 System Efficiency

We use CPU cycles spent per packet as a proxy to quantify the efficiency and scalability of different rate enforcement schemes. This indirectly captures other overheads e.g. storing and retrieving of packets from the memory. Moreover, spending more CPU cycles per packet results in the system being able to handle fewer packets per second. Figure 7 reports the average number of CPU cycles spent on each packet with different schemes. BC-PQP uses 5-7 × fewer CPU cycles per packet and is marginally costlier than a simple policer. A shaper spends several CPU cycles during its dequeue routine where it needs to gather packets from different queues of different shapers before sending them to the NIC to dequeue. On the other hand, all other schemes do not need to store packets and make the decision about the fate of the packet on its enqueue thus avoiding spending CPU cycles on costly memory trips. Policer and BC-PQP furthermore are more efficient than FP because we can batch phantom dequeues or token replenishing, and only call phantom dequeue or token generator when the phantom queue is full or the token bucket is empty. On the other hand, FP makes decisions to drop incoming packets based on a dynamic threshold which is a function of up-to-date per-flow residual bucket space. This requires generating and allocating tokens on each enqueue of a packet.

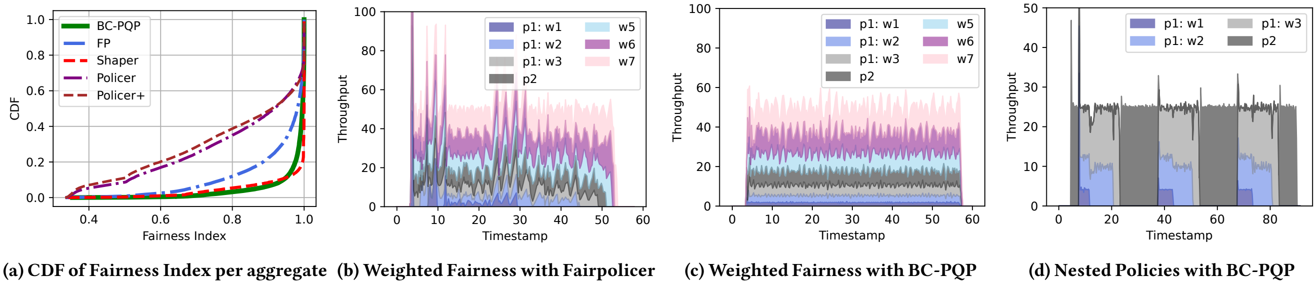
6.4 Policy Enforcement

In this section, we look at how well BC-PQP enforces rate-sharing policies within an aggregate.

6.4.1 Per-flow Fairness. For the experiment from §6.2, we measure the per-flow throughput over 250 ms windows within each aggregate and estimate fairness using Jain's Fairness Index [27]. CDF of this fairness index is reported in Figure 8a. We can see that the shaper achieves close to perfect fair sharing of the enforced rate as expected, but Policer and Policer+ are unable to do so. BC-PQP achieves fairness comparable to the shaper. While FP does better than policers, it falls a bit short for the following reason – flows with different RTTs require differently sized buckets for correct rate enforcement. FP does not differentiate between them and instead splits a single (statically configured) bucket equally among different flows. This creates large bursts in flows with small RTTs (that require smaller buckets), and lower than desired average rates for flows with large RTTs (that require larger buckets), thereby causing unfairness. BC-PQP, in contrast, dynamically adapts the phantom queue configuration by adding and removing magic packets based on flow arrival rates as described in §4 – it naturally fills up the queue with magic packets sooner for flows with smaller RTTs that ramp up faster, when compared to flows with longer RTTs.

6.4.2 Weighted Fairness. We next run a microbenchmark experiment to show how BC-PQP can do accurate weighted sharing within an aggregate. We experiment with a single aggregate, with an enforced rate of 50 Mbps shared between 7 flows, each with weights from 1 to 7. All flows start at the same time and are sized proportional to their weights so that they should complete at the same time if the rate is shared between them as per their weights. We also adapt the token allocation logic in FairPolicer to make it do similar weighted sharing, without changing the bucket sizing. It is non-trivial to adapt FP's bucket sizing logic for arbitrary rate-sharing policies. Figures 8b and 8c show the time series of per-flow and aggregate throughput for FP and BC-PQP. BC-PQP enforces weighted sharing correctly resulting in all flows completing at the same time. FP fails to do so, this is because while FP tries to allocate tokens in a weighted fair manner, the way it sizes each flow's bucket works only for fair sharing. It sets each flow's bucket capacity to be equal to the number of tokens remaining in the main token bucket. Thus, each flow with different weights gets approximately same-sized token buckets even though a flow with a higher weight should get a larger one. It is not trivial to extend FP's bucket sizing algorithm to support arbitrary rate-sharing policies.

6.4.3 Prioritization and Nested Policies. In this section, we show the feasibility of enforcing prioritization and nested policies with



(a) CDF of Fairness Index per aggregate (b) Weighted Fairness with Fairpolicer (c) Weighted Fairness with BC-PQP (d) Nested Policies with BC-PQP

Figure 8: Policy enforcement with BC-PQP and other baselines: 8a shows per-flow fairness between flows within an aggregate, 8b and 8c show weighted fairness achieved by FairPolicer and BC-PQP respectively and 8d shows a nested policy with prioritization and weighted fairness using BC-PQP.

BC-PQP, using a microbenchmark. We experiment with a single aggregate with 4 flows divided into two priority groups: (i) p1 is the higher priority group with 3 on-and-off flows, and within p1, the rate is shared in a weighted fair manner between the 3 flows, and (ii) p2 is the lower priority group with a single backlogged flow. Figure 8d shows how BC-PQP allocates all bandwidth to p1 flows in a weighted fair manner whenever they're active and only allocates bandwidth to p2 flow when no p1 flow is active.

6.5 Real World Applications

We show the effectiveness of BC-PQP in enforcing aggregate rates and rate-sharing policies for different real-world applications. We look at scenarios where an enforced aggregate rate r is shared by different kinds of flows.

6.5.1 Video Streaming. We look at a scenario where an aggregate rate is shared between a video stream and some other traffic. Cellular service providers and ISPs often use policers or single queue shapers to do such rate enforcement for each user. We simulate this scenario with two different rates i.e. 3 Mbps and 10 Mbps. The aggregate rate is shared between video flow and the rest of the traffic (webpage loads, downloads, etc.). We further try to implement different rate-sharing policies within an aggregate, with 3 Mbps, we do 1:1 fairness and with 10 Mbps, we do 4:1 weighted fairness between video flow and the rest of the traffic respectively.

We repeat this experiment using 3 videos from YouTube and 2 videos from Netflix. YouTube uses BBR [3], while Netflix uses New Reno [47] for its congestion control protocols. For baselines, we use a simple policer and a single queue shaper as well as a shaper with deficit round robin. The first two are the status quo mechanisms to do rate enforcement.

Figure 9a shows the results with a 3Mbps aggregate rate that is shared in a 1:1 fairness ratio between the video flow and the rest of the traffic. It plots the video quality on the Y-axis and the fairness index on the X-axis. We find that BC-PQP shares the rate fairly between the video stream and the other flows, and also achieved high video quality with both Netflix and YouTube. Single queue shaper and policer, on the other hand, are not able to share the rate fairly. While shaper with DRR ensures fairness between traffic, YouTube videos' quality suffers – this is likely due to an additional queuing delay introduced due to the buffering of packets. [8]

Figure 9b shows the average video quality achieved by the video streams when an aggregate rate of 10Mbps is split to give higher weightage to video flows. Video quality is similar for Netflix irrespective of rate enforcement mechanism and rate sharing policy, this is because Netflix caps the video quality at 720p on web browsers. However, BC-PQP outperforms the other baselines for YouTube flows, resulting in higher video quality compared to policer and single-queue shaper due to better policy enforcement, and even compared to shaper with DRR (presumably due to YouTube's adverse reaction to queuing delay, as mentioned above).

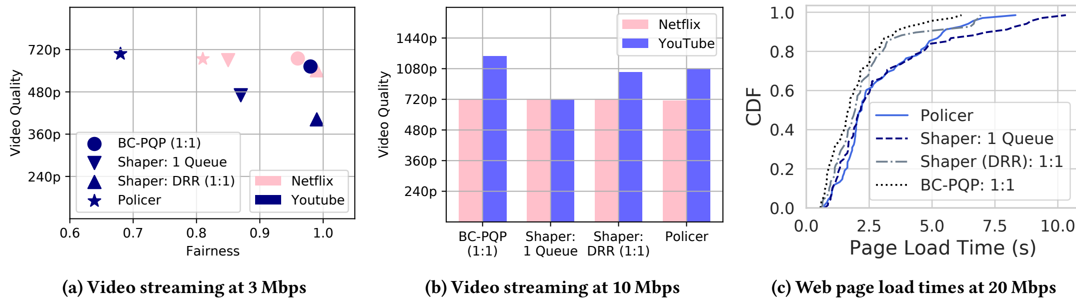
6.5.2 Web Browsing. Similar to the previous setup, we share a 20 Mbps aggregate rate between a download flow and web browsing traffic. We open 75 web pages for each experiment in the presence of a bulk download flow. We use DRR shaper and BC-PQP to enforce weighted sharing of rate between bulk flow and web browsing flows in ratio of 1:1. CDF of web page load times is reported in figure 9c. BC-PQP achieves up to 2× lower page load times compared to status quo baselines of policer and single queue shaper.

7 Related Work

Rate enforcement is a key building block for any kind of network management. Mechanisms to do rate enforcement correctly have been explored in the past [7, 23, 39–42]. These solutions usually include traffic shapers [7, 23, 39, 40], which buffer packets in memory or traffic policers [41, 42, 50]. Previous works have noted the limitations of traffic shapers and policers, namely traffic shapers are expensive to implement [21, 41], whereas traffic policers suffer from poor rate enforcement and high packet losses [21, 28, 50].

Traffic policers are known to be difficult to configure [21, 50]. Guidelines around configuring policer bucket sizes are not homogenous and depend largely on which factor is more important: correct rate enforcement [41, 42] or small burst [6, 21]. [50] presents a dynamically sized bucket that is adapted based on how long it takes for a flow to ramp up after packet losses. This is an iterative process that takes multiple attempts to gauge the correct size needed for a flow, eventually setting the size to $O(BDP^2)$ for a Reno flow. BC-PQP does

[8] The precise reason for this is not clear because of the lack of visibility into YouTube's ABR algorithm.



(a) Video streaming at 3 Mbps (b) Video streaming at 10 Mbps (c) Web page load times at 20 Mbps

Figure 9: Video streaming and web browsing in the presence of background traffic with BC-PQP and baselines.

not need to vary phantom queue sizes – instead, it relies on its unique burst control mechanism to avoid bursts and enforce correct rates.

Enforcing different policies within a traffic aggregate is desirable for the operators as well as users [16, 29, 49]. Past work has looked into various mechanisms to implement policies like per-flow fairness, weighted fair queuing, or prioritization [20, 34, 36, 37, 43, 45]. These works usually depend on buffering packets, sometimes into multiple queues, whereas other times into a single shared buffer to be more space efficient [34, 36, 43]. Using bufferless mechanisms for such policy enforcement has not been explored much. Recent work attempts to make token bucket policers fair when flows with different congestion control protocols pass through it [41, 42]. However, this does not extend to other rate-sharing policies. Moreover, it suffers from burstiness and some level of RTT unfairness [41].

Phantom queues have been proposed under different names for different functionalities. More recently, they have been popularized as an active queue management scheme [8, 31, 32]. However, some of the earliest works in ATM networks used "leaky buckets as a meter" for rate enforcement, which works the same way as a token bucket in principle, albeit it has also been called a pseudo queue [15, 22, 26]. Our key contribution lies in augmenting a policer with *multiple* phantom queues, and showing how it can do policy-rich rate enforcement.

8 Conclusion and Discussion

Even though we, as users, do not like the idea of ISPs rate limiting our traffic, it is prevalent and we cannot escape it – the need for it is inherently coupled with Internet economics. In this paper, we embrace the idea of rate limiting and focus on doing it right. This requires enabling the ISPs to enforce different rate-sharing policies (fairness across flows using different congestion control algorithms, weighted rate sharing across a given user's flows as per their preferences, etc) at scale. Our system BC-PQP enables that by providing the system-level efficiency of a policer (by not buffering any packets) but the network-level properties of a shaper (characterized by its ability to correctly enforce the desired policy and rate). Several dimensions remain open for future research.

Expressive and customizable rate enforcement: While we present a cost-effective solution to implement expressive rate-sharing policies at the ISP for rate-limited traffic aggregates, more work needs to be done towards standardizing how these policies can be customized by end-users or applications (along the lines of [12, 18, 24, 30, 54, 55]).

Trade-off between shapers and policers: A fundamental trade-off between shapers and policers is that of queuing delay and packet drops. With the existing congestion control algorithm, drops incurred by BC-PQP are unavoidable. For example, as per Mathis' TCP model, for AIMD protocols like Reno, the drop rate is inversely related to the square of $rate \times RTT$ [35]. Therefore, in the absence of queuing delay, as for BC-PQP and policer, drop rates are high but with queuing delay, RTT is inflated, and thus drop rates go down. While the applications we tested are not harmed by higher packet drops caused by BC-PQP when compared to the queuing delay induced by a shaper, certain other applications' QoS may be more susceptible to packet drops compared to queuing delay. We leave such exploration with different applications to future work.

Hardware implementation of BC-PQP: The principles we discussed that make software implementation of BC-PQP a lot more performant than shapers, also apply to hardware implementation. A hardware implementation of BC-PQP will require a much smaller amount of SRAM and shorter packet pipelines compared to shapers. We leave such hardware implementation to future work.

This work does not raise any ethical concerns.

BC-PQP's implementation is open-source and can be found at: https://github.com/PhantomQueuePolicer.

9 Acknowledgements

We would like to thank our shepherd, Peter Steenkiste, and the anonymous reviewers for their helpful feedback. This work was supported in parts by USDA NIFA (award number 2021-67021-34418), NSF (award number 2217144), and Microsoft.

References

[1] [n. d.]. AT&T: Learn About Video Management. https://www.att.com/support/article/wireless/KM1169198/. ([n. d.]).
[2] [n. d.]. Linux Hierarchical Token Buckets. http://luxik.cdi.cz/~devik/qos/htb/. ([n. d.]).
[3] [n. d.]. TCP BBR congestion control comes to GCP – your internet just got faster | google cloud blog. ([n. d.]).
[4] [n. d.]. VMWare SD-WAN. https://docs.vmware.com/en/VMware-SD-WAN/3.3/VMware-SD-WAN-by-VeloCloud-Administration-Guide/GUID-EE8C35B8-FA4E-4C59-9AC2-4FD14509F60C.html. ([n. d.]).
[5] [n. d.]. What Is SD-WAN? https://www.cisco.com/c/en/us/solutions/enterprise-networks/sd-wan/what-is-sd-wan.html. ([n. d.]).
[6] 2023. (Sep 2023). https://www.cisco.com/c/en/us/support/docs/quality-of-service-qos/qos-policing/19645-policevsshape.html#traffic
[7] Saamer Akhshabi, Lakshmi Anantakrishnan, Constantine Dovrolis, and Ali C Begen. 2013. Server-based traffic shaping for stabilizing oscillating adaptive streaming players. In *23rd ACM Workshop on Network and Operating Systems Support for Digital Audio and Video.*

[8] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. 2012. Less is more: Trading a little bandwidth for {Ultra-Low} latency in the data center. In *9th USENIX Symposium on Networked Systems Design and Implementation*.

[9] Guido Appenzeller, Isaac Keslassy, and Nick McKeown. 2004. Sizing router buffers. *ACM SIGCOMM Computer Communication Review* (2004).

[10] Eneko Atxutegi, Fidel Liberal, Habtegebreil Kassaye Haile, Karl-Johan Grinnemo, Anna Brunstrom, and Ake Arvidsson. 2018. On the use of TCP BBR in cellular networks. *IEEE Communications Magazine* (2018).

[11] Ralph P Boas Jr and John W Wrench Jr. 1971. Partial sums of the harmonic series. *The American Mathematical Monthly* (1971).

[12] Ilker Nadi Bozkurt, Yilun Zhou, Theophilus Benson, Bilal Anwer, Dave Levin, Nick Feamster, Aditya Akella, Balakrishnan Chandrasekaran, Cheng Huang, Bruce Maggs, et al. 2015. Dynamic prioritization of traffic in home networks. In *CoNEXT Student Workshop*.

[13] Lawrence S. Brakmo, Sean W. O'Malley, and Larry L. Peterson. 1994. TCP Vegas: New Techniques for Congestion Detection and Avoidance. In *ACM Conference on Communications Architectures, Protocols and Applications*.

[14] Lloyd Brown, Yash Kothari, Akshay Narayan, Arvind Krishnamurthy, Aurojit Panda, Justine Sherry, and Scott Shenker. 2023. How I Learned To Stop Worrying About CCA Contention. In *31st Workshop on Hot Topics in Networks*.

[15] Milena Butto, Elisa Cavallero, and Alberto Tonietti. 1991. Effectiveness of the'leaky bucket'policing mechanism in ATM networks. *IEEE Journal on selected areas in communications* (1991).

[16] Frank Cangialosi, Akshay Narayan, Prateesh Goyal, Radhika Mittal, Mohammad Alizadeh, and Hari Balakrishnan. 2021. Site-to-site internet traffic control. In *16th European Conference on Computer Systems*.

[17] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2016. BBR: Congestion-Based Congestion Control. *ACM Queue* (2016).

[18] Saoussen Chaabnia and Aref Meddeb. 2018. Slicing aware QoS/QoE in software defined smart home network. In *IEEE/IFIP Network Operations and Management Symposium*.

[19] Saahil Claypool, Jae Chung, and Mark Claypool. 2021. Measurements comparing TCP cubic and TCP BBR over a satellite network. In *IEEE 18th Annual Consumer Communications & Networking Conference (CCNC)*.

[20] Alan Demers, Srinivasan Keshav, and Scott Shenker. 1989. Analysis and simulation of a fair queueing algorithm. *ACM SIGCOMM Computer Communication Review* (1989).

[21] Tobias Flach, Pavlos Papageorge, Andreas Terzis, Luis Pedrosa, Yuchung Cheng, Tayeb Karim, Ethan Katz-Bassett, and Ramesh Govindan. 2016. An Internet-Wide Analysis of Traffic Policing. In *ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*.

[22] G Gallassi, G Rigolio, and Luigi Fratta. 1989. ATM: Bandwidth assignment and bandwidth enforcement policies. In *IEEE Global Telecommunications Conference and Exhibition'Communications Technology for the 1990s and Beyond'*.

[23] Leonidas Georgiadis, Roch Guérin, Vinod Peris, and Kumar N Sivarajan. 1996. Efficient network QoS provisioning based on per node traffic shaping. *IEEE/ACM transactions on networking* (1996).

[24] Hassan Habibi Gharakheili, Jacob Bass, Luke Exton, and Vijay Sivaraman. 2014. Personalizing the home network experience using cloud-based SDN. In *Proceeding of IEEE International symposium on a world of wireless, mobile and multimedia networks 2014*.

[25] Sangtae Ha, Injong Rhee, and Lisong Xu. 2008. CUBIC: a New TCP-friendly High-Speed TCP Variant. *ACM SIGOPS Operating System Review* (2008).

[26] Joseph SM Ho, Hüseyin Uzunalioglu, and Ian F Akyildiz. 1995. Cooperating leaky bucket for average rate enforcement of VBR video traffic in ATM networks. In *Proceedings of INFOCOM'95*.

[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* (1984).

[28] Arash Molavi Kakhki, Fangfan Li, David Choffnes, Ethan Katz-Bassett, and Alan Mislove. 2016. Bingeon under the microscope: Understanding t-mobiles zero-rating implementation. In *Proceedings of the 2016 workshop on QoE-based Analysis and Management of Data Communication Networks*.

[29] Alok Kumar, Sushant Jain, Uday Naik, Anand Raghuraman, Nikhil Kasinadhuni, Enrique Cauich Zermeno, C Stephen Gunn, Jing Ai, Björn Carlin, Mihai Amarandei-Stavila, et al. 2015. BwE: Flexible, hierarchical bandwidth allocation for WAN distributed computing. In *ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*.

[30] Himal Kumar, Hassan Habibi Gharakheili, and Vijay Sivaraman. 2013. User control of quality of experience in home networks using SDN. In *2013 IEEE International conference on advanced networks and telecommunications systems (ANTS)*.

[31] Srisankar Kunniyur and Rayadurgam Srikant. 2001. Analysis and design of an adaptive virtual queue (AVQ) algorithm for active queue management. *ACM SIGCOMM Computer Communication Review* (2001).

[32] Srisankar S Kunniyur and Rayadurgam Srikant. 2004. An adaptive virtual queue (AVQ) algorithm for active queue management. *IEEE/ACM Transactions on networking* (2004).

[33] Fangfan Li, Arian Akhavan Niaki, David Choffnes, Phillipa Gill, and Alan Mislove. 2019. A large-scale analysis of deployed traffic differentiation practices. In *Proceedings of the ACM Special Interest Group on Data Communication*. 130–144.

[34] Robert MacDavid, Xiaoqi Chen, and Jennifer Rexford. 2023. Scalable real-time bandwidth fairness in switches. *IEEE/ACM Transactions on Networking* (2023).

[35] Matthew Mathis, Jeffrey Semke, Jamshid Mahdavi, and Teunis Ott. 1997. The macroscopic behavior of the TCP congestion avoidance algorithm. *ACM SIGCOMM Computer Communication Review* (1997).

[36] Rong Pan, Lee Breslau, Balaji Prabhakar, and Scott Shenker. 2003. Approximate fairness through differential dropping. *ACM SIGCOMM Computer Communication Review* (2003).

[37] 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* (1993).

[38] S. Blake and D. Black and M. Carlson and E. Davies and Z. Wang and W. Weiss. 1998. An Architecture for Differentiated Services. RFC 2475. (1998).

[39] Ahmed Saeed, Nandita Dukkipati, Vytautas Valancius, Vinh The Lam, Carlo Contavalli, and Amin Vahdat. 2017. Carousel: Scalable traffic shaping at end hosts. In *ACM Special Interest Group on Data Communication (SIGCOMM)*.

[40] Ahmed Saeed, Yimeng Zhao, Nandita Dukkipati, Ellen Zegura, Mostafa Ammar, Khaled Harras, and Amin Vahdat. 2019. Eiffel: Efficient and flexible software packet scheduling. In *16th USENIX Symposium on Networked Systems Design and Implementation*.

[41] Danfeng Shan, Linbing Jiang, Peng Zhang, Wanchun Jiang, Hao Li, Yazhe Tang, and Fengyuan Ren. 2023. Enforcing Fairness in the Traffic Policer Among Heterogeneous Congestion Control Algorithms. *IEEE/ACM Transactions on Networking* (2023).

[42] Danfeng Shan, Peng Zhang, Wanchun Jiang, Hao Li, and Fengyuan Ren. 2021. Towards the Fairness of Traffic Policer. In *40th IEEE Conference on Computer Communications, INFOCOM*.

[43] 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*.

[44] M. Shreedhar and George Varghese. 1995. Efficient Fair Queueing Using Deficit Round Robin. *ACM SIGCOMM Computer Communication Review* (1995).

[45] Madhavapeddi Shreedhar and George Varghese. 1995. Efficient fair queueing using deficit round robin. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*.

[46] Madhavapeddi Shreedhar and George Varghese. 1996. Efficient fair queuing using deficit round-robin. *IEEE/ACM Transactions on networking* (1996).

[47] Bruce Spang, Shravya Kunamalla, Renata Teixeira, Te-Yuan Huang, Grenville Armitage, Ramesh Johari, and Nick McKeown. 2023. Sammy: smoothing video traffic to be a friendly internet neighbor. In *ACM SIGCOMM*.

[48] T-Mobile. 2024. Unlimited video streaming with Binge On™. (2024). https://www.t-mobile.com/tv-streaming/binge-on

[49] Ammar Tahir and Radhika Mittal. 2023. Enabling Users to Control their Internet. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 555–573.

[50] Ronald van Haalen and Richa Malhotra. 2007. Improving TCP performance with bufferless token bucket policing: A TCP friendly policer. In *2007 15th IEEE Workshop on Local & Metropolitan Area Networks*.

[51] George Varghese and Anthony Lauck. 1987. Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility. In *11th ACM Symposium on Operating System Principles, SOSP*.

[52] Verizon. 2024. Verizon customers can save more in 2024. (2024). https://www.verizon.com/about/news/verizon-customers-can-save-more-2024

[53] Gary R Wright and W Richard Stevens. 1995. *TCP/IP Illustrated, Volume 2 (paperback): The Implementation*. Addison-Wesley Professional.

[54] Yiannis Yiakoumis, Sachin Katti, Te-Yuan Huang, Nick McKeown, Kok-Kiong Yap, and Ramesh Johari. 2012. Putting home users in charge of their network. In *ACM Conference on Ubiquitous Computing*.

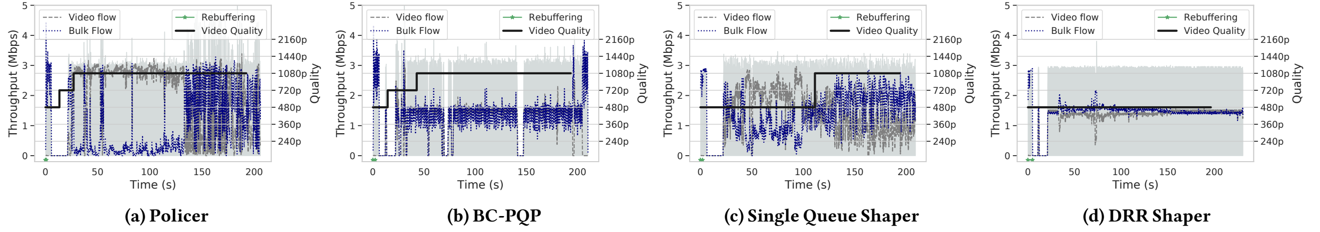
[55] Yiannis Yiakoumis, Sachin Katti, and Nick McKeown. 2016. Neutral Net Neutrality. In *Proceedings of the 2016 ACM SIGCOMM Conference*.

APPENDIX

Appendices are supporting material that has not been peer-reviewed.

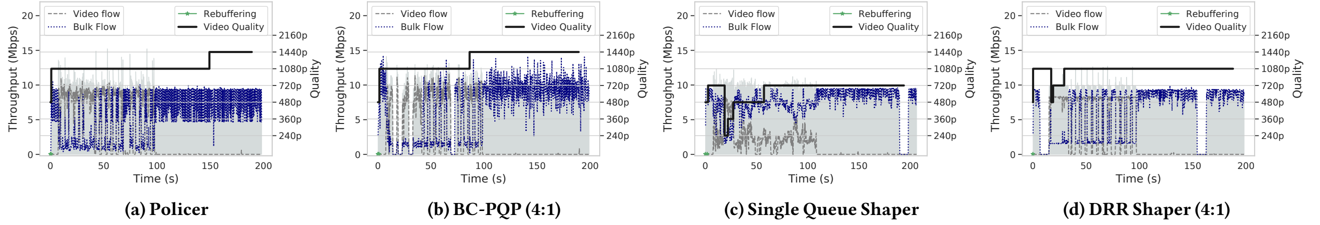
Appendix A Sizing the phantom queue for Reno

We analyze how to size the buffer B of a phantom queue Q, being served at rate r (in packets per second), for a backlogged Reno flow. Reno is a congestion window-driven additive increase, multiplicative decrease protocol that is sensitive to packet losses. The sender maintains a congestion window, $cwnd$, to cap the



(a) Policer (b) BC-PQP (c) Single Queue Shaper (d) DRR Shaper

Figure 10: A youtube video stream sharing 3 Mbps link with some other traffic with different schemes.



(a) Policer (b) BC-PQP (4:1) (c) Single Queue Shaper (d) DRR Shaper (4:1)

Figure 11: A youtube video stream sharing 10 Mbps link with some other traffic with different schemes.

inflight packets. On each successful packet delivery, the congestion window is updated as $cwnd = cwnd + 1/cwnd$. Whereas, on packet loss, the congestion window is halved: $cwnd = cwnd/2$. Since the phantom queue does not cause any queuing delay, all $cwnd$ packets' acknowledgments are received in one round trip time. Consider that a flow has a round trip time of RTT. In the steady state of Reno, when the phantom queue becomes full, a packet loss causes Reno to halve its congestion window, let's call this congestion window c_l. Thus, Reno sends c_l packets in the next round trip. After successfully delivery of all packets, $c_l + 1$ packets are sent in the next round, and so on. As the congestion window increases additively, the phantom queue is drained at rate r packets per second. Suppose it takes n RTTs for the queue to become full again. At this point, we reach the highest congestion window – let's call it c_h. Thus over time duration $nRTT$, if the queue does not go to zero, we phantom dequeue $nRTT \times r = n \times BDP$ packets from the phantom queue, and $\sum_{i=1}^{n}(c_l + i)$ more packets are accepted over this duration. Thus, we have:

$$n \times BDP = \sum_{i=1}^{n}(c_l + i)$$

We have following relationship between c_l and c_h: $c_l = c_h/2$ and $c_h = c_l + n$, through which we have $c_l = n$. Plugging this in the above equation gives us values of $n = c_l \approx \frac{2}{3}BDP$ and $c_h \approx \frac{4}{3}BDP$. This means, that for correct rate enforcement with Reno, we need the instantaneous rate (over RTT period) to vary between $\frac{2r}{3}$ and $\frac{4r}{3}$. When B is not large enough, we are unable to phantom dequeue $n \times BDP$ packets over the given duration, which results in the average enforced rate being less than r. We need B to be at least as large as the area of the shaded in Figure 12 to hold the additional packets that are sent beyond rate r, which comes out to be $\frac{BDP^2}{18} \times MSS$ bytes.

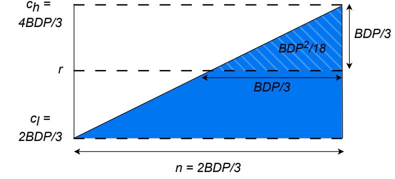


Figure 12: Reno's $cwnd$ progression over n RTTs, we need phantom queue to be at least the size of shaded region.

Appendix B YouTube's video stream analysis

The time series for one video with different schemes is shown in Figure 10 for 3 Mbps and Figure 11 for 10 Mbps. Since YouTube uses BBR, with a policer, the video flow hogs most of the bandwidth thus achieving high video quality. On the other hand, with a shaper, the video flow is not as aggressive. There can be two explanations for this, firstly since competing traffic carries buffer-filling flows, BBR yields bandwidth to bring down queuing delay. The second plausible reason could be that YouTube's ABR algorithm is also sensitive to queuing delay. The result with DRR-shaper gives more weight to this conclusion. Even though YouTube flow has a separate queue, its video quality still suffers.