Efficient Policy-Rich Rate Enforcement with Phantom Queues

Ammar Tahir[¶,†], Prateesh Goyal[¶], Ilias Marinos[¶], Mike Evans[¶], Radhika Mittal[†]

[¶]Microsoft, [†]UIUC

ABSTRACT

Rate enforcement is routinely employed in modern networks (e.g. ISPs rate-limiting users traffic to the subscribed rates). 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). And all of this must be done at scale to efficiently support the vast magnitude of users. 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 light-weight and scalable, but do not 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 which impacts scalability. In this paper, we explore whether we can get the best of both worlds – the scalability of a policer with the rate and policy enforcement properties of a shaper. We answer this question in the affirmative with our system BC-PQP. BC-PQP 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. We implement our rate-limiter as a middlebox over DPDK. Our evaluation shows how BC-PQP achieves the rate and policy enforcement properties close to that of a shaper while being up to $7\times$ more efficient.

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, 45]. Programs like T-Mobile's "Binge on" [45] and Verizon's "Netflix & Max" [49] provide unlimited access to specific video streaming services, but limit the subscribers' network traffic outside of those services.

The rate limiting mechanim 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 satisy two more 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 [50], Cubic [26], Vegas [13], etc) to compete fairly with one another [34, 38, 39]. 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 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 the specified rate (by incrementing a counter) – a packet is allows 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 [22]. But the scalability provided by this choice has come along with notable

[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, 25, 30, 51, 52].

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, 22, 28, 47]. Shapers can adequately address these downsides of a policer, but at the cost of lower system efficiency (and scalability).

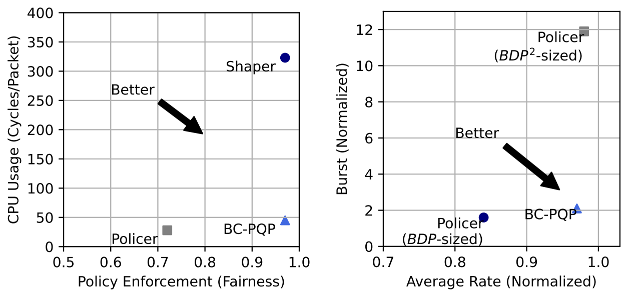
The question we explore in this paper is whether we can get the best of both worlds: *can we have the system efficiency and scalability of a policer, along with the network-level properties (ability to enforce desired rates and policies) of a shaper?*

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 a packet is transmitted, 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 mimicks the behavior of a token-bucket filter. In order to enforce different rate sharing policies, we extend such a policing system for each traffic aggregate to use *multiple* phantom queues – an incoming packets gets classified into one of these phantom queues (based on flow identifiers in the packet header fields), and is immediately transmitted or discarded based on the queue occupancy as described above. The phantom packets in each of these phantom queues are then dequeued (i.e. the byte counters are decremented) 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 an average, as long as the phantom queues are sufficiently sized.

While PQP correctly enforces the desired rates on average, the instantaneous rates can burst to much higher values (with the burst increasing with queue size). The minimum size required for enforcing correct average rates with phantom queues is very large to begin with ($O(BDP^2)$ as opposed to $O(BDP)$ 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,



(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

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 phantim 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.

We implement our system on a testbed comprising of 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 6× smaller burst (tail throughput deviation from 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 are often implemented on network routers, and dedicated hardware appliances. Recently, with the advent of software defined networking (SDN), traffic shapers are implemented in software as virtualized network functions for flexibility. They can support a large set of Quality of Service mechanisms such as Priority Queueing, Weighted Fair Queueing (WFQ) [41, 43].

Rate enforcement with traffic shapers. Traffic shapers maintain separate buffer for each traffic aggregate. An incoming packet gets 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 any 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 [48] 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. On modern x86 CPUs that feature Intel's Data Direct IO (DDIO), incoming packets are being DMAed to the CPU's Last Level Cache (LLC) and the CPU can classify them without incurring cache misses, but since they cannot be dequeued immediately they are eventually evicted to the main memory (DRAM). The CPU is constantly polling all available shapers (i.e., queues) and instructs the NIC to DMA packets out when allowed. This could be a relatively efficient operation if the shaper maintains a single FIFO queue and the packets are being buffered to contiguous memory, but it could become quite expensive when enforcing policies like DRR or prioritization with numerous flows: 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 per instruction) due to LLC misses.

This can be seen in Figure 1a. While the policy enforcement (fairness in this case) is quite effective, far more CPU cycles are spent per packet compared to a policer. Throughout the paper, we use CPU efficiency as a proxy for scalability. If a rate-limiting mechanism consumes higher number of CPU cycles per packet, it will require proportionally larger number of cores (and servers) to meet the scalability requirements.

2.2 Traffic Policer

Unlike traffic shapers that attempt to regulate traffic by buffering and delaying packets, network policers enforce rate limiting by dropping them 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, policer does not need to store any packets, and hence eliminates the overhead of memory-related bottlenecks.

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

1. Poor rate enforcement. Network 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 sometimes orders of magnitude higher than the enforced rate. Figure 1b 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 [22, 47]), 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 [38, 39]. 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 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 TBF as a Phantom Queue

Consider a policer implemented using a token-bucket filter, that enforces a rate of r and allows a burst of size B (as described in §2.2). We can realize the same policing system using a phantom queue with a (simulated) buffer of size B, served at rate r. When a packet of size s arrives at the policer, we first check if there is sufficient capacity in the phantom queue's simulated buffer. If the remaining capacity in the phantom queue is atleast 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 rightaway upon arrival. The phantom packets in the phantom queue are simply maintained as byte counters that get incremented and decremented upon enqueue and phantom 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.

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. 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, hash of source-destination addresses, etc). If the remaining buffer capacity in Q_i is atleast 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 analagous 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 "analagous shaper system" to refer to a hypothetical shaper system that applies the same enqueing and deqeuing 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 we can realize with PQP.

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, 41]. 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 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 experiences queuing delay from low priority if all downstream hops have capacity greater than r. In contrast, PQP can transmit burst of (real) low priority packets before transmitting high priority packets that arrive later (while the phantom low priority packets wait behind 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 – how the cumulative rate r is divided between individual queues on average. 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 $r_i = \frac{w_i r}{\sum_{Q_j notempty} w_j}$, where w_i is the weight of Q_i. A prioritization policy will serve a lower priority queue at rate of r minus the rate at which the higher priority queues are served (as driven by their packet arrival rates). [2]

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 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 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 long enough 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 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 are 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, inspite 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.

[2]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).

Theorem 1: *Over any time interval $\Delta t = t_2 - t_1$, as long as phantom queue Q does not go to zero i.e. $L(Q,t) > 0$, where $t_1 < t < t_2$, then the number of packets it accepts over duration Δt is bounded by $(r\Delta t \pm B)^+$.*

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 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 is provably achieved only as long as the phantom queue remains non-empty over the duration Δt. This requires correctly sizing the queue (as we discuss 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 that 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 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 a 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 (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

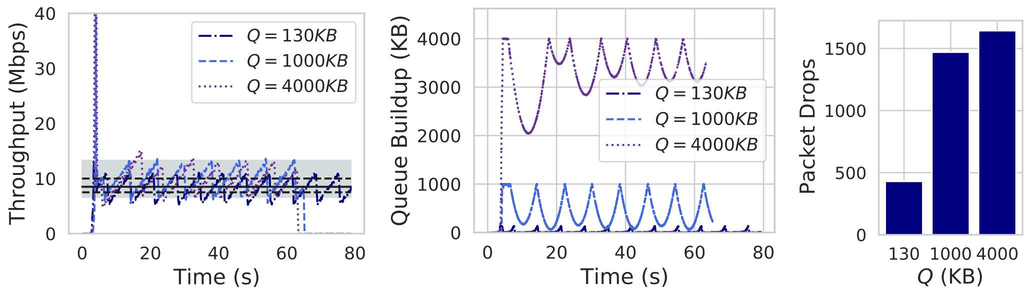
Guidelines on how to size the phantom queue depend on factors like rate r, RTT, and congestion control protocol used by the flow. We now analyze how phantom queues should be sized for correct average rate enforcement.

Notice that our bounds on enforced rates 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 rate higher than the policed rate of r) is able to keep it occupied. [3] This is analogous to how we reason about sizing shaper queues (that manage real packets)[9]. However, we find that 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[44]), 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 they are occupied by 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 atleast $\frac{BDP^2}{18} \times MSS$ 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

[3]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) Packet drops**

Figure 2: A Reno flow's behavior with differently sized phantom queues.

(additive increase multiplicative decrease) phase, and a phantom queue with buffer size atleast $\frac{BDP^2}{18} \times MSS$ bytes is needed to support this rate variation.

Why not BDP-sized queues?. The reason for larger buffer size requirement with 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 *cwnd* exceeds BDP, additional packets are queued and dequeued at rate r. Only when the acknowledgment for these additional packets has reached the sender, *cwnd* is updated by 1 – this slows down the feedback loop and gives more time for the buffered packets of the previous *cwnd* round to drain the queue, making more room for newer packets in the next *cwnd* round with a relatively smaller buffer. In the phantom queue, however, acknowledgment reaches within RTT time (irrespective of however long it takes the phantom queue to drain). With the shorter feedback loop, by the time packets for the next *cwnd* arrive, phantom packets for the previous *cwnd* are also present. So, in a physical queue, queue build-up increases by 1 packet after each *cwnd* update, whereas in phantom queues 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. During the slow start of a flow, it can burst at a very large rate. For example, consider a phantom queue sized for enforcing a rate of 15 Mbps and with a max RTT of 100 ms. 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. This also results in a high drop rate, as *cwnd* after a slow start is so high that it takes multiple rounds of packet losses and *cwnd* halving before it comes down to a value comparable to BDP, thus resulting in correct rate enforcement.

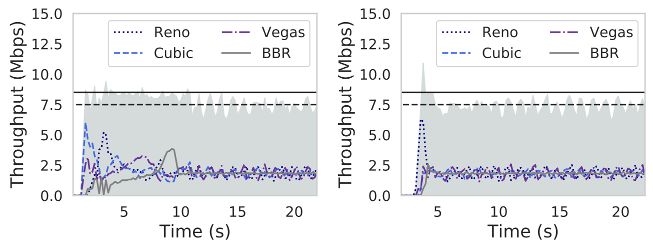
Empirical results. Figure 2 shows the impact of how we size the phantom queue buffer (B) on a Reno flow. We have a Reno flow with RTT of 100 ms and we want to enforce rate r of 10 Mbps. For such a flow, B needs to be at least 1000 KB. When B is set to a smaller size, queue occupancy ends up going to 0 more often which results in Reno *cwnd* not being able to reach the required peak, and thus incorrect rate enforcement. Whereas, when B is too large, while we have correct rate enforcement in the steady state, we can have a very large burst and higher drop rate. 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. Should each queue be sized by $O(BDP^2)$ rule to ensure correct rate enforcement even when only one queue is active? 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 3a 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[4]. Since phantom queues allow such a large burst to go through, the packets are really bottlenecked at the secondary bottleneck which results in poor policy enforcement i.e. fairness in this case.

Most policers deployed today are sized by a fixed value [21] and as we discuss this has an inherent trade-off between correct rate enforcement and high burst. Previous work has explored the idea of adapting the policer bucket size by looking at the behavior of the flow passing through the policer e.g. by increasing the token bucket size if the flow does not borrow tokens from the bucket for large periods of time [47]. This takes multiple adjustments to converge to a correct bucket size. Doing this for multiple queues is more challenging because their queue/bucket sizes would also depend on the demands of flows in other queues/buckets.

So how do we deal with this conundrum? We need the phantom queue buffers to be large enough for correct rate

[4]This is a common occurrence e.g. service providers to rate enforcement before flows actually hit RAN which may have bandwidth comparable to enforced rate.



(a) Phantom Queues sized Q

(b) Phantom Queues with magic packets.

Figure 3: $r = 7.5$ **Mbps shared across 4 flows with different CC protocols with a secondary bottleneck of** 8.5 **Mbps afterward.**

enforcement in the steady state, but would still like to avoid the large transient burst. 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 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 of which depends on various factors like flow's congestion control protocol, RTT, enforced rate r, and demands of other flows, we ask how can we 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, 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 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 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 P. 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, while 0 otherwise. 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 active queues.

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). Whereas, if 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 [5]. 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. 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 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, any queue Q_i gets $X_i = 0$ if any of the other higher-priority queues are active. As before, the cost of this small burst further amortized, the longer the queues remain occupied.

We configure T according to p99 *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 3b shows how flows see a very small and controlled burst which results in fair sharing of 7.5 Mbps across 4 different flows.

[5]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.

We discuss some design insights below:

Why do we need to drain the magic packets? Firstly, to avoid under-utilization. Moreover, since we drain the queue as soon as the flow becomes inactive, we can immediately allocate the spare rate elsewhere. In normal PQP with very large queues, when a flow becomes inactive, it takes a long time before its phantom queue is drained, this results in transient under-enforcement of rate even though other flows are active.

How BC-PQP automatically adjusts to different dequeue rates of different phantom queues? A burst-controlled phantom queue estimates its dequeue rate independently and automatically adapts to avoid bursts and do correct rate enforcement. This makes it easily composable within any arbitrary rate-sharing policy.

How to set the phantom queue buffer size B_i in a BC-PQP? A full phantom queue for an active flow automatically makes room for new packets at the correct rate. As long as the queue is large enough to not go to zero during normal operation of a saturating flow, it does not matter how high a value we set for the phantom queue size.

5 IMPLEMENTATION

For our evaluation, we use microbenchmarks to test various aspects of rate and policy enforcement of our design against baselines. We have developed a middlebox responsible for transparently enforcing network traffic rates using a kernel-bypass stack based on Intel's DPDK. For this, we implement BC-PQP and other mechanisms discussed previously including shaper, policer, and fairpolicer. We run microbenchmarks on Azure public cloud and use three standard F8sv2 Virtual Machines running Ubuntu 22.04 dedicated for the client, server, and the middlebox respectively. For traffic, we create flows of different sizes using TCP sockets and control the Congestion Control algorithm at a per flow granularity. We use Linux kernel implementation of all congestion control protocols. Furthermore, we also 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 now evaluate the following:

• BC-PQP's ability to do correct rate and policy enforcement efficiently. We compare BC-PQP against baselines like traffic shapers, traffic policers, and FairPolicer in §6.1. We also report the system efficiency of BC-PQP compared to other baselines in §6.2

• We also show the feasibility of different rate-sharing policies that can be enforced correctly using BC-PQP in §6.3.

• Lastly, we show some experiments with real-world applications like video streaming and web browsing to demonstrate how BC-PQP can help improve application QoE in §6.4.

6.1 Rate Enforcement

In this experiment, we do rate enforcement for 100 flow aggregates, each consisting of multiple flows. Our setup consists of 3 machines, a sender machine, a rate enforcer machine, and a receiver machine. The sender machine starts multiple flows of different sizes ranging from a few 10s of KBs to 100s of MBs at different times, using different congestion control protocols amongst New Reno, Cubic, BBR, and Vegas. 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. We have a mix of aggregates, in half of the aggregates all flows use the same congestion control protocol and have the same RTT, 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, whereas others only have short on-and-off flows, whereas a third subgroup has both backlogged and short on-and-off flows.

The traffic from the sender machine is routed through the rate enforcer machine. Our goal is to enforce a rate r for each aggregate and do per-flow fairness within each aggregate. We test with different enforced rates r, i.e. 1.5, 7.5, 25, 50, and 100 Mbps. We compare BC-PQP with following baselines:

• **Shaper**: Each shaper has multiple queues each sized according to maximum BDP.

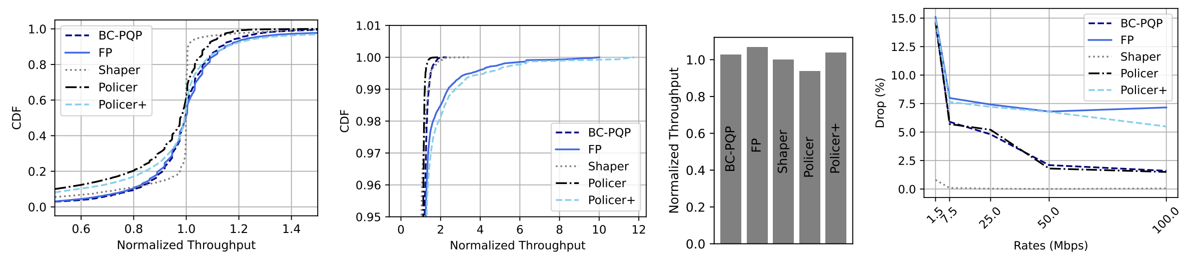
• **FairPolicer (FP)**: We size the bucket B for FP to be the maximum of any flow's requirement i.e. we pick the maximum RTT and compute B needed for correct rate enforcement for New Reno and Cubic, and pick the maximum value. For small values of RTT and rate, Cubic requires a larger bucket size, whereas in other cases New Reno requires a larger bucket size.

• **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 100 ms respectively.

The rate enforcer machine forwards traffic to the receiver machine, where per-flow throughput is measured over 250 ms windows. We sum the throughput of each flow within each aggregate over these 250 ms windows and normalize this aggregate throughput by enforced rate r. Figure 4 summarizes the



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

Figure 4: Aggregate rate enforced by BC-PQP and other baselines, 4a and 4b show distribution of aggregate throughput measured over 250 ms windows normalized by enforced rate, 4c shows normalized average aggregate throughput and 4d shows drop rate at different enforced rates

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 4a. It can be seen that the shaper does accurate rate enforcement over short time scales, but for the most part, the instantaneous rate for other baselines also stays within small bounds (roughly $0.8r$ to $1.2r$).

2- However, Policer+ and FP cause a much larger burst and have a long tail for aggregate throughput (figure 4b). 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 (as shown previously in figure 3).

3- It can also be observed in figure 4a that the line for policer is slightly shifted left which results in average throughput being lower than desired rate r. This can be seen in figure 4c, which reports the average of all non-zero aggregate throughput measurements. The number is higher for FP and Policer+ because bursty throughput points skew the average up.

4- Due to lack of buffering, all schemes other than the shaper [6] induce a higher number of packet drops. 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 go into their slow start and overestimate their congestion window. 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 delay. 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 (§6.4) – so the trade-off between them is unclear.

Overall, BC-PQP results in correct aggregate rate enforcement with a small burst compared with FP or correctly sized policer which can burst more than 10 times the enforced rate.

[6]While deep buffers in shapers can reduce or for some CCs eliminate packet drops, this alternatively induced high queuing delays .

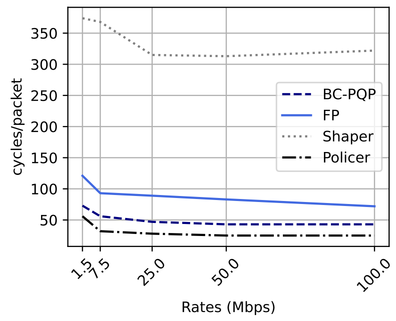
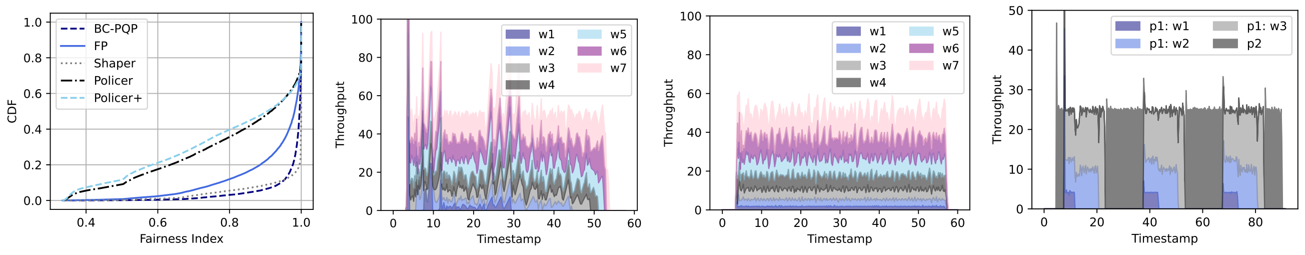


Figure 5: CPU cycles spent per packet

6.2 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 a system being able to handle fewer packets per second. Figure 5 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.



(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 6: Policy enforcement with BC-PQP and other baselines: 6a shows per-flow fairness between flows within an aggregate, 6b and 6c show weighted fairness achieved by FairPolicer and BC-PQP respectively and 6d shows a nested policy with prioritization and weighted fairness using BC-PQP.

6.3 Policy Enforcement

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

6.3.1 Per-flow Fairness. For the experiment from §6.1, we measure the per-flow throughput over 250 ms windows within each aggregate and estimate fairness using Jain's Fairness Index. CDF of this fairness index is reported in figure 6a. We can see the shaper achieves close to perfect fair sharing of enforced rate as expected and policers are unable to do so. BC-PQP also achieves fairness comparable to the shaper. While FP does better than policers, it falls a bit short for two reasons. Firstly, since it allows a large burst to go through, flows with smaller RTT or aggressive CC can burst faster. Secondly, for AIMD-style CC protocols, if flows have large RTTs, they achieve lower than their fair share with FP, as also reported in FP paper [38]. This happens because such a flow needs a very large bucket size, if it is not large enough and the flow cannot keep the bucket active, it won't be able to achieve its fair share. Setting a very large bucket, as we have observed, can cause large bursts, high packet drop rates, and poor rate enforcement. BC-PQP is able to get around this by sizing queues to a very large number but also reacting to queue fill rate as described in §4.

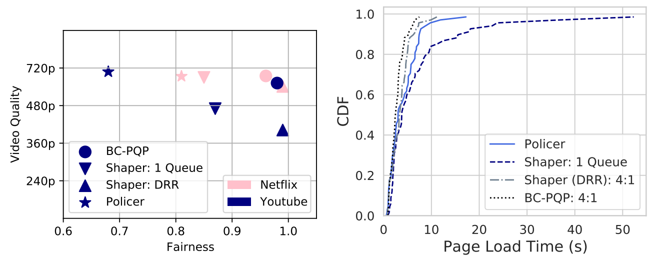
6.3.2 Weighted Fairness. In this microbenchmark, we show how BC-PQP can do accurate weighted sharing within an aggregate. We enforce a rate of 50 Mbps and share it between 7 flows, each with weights from 1 to 7. All flows start at the same time and are sized proportional to their weights so that they should complete at the same time if the rate is shared fairly between them. We also adapt the token allocation logic in FairPolicer to make it do similar weighted sharing as well. Figures 6b and 6c 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 bucket's 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.

6.3.3 Prioritization and Nested Policies. In this section, we show the feasibility of enforcing prioritization and nested policies with a phantom queue with a microbenchmark. We have 4 flows divided into two priority groups: 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, p2 is the lower priority group with a single backlogged flow. Figure 6d 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.

6.4 Real World Applications

We show the feasibility of using BC-PQP to enforce rates with policy enforcement for different applications. We look at scenarios where an enforced rate r is shared by different kinds of flows.

6.4.1 Video Streaming. We look at a scenario where a rate of 3 Mbps is shared between a video stream and some other traffic. Cellular service providers often use policers or single queue shapers to do such rate enforcement for each user. Such rate enforcement is done to ensure careful resource management of limited RAN resources otherwise bandwidth hungry video stream flows can hog all resources resulting in uneven and poor service across users. We simulate this scenario by sharing 3 Mbps between video flow and the rest of the traffic (webpage loads, downloads, etc.). We want to ensure these categories of flow share the enforced rate of 3 Mbps fairly.



(a) Fairness vs Video QoE **(b) Web page load times**

Figure 7: Video streaming and web browsing with PQP.

We repeat this experiment using 3 videos from YouTube and 2 videos from Netflix. YouTube uses BBR [3], while Netflix uses New Reno [44] 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 7a shows the average video quality achieved vs the fairness index ensured between video flow and the rest of the traffic. BC-PQP shared the rate perfectly between the 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, and in the process affect video quality or fairness with the rest of the traffic. While shaper with DRR ensures fairness between traffic, YouTube videos' quality suffers. This is likely due to an additional queuing delay introduced due to buffering of packets. However, the precise reason for this is not clear because of the lack of visibility into YouTube's ABR algorithm.

6.4.2 Web Browsing. Similar to the previous setup, we share the 3 Mbps link between a download flow and web browsing traffic. We open 50 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 4:1. CDF of web page load times is reported in figure 7b. BC-PQP achieves 2-8× 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, 24, 36–39]. These solutions usually include traffic shapers [7, 24, 36, 37], which buffer packets in memory or traffic policers [38, 39, 47]. Previous works have noted the limitations of traffic shapers and policers, namely traffic shapers are expensive to implement [21, 38], whereas traffic policers suffer from poor rate enforcement and high packet losses [21, 28, 47].

Traffic policers are known to be difficult to configure [22, 47]. Guidelines around configuring policer bucket sizes are not homogenous and depend largely on which factor is more important: correct rate enforcement [38, 39] or small burst [6, 22]. [47] 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 not need to vary phantom queue sizes, instead, it relies on the 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, 46]. Past work has looked into various mechanisms to implement policies like per-flow fairness, weighted fair queuing, or prioritization [20, 34, 35, 40, 42]. 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, 40]. Using bufferless mechanisms for such policy enforcement has not been explored that much. Recent work attempts to make token bucket policers fair when flows with different congestion control protocols pass through it [38, 39]. However, this does not extend to other rate-sharing policies. Moreover, it suffers from burstiness and some level of RTT unfairness [38].

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 work the same as a token bucket in principle, albeit it has also been called a pseudo queue [15, 23, 27]. Our key contribution lies in augmenting a policer with *multiple* phantom queues, and showing how it can do policy-rich rate enforcement.

8 CONCLUSION

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).

This work does not raise any ethical concerns.

REFERENCES

[1] [n. d.]. ([n. d.]). https://www.att.com/support/article/wireless/KM1169198/

[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 *Proceeding of the 23rd ACM workshop on network and operating systems support for digital audio and video*. 19–24.

[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 (NSDI 12)*. 253–266.

[9] Guido Appenzeller, Isaac Keslassy, and Nick McKeown. 2004. Sizing router buffers. *ACM SIGCOMM Computer Communication Review* 34, 4 (2004), 281–292.

[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* 56, 3 (2018), 172–179.

[11] Ralph P Boas Jr and John W Wrench Jr. 1971. Partial sums of the harmonic series. *The American Mathematical Monthly* 78, 8 (1971), 864–870.

[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 *Proc. 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 *Proc. ACM Conference on Communications Architectures, Protocols and Applications*. 24–35.

[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 *Proceedings of the Thirty-First Workshop on Hot Topics in Networks (HotNets) (HotNets '23)*. Association for Computing Machinery, New York, NY, USA.

[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* 9, 3 (1991), 335–342.

[16] Frank Cangialosi, Akshay Narayan, Prateesh Goyal, Radhika Mittal, Mohammad Alizadeh, and Hari Balakrishnan. 2021. Site-to-site internet traffic control. In *Proceedings of the Sixteenth European Conference on Computer Systems*. 574–589.

[17] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2016. BBR: Congestion-Based Congestion Control. *ACM Queue* 60, 2 (2016), 58–66.

[18] Saoussen Chaabnia and Aref Meddeb. 2018. Slicing aware QoS/QoE in software defined smart home network. In *NOMS 2018-2018 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 *2021 IEEE 18th Annual Consumer Communications & Networking Conference (CCNC)*. IEEE, 1–4.

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

[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 *Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016*, Marinho P. Barcellos, Jon Crowcroft, Amin Vahdat, and Sachin Katti (Eds.). ACM, 468–482. https://doi.org/10.1145/2934872.2934873

[22] 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 *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA, 468–482. https://doi.org/10.1145/2934872.2934873

[23] G Gallassi, G Rigolio, and Luigi Fratta. 1989. ATM: Bandwidth assignment and bandwidth enforcement policies. In *1989 IEEE Global Telecommunications Conference and Exhibition'Communications Technology for the 1990s and Beyond'*. IEEE, 1788–1793.

[24] 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* 4, 4 (1996), 482–501.

[25] 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*.

[26] Sangtae Ha, Injong Rhee, and Lisong Xu. 2008. CUBIC: a New TCP-friendly High-Speed TCP Variant. *ACM SIGOPS Operating System Review* (2008), 64–74.

[27] 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*, Vol. 3. IEEE, 1248–1255.

[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*. 43–48.

[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 *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. 1–14.

[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* 31, 4 (2001), 123–134.

[32] Srisankar S Kunniyur and Rayadurgam Srikant. 2004. An adaptive virtual queue (AVQ) algorithm for active queue management. *IEEE/ACM Transactions on networking* 12, 2 (2004), 286–299.

[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] Rong Pan, Lee Breslau, Balaji Prabhakar, and Scott Shenker. 2003. Approximate fairness through differential dropping. *ACM SIGCOMM Computer Communication Review* 33, 2 (2003), 23–39.

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

[36] Ahmed Saeed, Nandita Dukkipati, Vytautas Valancius, Vinh The Lam, Carlo Contavalli, and Amin Vahdat. 2017. Carousel: Scalable traffic shaping at end hosts. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication.* 404–417.

[37] 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 (NSDI 19).* 17–32.

[38] 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).

[39] 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 2021, Vancouver, BC, Canada, May 10-13, 2021.* IEEE, 1–10. https://doi.org/10.1109/INFOCOM42981.2021.9488761

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

[41] M. Shreedhar and George Varghese. 1995. Efficient Fair Queueing Using Deficit Round Robin. *ACM SIGCOMM Computer Communication Review* (1995), 231–242.

[42] 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.* 231–242.

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

[44] 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 *Proceedings of the ACM SIGCOMM 2023 Conference.* 754–768.

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

[46] 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.

[47] 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.* IEEE, 72–77.

[48] George Varghese and Anthony Lauck. 1987. Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility. In *Proceedings of the Eleventh ACM Symposium on Operating System Principles, SOSP 1987, Stouffer Austin Hotel, Austin, Texas, USA, November 8-11, 1987,* Les Belady (Ed.). ACM, 25–38. https://doi.org/10.1145/41457.37504

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

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

[51] 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 *Proceedings of the 2012 ACM Conference on Ubiquitous Computing.*

[52] Yiannis Yiakoumis, Sachin Katti, and Nick McKeown. 2016. Neutral Net Neutrality. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16).* Association for Computing Machinery, New York, NY, USA, 483–496. https://doi.org/10.1145/2934872.2934896

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 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 8 to hold the additional packets that are sent beyond rate r, which comes out to be $\frac{BDP^2}{18} \times MSS$ bytes.

B YOUTUBE'S VIDEO STREAM ANALYSIS

The time series for one video with different schemes is shown in figure 9. Since YouTube uses BBR, with a policer, the video

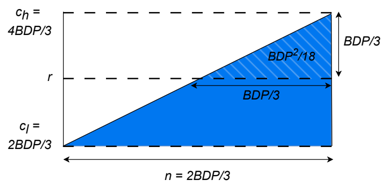
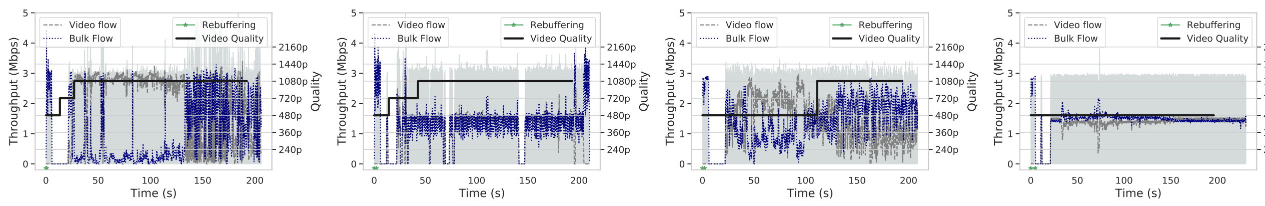


Figure 8: Reno's *cwnd* progression over *n* RTTs, we need phantom queue to be at least the size of shaded region.

flow hogs most of the bandwidth thus achieving high video quality but at the cost of affecting the rest of traffic sharing the link. 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.



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

Figure 9: A youtube video stream sharing 3 Mbps link with some other traffic with different schemes.