Cooperative Rate Control for the Internet

Christopher Canel **Zecheng He**
Elliot Lockerman **Adithya Abraham Philip**
Justine Sherry **Srinivasan Seshan**

May 2024
CMU-CS-24-129

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This work was supported in part by the National Science Foundation through grant 2212390.

Keywords: Internet, rate allocation, Transmission Control Protocol (TCP), congestion control, fairness, machine learning

Abstract

On the Internet, rate control objectives differ between sender and receiver. For example, senders often want their service to claim as much bandwidth as possible, while receivers must balance the performance of the many services they interact with. Disconnects like this lead to conflicting expectations about rate allocation granularity, algorithm, and optimization metric. This paper explores these challenges and proposes *cooperative rate control*, whereby the receiver jointly enforces rate control instead of relying on the sender alone, leading to a natural union of their individual objectives. As exemplars of this cooperative approach, we design two receiver-side policies: (1) ServicePolicy, which enforces per-service rate control rather than per-flow fair shares, and (2) FlowPolicy, which ameliorates undesirable interactions between congestion control algorithms. To enable these policies, we present a general receiver mechanism for monitoring arriving packets, inferring whether a flow is adhering to the receiver's policy, and leveraging TCP's built-in flow control to transparently signal the sender to back off. Our techniques do not modify the sender, network devices, or TCP header. We show in emulation and with Internet experiments that when endpoint policies differ, cooperative rate control improves compliance with the receiver's policy by 24%–221% with an acceptable penalty to overall utilization.

1 Introduction

For decades, the research community has faced the challenge of how to allocate bandwidth between competing flows on the Internet. While these debates play out between humans [15, 26], there are two places that this *tussle* [21] comes to a head mechanistically on the Internet. First, *senders* use TCP congestion control algorithms (CCAs) with some approach to rate control, such as converging towards equal-rate bandwidth allocations [18] — although not all CCAs achieve their goals [71]. Sending applications map their demand onto multiple congestion-controlled flows according to their own design criteria. Second, *routers* may use advanced queuing schemes to enforce the network operator's rate policy, such as access control based on user contracts, max-min flow-rate fairness [22], or traffic prioritization (e.g., VoIP traffic over web browsing [8]). The final rate achieved by a flow is therefore the *union* of the sender and network operator's rate control policies.

We argue that an important stakeholder is left out of this discussion on today's Internet: the rate policies, requirements, and observations of the *receiver*. Specifically, we identify three challenges that arise due to conflicting receiver policies regarding rate control (1) granularity, (2) algorithm, and (3) optimization metric. Consider the following common scenarios:

- **Example 1 — Differences in rate control granularity:** A client downloads files from a file sharing service that opens multiple parallel connections. The client also streams a video over a single flow and would like to share bandwidth equally at the service granularity. However, each service implements rate control at the per-flow granularity, and the network uses per-flow fair queuing [22, 55].

- **Example 2 — Differences in rate control algorithm:** Most traffic on a network uses a particular loss-based CCA, like TCP Reno [42]. A client accesses a service using a different loss-based CCA, such as TCP Cubic [33], that implements a convergence algorithm that is tailored for higher bandwidth networks. The network provider does *not* support fair queuing, but the client wants to prevent the Cubic flow from negatively impacting other traffic sharing the same bottleneck.

- **Example 3 — Differences in rate control objective:** A company seeking to reduce request latency deploys in their internal network a delay-based CCA that optimizes for keeping switch buffers short [13]. However, internal flows must coexist with incoming Internet traffic that uses a buffer-filling algorithm that prioritizes high throughout and claims the majority of the bandwidth. Routers do not use separate queues for external traffic.

In the above scenarios, the client (receiver) has a rate allocation policy that differs from that of the sender and which the network either will not or cannot enforce on their behalf. Today, clients have no recourse in these scenarios.

In this paper, we propose *cooperative rate control*, which gives both endpoints a stake in bandwidth allocation decisions. Services, routers, and the sender congestion control algorithm — whether based on loss, delay, or network feedback — remain unchanged, but we advocate for the creation of a parallel mechanism on the receiver. We envision the goal of cooperative control to be a natural union of well-intentioned policies from these entities, *not* as a mechanism to provide

strong security guarantees. An actively malicious sender may still transmit arbitrarily quickly and, unlike queuing-based techniques, cooperative control does not interfere with the packet stream to delay or drop packets.

To be clear, cooperative control is also not a new CCA and, in fact, implementation of such a mechanism at the receiver poses different challenges than traditional CCA design. For example, since the receiver does not control the packet stream or observe state in the network stack related to ongoing congestion avoidance, we must implement the measurement and control of rate both separately and differently from the core logic of a CCA, which already tracks network state such as loss rate and latency. We also cannot rely on techniques such as bandwidth probing or packet scheduling as part of the rate control approach.

As a proof-of-concept for cooperative rate control, we design RateMon, a receiver-only extension to the TCP stack that detects when a flow exceeds a *receiver-defined* rate policy and triggers the sender to decrease its rate. RateMon infers characteristics of the bottleneck link based only on information implicitly encoded in a flow's arriving packets (e.g. packet interarrival time, loss rate, latency). When the receiver identifies a flow which violates its policy, it commandeers TCP's built-in flow control and tunes the receiver's advertised window to signal the sender to back off. RateMon is backwards compatible with standard TCP variants and requires modifications to the receiver only; not the sender, network switches, or the TCP header. Furthermore, cooperative control does not require coordination between receivers and does not require a broad consensus on rate policies, such as fairness.

Note that the above description of cooperative control focuses on control *mechanisms*, meaning the implementation of how the receiver monitors and modifies the rate. The receiver's rate control *policy* is separate from these mechanisms. In cooperative control, the receiver is free to implement an arbitrary policy based on user requirements, as described in the above examples. We demonstrate the efficacy of cooperative rate control by implementing and evaluating two separate receiver policies that address the three challenges above. First, to address differences in endpoint policy granularity (Example 1), *ServicePolicy* intercepts services that open multiple connections and holds the entire service to a rate appropriate for a single flow. The receiver calculates a rate for each service as if they were using a single TCP Reno flow [51], then divides that rate among the service's flows. Our results show that ServicePolicy prevents applications that use multiple flows to transfer data from acquiring a disproportionate share of the network bandwidth, improving compliance with the receiver's policy of service-level fairness by 23.9%.

The second policy, *FlowPolicy*, focuses on policy differences regarding rate control algorithm (Example 2) and optimization metric (Example 3) at the per-flow granularity. FlowPolicy leverages a machine learning model based on statistics from incoming packets to infer if a flow is above, below, or near the policy's target behavior, and then provides feedback to the sender using a simple multiplicative increase, multiplicative decrease controller. Our evaluation shows that FlowPolicy is effective at sharing resources evenly at a per-flow granularity in diverse scenarios: FlowPolicy improves per-flow rate fairness by 32.4%–220.9%, at the cost of decreased utilization in some cases, benefits all CCAs in Linux, supports flows across the Internet, and generalizes to application-limited flows as well.

In these policies, we do not assume that the receiver observes the bottleneck link or all flows

that traverse it, or that all of the receiver's flows share the same bottleneck. While we focus on ServicePolicy and FlowPolicy, RateMon is a general framework for cooperative rate control and can implement other policies related to access control, fairness, or resource management. RateMon is available open source at **github.com/cmu-snap/ratemon**.

Section 2 examines the above three challenges in detail as well as existing steps towards cooperative rate control. Section 3 describes the design of RateMon, including its mechanism for cooperative control. Sections 4 and 5 present ServicePolicy and FlowPolicy, respectively. Section 6 evaluates these techniques to demonstrate the efficacy of cooperative control in giving the receiver a stake in rate allocation on the Internet. Section 7 discusses related work, and Section 8 concludes.

Ethics statement: This work raises no ethical concerns.

2 Background & Motivation

Requirements for bandwidth allocation have evolved with the growth of the Internet, and will continue to do so for emerging use cases. This section argues that to avoid being "left behind" in this evolution, the receiver deserves a stake in bandwidth allocation decisions. We first define the relationship between rate control and fairness. Then, we explore three challenges with the state-of-the-art in bandwidth allocation arising from differing endpoint policies related to rate allocation (1) granularity, (2) algorithm, and (3) optimization objectives. Finally, we describe existing multi-entity mechanisms that point the way towards our proposal for cooperative rate control.

2.1 Rate control and fairness

Bandwidth allocation is commonly discussed in the context of a *fair* bandwidth allocation. We consider each entity (senders, receivers, routers, application developers, network operators, etc.) to have an individual (i.e., local) rate allocation policy that governs their approach to fairness. Many systems try to allocate resources fairly, but their rate allocation policies, and therefore their definitions of fairness, often differ. For example, systems may choose objectives such as max-min fairness [29], round trip time (RTT) fairness (where the flow rates are scaled inversely proportionally to the RTT), quality-of-experience fairness [56], absolute fairness, slowdown [75], etc. They also choose different granularities of fairness, such as per flow, per service, per user, per network, and per dollar. In achieving their definition of fairness, entities turn to unique optimization criteria, like high throughput or low delay, and use different algorithms to achieve their goals. This paper does not assume one definition and instead uses fairness to refer generally to the study of rate allocation, which encompasses application design, load balancing, traffic engineering, congestion control, flow control, and other related concepts.

Throughout this paper, we use the terms *incumbent* and *newcomer* to differentiate applications, flows, and algorithms. Incumbent techniques are those that already adhere to the receiver's local rate control policy, thus defining the status quo for fairness on the receiver's network. A newcomer is a technique that performs rate allocation using a different granularity, algorithm, or optimization goal than the incumbents and therefore may require intervention from cooperative rate control to

adhere to the receiver's policy. For example, a network dominated by incumbent TCP Reno flows that achieve per-flow fairness interacts with a newcomer TCP Cubic flow using a different ramp-up algorithm.

2.2 Challenges with bandwidth allocation today

We present three categories of challenges that arise due to differing entity policies related to rate allocation (1) granularity, (2) algorithm, and (3) optimization metric. For each of these categories, we introduce a running example to demonstrate how cooperative rate control offers benefits over the status quo for that particular class of problem.

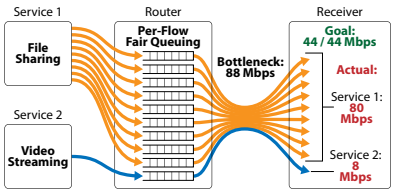
2.2.1 Challenge 1: Different allocation granularity

Entities consider rate allocation at different granularities. To illustrate this challenge, we explore how an application that uses multiple concurrent flows can claim more bandwidth than an environment based on per-flow rate control would normally allocate to another application that uses only a single flow.

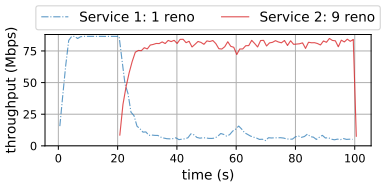
Sending applications, also known as "services", impact rate allocation in two ways: (1) their offered demand and (2) how they map that demand onto flows. First, some applications explicitly adapt their sending rate based on bandwidth estimates, e.g., video bitrate adaptation that provides a smooth viewing experience. This paper focuses on bulk download applications that are bottlenecked in the network, although we evaluate interactions with application-bottlenecked flows in Section 6.2.4. Second, applications impact their rate allocation by how they divide demand across flows. Application developers have long recognized the importance of logically separate streams between two endpoints for the purposes of eliminating head-of-line blocking and simplifying application design. Some application designs, such as HTTP/1.1 persistent connections [57], HTTP/2 [69], and QUIC [40] multiplex separate streams across a single flow. However, to improve load balancing, avoid network hotspots, improve loss recovery, and achieve a higher aggregate bandwidth, many applications leverage multiple TCP flows. A variety of systems (such as [9, 10, 11, 58, 70], described in Section 7) have considered rate allocation at the granularity of parallel flows between two endpoints.

Consider **Example 1** (Figure 1) from the introduction. Suppose that a receiver's policy is to share bandwidth equally between services. Figure 1a shows a client accessing two services, an incumbent video streaming service, such as YouTube [74], that uses a single connection, and a newcomer file sharing service, such as MEGA [52], that opens multiple parallel connections, in this case nine. The client's access link is 88 Mbps, the RTT is 23 ms, and the bottleneck queue size is 512 packets ($\approx$ 3$\times$BDP [1]). All flows use the same CCA (Reno), and each flow independently converges to an equal share of the bandwidth. Since the sender CCAs provide per-flow rate allocation, the file-sharing service using nine flows consumes almost all of the link bandwidth. Figure 1b shows a throughput trace for this example, where the video streaming service starts first, followed 20 seconds later by the file sharing service.

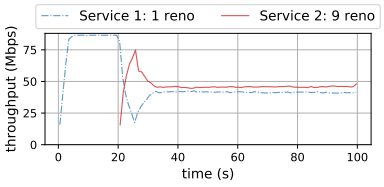
[1] The bandwidth-delay product (BDP), which is equal to the bottleneck bandwidth times the RTT, is the volume of data that a network path can transport before inducing queuing.



(a) A file sharing service uses nine flows and a video streaming service uses one flow.

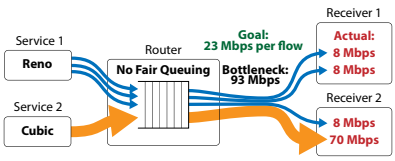


(b) The file sharing service claims over 75% of the bandwidth, but the client desires that both services get the same allocation.

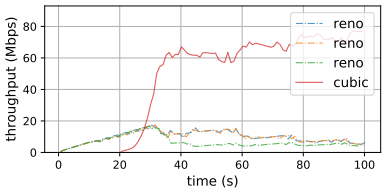


(c) ServicePolicy limits all the flows in each service collectively to the behavior of one Reno flow.

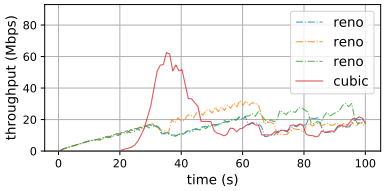
Figure 1: Example 1: In an environment of per-flow rate allocation, a service using many flows claims a higher rate.



(a) Service 1 uses three Reno flows and Service 2 uses one Cubic flow, with no fair queuing in switches.

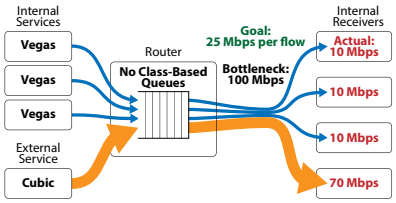


(b) The one Cubic flow claims over 65% of the bandwidth, but the client desires that each flow get a 25% bandwidth allocation.

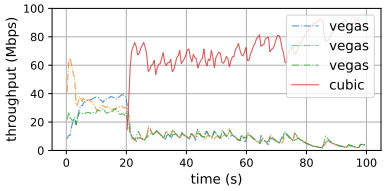


(c) FlowPolicy detects that the Cubic flow is unfair and dynamically guides is to an appropriate rate.

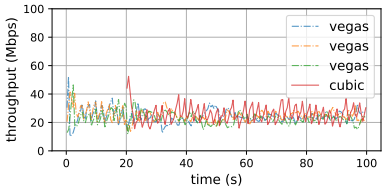
Figure 2: Example 2: Two CCAs with the same metric (loss) but different algorithms do not converge to equal rates.



(a) Internal traffic uses delay-based Vegas and external traffic uses buffer-filling Cubic.



(b) The Cubic flow claims ≈70% of the bandwidth, when it should claim 25%, because the Vegas flows back off earlier.



(c) FlowPolicy detects that the Cubic flow is causing loss and high delay, and forces it to back off.

Figure 3: Example 3: CCAs with different congestion metrics (delay vs. loss) do not converge to equal rates.

Today, a client seeking to enforce *per-service* control instead of per-flow control is left with little recourse. Some network routers implement per-flow fair queuing (FQ) [22, 55] to provide strong enforcement of flow-level fairness (as illustrated in Figure 1a), but per-flow FQ has no benefit in this situation because the receiver's policy is for per-service fairness. Employing FQ at a service granularity is a potential solution but typically faces significant deployment hurdles. Service-level rate control proposals (such as [9, 10, 58, 70]) typically operate at the sender (i.e., the service), not the client. The client could prevent parallel flows by blocking some connections with a firewall, but this risks negating other benefits of multiple flows and significantly impacting application behavior. Cooperative rate control offers an alternative: give the receiver influence over rate allocation at the transport layer, externally from the application. Figure 1c shows how ServicePolicy, described in Section 4, holds each service to its independent fair rate.

2.2.2 Challenge 2: Different rate control algorithms

Even within the same rate control granularity, such as per-flow fairness, congestion control algorithms often implements techniques and optimize for congestion metrics that are incompatible. We start by exploring interactions between CCAs that optimize for the same metric (in this case, high throughput) but do so using different algorithms, and then consider differing optimization metrics (throughput vs. delay) in Challenge 3.

A CCA's rate control algorithm — how it probes for bandwidth and backs off under congestion — determines its interactions with other flows and the rate outcome. A simple example is TCP Tahoe and its descendants, where rate convergence and fairness arise due to the sender's additive increase, multiplicative decrease algorithm [18, 43].

To illustrate this challenge, we explore two congestion control algorithms, Reno and Cubic, which both use loss as a signal of congestion and have similar optimization goals, but do not converge to equal rates due to differences in their algorithms. Consider **Example 2** (Figure 2) from the introduction. Suppose that a receiver's policy is to adhere to per-flow fairness. Suppose that most traffic on the receiver's network, including Service 1, uses Reno. Suppose that the client concurrently accesses Service 2, which uses Cubic [33]. Designed for higher BDP networks, Cubic ramps up quickly when its current rate is far from its estimate of its bandwidth fair share, but slowly as it nears it. CCAs may be fair to other flows using the same CCA: Reno and Cubic flows converge to equal-rate fairness (assuming equal RTT) with their own traffic due to the use of algorithms like additive-increase, multiplicative-decrease [18].

However, troublesome interactions arise when algorithms mix in the wild. While Reno and Cubic respond to the same congestion signal and both converge to a full bottleneck queue, differences in their ramp-up characteristics mean that when the two interact, a Cubic flow typically consumes more bandwidth. Figure 2a shows an example where three Reno flows compete with one Cubic flow in a network with a 93 Mbps bottleneck link, an RTT of 152 ms, and a bottleneck queue of 1024 packets ($\approx 1\times$BDP). Figure 2b shows the throughput over time, where the incumbent Reno flows are joined 20 seconds later by the newcomer Cubic flow. The single Cubic flow claims the majority of the bandwidth. Figure 2c shows how FlowPolicy, described in Section 5, helps network traffic adhere to a policy of per-flow fairness in the presence of diverse CCAs that are not designed to interact with each other.

2.2.3 Challenge 3: Different optimization metrics

Finally, a CCA's optimization metric also impacts its interaction with other flows. Commonly-used CCAs like Reno and Cubic optimize for high throughput. To that end, they assume that the network is not congested until packets are lost. This means that such *loss-based* algorithms wait for the bottleneck queue to fill and drop packets before the algorithm detects congestion and backs off. This queuing induces significant delays for network traffic. To summarize, loss-based CCAs optimize for throughput at the cost of high delay. In contrast, *delay-based* algorithms, such as TCP Vegas [13], monitor the RTT as a measure of congestion: as congestion occurs and queues build up, the RTT increases and these algorithms back off *before* losses occur. Vegas, in particular, attempts to keep the queue length between one and three BDPs [13]. However, by keeping queues short, these techniques risk under-utilizing the bottleneck link. In essence, delay-based CCAs optimize for low delay at the potential cost of underutilization.

When loss-based and delay-based CCAs coexist, the delay-based CCAs will decrease their rate earlier than the loss-based CCAs, causing the latter to obtain a higher fraction of the available bandwidth. Consider **Example 3** (Figure 3) from the introduction. Suppose that a company wants to reduce network latency to improve response time. The company deploys delay-based Vegas on its internal servers, but flows originating from the Internet still use loss-based CCAs like Cubic. Figure 3a shows an example where three incumbent Vegas flows are joined by a single newcomer Cubic flow. The bottleneck link is 100 Mbps, the RTT is 23 ms, and the bottleneck queue is 128 packets ($\approx 0.5\times$BDP). Because the Cubic flow prioritizes high throughput at the expense of inducing queuing, it forces the Vegas flows to back off and the Cubic flow consumes the majority of the bandwidth, as shown in Figure 3b. The company could configure its routers to divide the internal and external flows into separate traffic classes served by separate queues, but Section 5 demonstrates that an endpoint-based solution using FlowPolicy can prevent loss-based flows from stealing bandwidth from delay-based flows without requiring network support. Figure 3c shows FlowPolicy, described in Section 5, holding the Cubic flow to a fair rate. Note that FlowPolicy is meant to protect the Vegas flows that traverse the internal network; flows that have bottlenecks on links outside the company's network would need to contend with the incumbent CCA at that bottleneck.

While this example focuses on Cubic and Vegas, the evaluation in Section 6 also considers interactions between Cubic and rate-based BBR, which is used by many Internet services.

2.3 Steps towards cooperative bandwidth allocation

To address these three challenges, we consider alternatives to sender-side rate control that point the way towards a model where multiple network entities combine their policies.

2.3.1 In-network techniques

Researchers have considered in-network bandwidth allocation enforcement directly in the switch queues where congestion arises. These techniques are agnostic to the CCA and application. Switches may enforce prioritization or weighted fairness subject to policies that the network oper-

ator knows about but are invisible to the sender and receiver. The canonical example is fair queuing (FQ) [22, 55], which divides incoming flows into separate queues that are serviced at the same rate. Controls also exist for the purposes of isolation and billing in cellular networks and Internet service provider (ISP) gateways. Deploying such techniques does not eliminate the need for a sender-side CCA, since it is preferable to not waste network resources transmitting packets that will eventually be dropped [27]. Thus, the resulting bandwidth allocation is the union of both the sender's policy and the network operator's policy. We believe that network-level techniques represent an inspiring example of joint control already in practice, and seek to extend this union to include the receiver as well. Our proposal for cooperative control provides similar benefits to in-network techniques: expressive bandwidth allocation neither tied to a particular CCA nor application, nor dependent on sender cooperation.

2.3.2 Receiver-based techniques

Researchers have considered proposals that move the CCA to the receiver. Early work focused on specific scenarios, such as when the bottleneck is at the receiver downlink [66], specifically for web content [32], or at the boundary of rate-controlled networks [46]. More recent proposals such as ExpressPath [19], Homa [54], NDP [35], and pHost [30] focus on datacenter networks and involve fine-grained receiver scheduling to achieve high throughput and low latency, but oftentimes require switch support like priority queues or traffic shaping logic.

A potential strawman for cooperative control would be to take an existing receiver CCA and deploy it in parallel with a sender-side CCA, creating dual congestion control at both endpoints. However, as both sender and receiver CCAs expect complete control over transmissions to perform tasks such as bandwidth probing, the parallel deployment of CCAs is not practical without coordination in their design and implementation. We believe that the right design for end-to-end cooperative control on the Internet is to specifically avoid integrating the objective of cooperative control with that of a specific congestion control algorithm. Instead, we desire a general approach to bandwidth allocation policies that does not depend on the CCA in use.

The Customizable Receiver-driven Allocation of Bandwidth (CRAB) [67] system demonstrates that another mechanism by which the receiver can influence a flow's rate is by using ingress queues to shape traffic. CRAB provides *out-of-band* feedback in the form of induced delay and loss from which the sender must infer congestion to avoid wasting network resources. Cooperative control, on the other hand, leverages an *in-band* technique, TCP flow control, to directly inform the sender of a rate limit, thus causing an instantaneous change without the need for the sender to reconverge. Our in-band approach does not introduce additional latency and does not waste network resources by dropping packets that have already arrived at the receiver.

2.3.3 Datacenters techniques

Many systems discuss rate allocation in datacenters [20, 45, 59, 60, 64, 68], including traffic shaping [61], isolation [50], and congestion control [37, 54]. Some techniques propose receiver-based or receiver-assisted rate control [19, 30, 35, 54, 72]. While we take inspiration from these techniques, we target the Internet, not datacenters, and therefore face different constraints. Datacenters

exhibit different network characteristics — such as orders-of-magnitude lower RTT, higher bandwidth, and lower loss — that call for more reactive algorithms. Datacenters benefit from being a single administrative domain with homogeneous hardware on which the operator can deploy tightly integrated techniques involving *both* endpoints.

3 Design Overview

To address the three challenges in Section 2.2, this section describes cooperative rate control and its implementation in the RateMon platform. First, we explain the design constraints faced by cooperative control and how these differ from CCA design. Second, we describe receiver window tuning, the mechanism by which RateMon influences the sender's rate. Finally, we outline the RateMon architecture, including its pluggable rate control policy.

3.1 Design constraints

Our goal is to implement RateMon at the receiver only without modifying the TCP stack or interfering with the packet stream. Furthermore, RateMon must not alter applications, modify the TCP header's structure, require coordination between receivers, or require that all receivers on the Internet use RateMon or adhere to the same rate policy. Since cooperative control does not integrate with an existing sender or receiver CCA, it faces separate challenges from CCA design. This distinction has two implications. First, TCP senders control the packet stream and probe for bandwidth while maintaining internal state that measures congestion, such as the window size, retransmission counters, and when timeouts expire. The receiver, however, is limited to inferring state from the packet stream without explicit control, or access to the CCA's measurements. Second, the receiver need not produce a rate decision that achieves typical CCA goals like high utilization and low delay. Since the sender's CCA runs at the packet level, the receiver does not need to produce an action for every packet. Instead, it only needs to act when it detects that an incoming flow is violating the receiver's policy. Consequently, this freedom for the receiver to act less frequently reduces compute overheads and allows it to make better estimates before reacting, with the downside of potential error between decisions.

3.2 Leveraging TCP flow control to enable cooperation

As described in Section 2.2.2, mapping application demand onto network resources has historically been the responsibility of the transport layer. For this reason, we believe that the transport layer is the right venue to implement the receiver's rate control policy, as an extension of the abstraction that the transport layer already provides to applications. Building on the design constraints above, this section provides an introduction to receiver window tuning, the simple transport-level technique that RateMon uses to enable cooperative rate control.

3.2.1 Flow control overview

In TCP *flow* control, the receiver advertises in the TCP header a window of space that represents how much memory it has reserved for this connection's incoming packets [1]. This buffer is where it will store out-of-order packets for reassembly and wait for an application to read the data. The receiver makes no promises about what it will do with packets received past its advertised window. The sender calculates its send window as the minimum of its locally-computed congestion window and the receiver's advertised window (RWND). This means that both the CCA *and* flow control limit how much data the sender can have in flight at once. Under normal operating conditions, it is desirable to have congestion control and not flow control constrain a flow's rate.

3.2.2 Receiver window tuning

Receiver window tuning (RWND tuning) is the process by which a controller on the receiver overwrites the advertised window field in the TCP header of outgoing ACK packets to leverage flow control to impose a customized constraint on the sender CCA. Instead of redesigning sender congestion control or adopting a full receiver-side CCA, RWND tuning is a complementary algorithm that allows the receiver to constrain an arbitrary sender CCA into a desired region of its window state space based on the receiver's policy. RWND tuning does not modify the TCP header structure or the sender, and does not impose high overheads on the receiver. RWND tuning also does not require application changes or application/sender buy-in. If a user wants greater control, then they can enable cooperative control in their local policy domain and immediately see benefits with no need for application changes, mass adoption, or coordination across users.

RWND tuning may call for shrinking the advertised window even though a corresponding amount of data has not been received. RFC 9293 [24] states that "a TCP receiver should not shrink the window" from the right in this way, but nevertheless senders "must be robust against window shrinking". In our experiments, the TCP implementation in Linux has not presented any errors due to this behavior. A simple enhancement would be to shrink the advertised window progressively, at the same rate as data arrives.

3.2.3 Existing uses of RWND tuning

RWND tuning is a well known technique for modifying sender behavior. In the datacenter space, incast congestion control for TCP (ICTCP) [72] advocates for tuning the receiver's TCP advertised window to force senders to back off, specifically during incasts where tens of senders transmit to a single receiver. ICTCP adopts a complex, online, and reactive approach that more closely resembles a full receiver CCA, whereas we propose a simple, lightweight approach that sits alongside the existing sender CCA. AC/DC TCP [37] uses RWND tuning to force virtual machines to adhere to a given traffic profile by modifying incoming ACKs to affect the behavior of virtual machines on the same machine. In contrast, we propose modifying outgoing ACKs to affect a remote host's behavior. rLEDBAT [7] is a recent proposal for a less-than-best-effort congestion control algorithm implemented at the receiver that uses RWND tuning to control the sender. This is an example of a specific technique that could be implemented in the RateMon system, aside from those described in this paper.

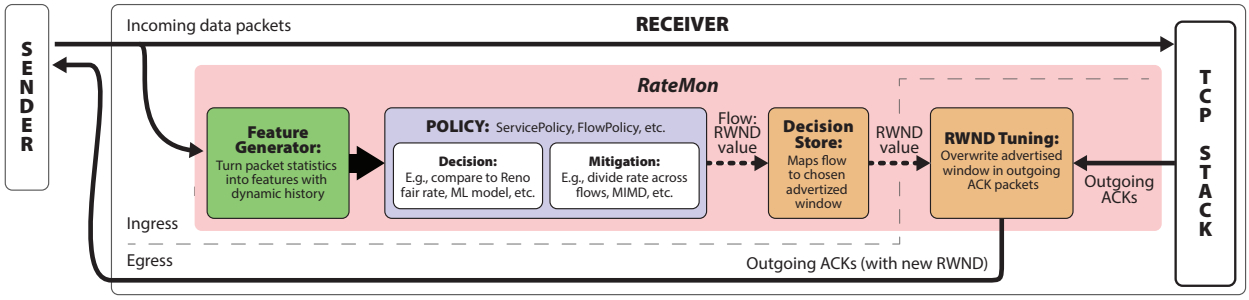


Figure 4: The RateMon platform. Solid lines represent data and dotted lines represent control.

We take inspiration from work by Spring *et al.* [66] on cooperative congestion control for access links that also employs RWND tuning to control senders' rates. However, this prior work differs in both assumptions and objectives. It assumes a tight interaction between the receiver and the bottleneck with the goal of allocating resources between short-lived, interactive, and bulk flows to improve webpage load times. It assumes a single bottleneck with a known rate and divides the BDP (and therefore the rate) equally between flows. Our work instead ensures more broadly that traffic through the network adheres to the receiver's local policies. Our techniques do not assume knowledge of or observability into the bottleneck and, in fact, each flow may have a different bottleneck. Traffic class scheduling is possible in cooperative rate control, but our example policies do not include this.

3.2.4 Limitations of cooperative rate control

Since RWND tuning is a form of negative control, the receiver cannot force the sender to *increase* its rate. This causes a risk of underutilization because the sender must ramp up on its own when given a larger advertised window. Similar to other work [6, 67], we believe that a minor degree of underutilization is acceptable to better address the receiver's objectives, and we evaluate the penalty of this.

Deliberately malicious flows are not the focus of this paper. The sender must obey TCP flow control for RateMon to be effective. The receiver does not control the packet stream, so cooperative rate control cannot prevent intentional misbehavior like a flooding attack. In the presence of a malicious sender, RateMon could escalate to dropping packets or notifying the user. Instead, we focus on bridging the gap between well-intentioned policies on the two endpoints.

If, after a flow is controlled by RWND tuning, its rate is lower than allowed by the receiver's chosen RWND (i.e., the sender has insufficient demand), then overall link utilization may decrease. This could be easily mitigated by extending the receiver to compare the throughput of each flow to its target rate and then redistribute surplus RWND to other flows should a flow be unable to saturate its allocation.

3.3 The RateMon platform

Now that we have described RateMon's underlying RWND tuning technique, this section outlines the system as a whole: its architecture, continuous rate monitoring, and implementation details.

3.3.1 Architecture

RateMon consists of two components: a general, flexible *mechanism* for receiver-assisted rate control based on RWND tuning, and a plugable rate *policy* that specifies how to determine whether a flow is adhering to a desirable rate and, if not, then how to compute a target rate for the flow. RateMon is implemented as a combination of a userspace Python controller and kernel-space eBPF [23] programs that run on a receiver and are separate from the TCP stack. Refer to Appendix A.2 for eBPF details and rationale. Figure 4 gives an architectural overview. Modules are described below.

- **Feature Generator:** RateMon generates a copy of incoming data packets using `libpcap` [31]. For each flow separately, a generic feature generator processes packets to build a rich set of statistics that the policy can use to make decisions (Section 5.2.2). Examples include the loss event rate [36], average RTT, and packet interarrival time. Policies specify which features they need, and only these are calculated. Some features, such as average throughput, take several RTTs to calculate, so there can be a delay between when a flow starts and when the feature generator is ready.

- **Policy:** The core of RateMon is a pluggable receiver rate policy that detects whether a flow, or set of flows, should adjust its rate. This consists of two components, a *decision* module that processes features from the feature generator to determine policy compliance, and a *mitigation* module that uses this decision to determine a new rate for a flow. The per-flow rate is then multiplied by the RTT to create a window size for use in RWND tuning. Both the decision and mitigation modules are policy-specific. Examples of local policies include ServicePolicy (Section 4) and FlowPolicy (Section 5). In most cases, policy configuration is done in the same way that a user "chooses" a CCA: it is either the default in their system, chosen by their administrator/organization, or manually overridden.

- **Decision Store:** The output of a policy's mitigation module is one or more mappings from a flow four-tuple to a custom RWND value. The policy encodes these mappings into the decision store, which is an eBPF map shared between the userspace policy module and the kernel-space RWND tuning module.

- **RWND Tuning:** This module is an eBPF program running in kernel-space. At ACK egress time, the RWND tuning module looks up a flow's custom RWND value in the decision store and places it into outgoing packets. See Appendix A.2 for implementation details.

3.3.2 Long-term rate monitoring

RateMon's process of feature generation, policy evaluation, and mitigation described above occurs continuously over a flow's life at a configurable cadence. Our prototype reevaluates the policy

every 100 ms. A policy's mitigation module describes how to evolve a flow's rate over time as the network conditions change, such as due to background flows starting and ending. Sections 4 & 5 describe the unique mitigation approaches taken by ServicePolicy and FlowPolicy, respectively.

4 ServicePolicy

In the next two sections, we describe two example RateMon policies that address the challenges arising from differing endpoint policies on rate allocation (1) granularity, (2) algorithm, and (3) optimization metric. We begin with ServicePolicy, which enforces a receiver objective of rate allocation at the granularity of services in an environment where CCAs provide only per-flow fairness. We explain ServicePolicy first because it illustrates the capabilities of cooperative rate control through a well-understood context (services using concurrent flows) and technique (Mathis equation for Reno throughput [51]). This section defines ServicePolicy's objectives, describes its implementation in RateMon, and revisits Example 1 (Figure 1) to demonstrate its performance.

4.1 ServicePolicy definition

Section 2.2.1 describes how applications use concurrent flows for many reasons, but that this behavior leads to services with higher flow counts claiming a larger share of the bandwidth. ServicePolicy decrees that if a service uses multiple flows, then together those flows should approximate the behavior of a single TCP Reno flow. Note that this is different from enforcing fair queuing between services, which would hold services that share a bottleneck to equal rates. Instead, ServicePolicy does not assume that services share a bottleneck; each service is independently held to the throughout it would achieve if it were using a single flow implementing Reno's congestion avoidance algorithm [51] over an arbitrary bottleneck. In our prototype, we consider flows to belong to the same service if they originate at the same sender host. With minor modifications, flows could easily be grouped into services based on IP prefix, TCP port number, or manual filters.

4.2 ServicePolicy implementation

ServicePolicy must estimate the target rate of a service using information gleaned only from arriving packets. To accomplish this, it uses the Mathis equation for Reno throughput [51], which specifies the rate that a well-behaved Reno flow should achieve given a certain RTT, loss rate (p), and maximum segment size (MSS): throughput $= (MSS \times C)/(RTT \times \sqrt{p})$, where C is a constant factor. The original Mathis equation is based on the pure loss rate, but instead we borrowed *loss event rate* from TCP Friendly Rate Control (TFRC) [36] to improve robustness under correlated losses. We make further small modifications to this equation based on measurements in our testbed.

From the RateMon feature generator, ServicePolicy requests for each flow the average RTT over the past 8 RTTs and the loss event rate. These features are calculated for each flow, so ServicePolicy must combine them to produce service-level features, as follows. Across flows, it computes the average RTT. For the loss event rate, each flow provides samples of the same

underlying loss distribution. To avoid overestimating the loss rate, the system must choose unique loss events across all flows. ServicePolicy could attempt to deduplicate loss events using a voting-based scheme, but to avoid overestimation, we found that using the minimum per-flow loss event rate performs well. Note that these aggregation techniques assume that flows *for a single service* share a bottleneck. This is similar to other solutions to service-level sharing [10, 58, 70]. This assumption may not hold in all situations, but could be relaxed by implementing false sharing detection [4]. This implementation still allows separate services to have separate bottlenecks.

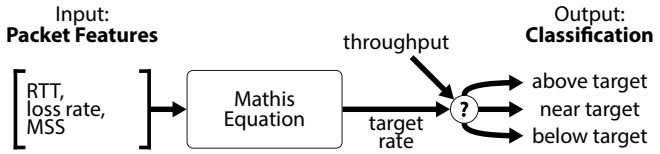
In ServicePolicy, RateMon manages all services, even those with a single flow, for their flows' entire lifetimes. ServicePolicy periodically re-evaluates the Mathis target rate to adapt to changes in RTT, loss event rate, and flow count. Once ServicePolicy estimates the target rate for a service, this total rate must then be divided between the service's flows. ServicePolicy could apply a service-specific algorithm such as providing per-flow fairness within a service, allocate proportionally higher rates to flows with greater demand, or prioritize those that serve certain content. For our prototype, the total rate is divided equally between the service's flows. Each per-flow rate is converted to a window size by multiplying by the estimated minimum RTT. RWND tuning communicates these new window limits to the senders. The result is that the receiver's services are forced to mimic a set of well-behaved Reno flows with well-understood fairness dynamics. Services already transmitting below their target rate are not effected by RWND tuning.

4.3 ServicePolicy demonstration

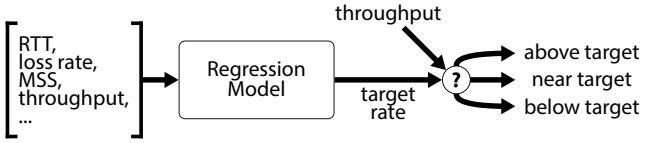
To show ServicePolicy in action, we revisit Example 1 from Section 2.2.1. Later, Section 6 evaluates ServicePolicy on a broader range of scenarios. In this example, a client accesses a video streaming service that uses a single flow and a file sharing service that uses nine flows. Figure 1b shows that without ServicePolicy, the nine file sharing flows claims the majority of the bandwidth. Figure 1c illustrates the utility of cooperative rate control to enable a common receiver policy of service-level fairness within a policy domain. With ServicePolicy running, after a period of gathering features, both flows are guided to their target Reno rates. When the second service begins, its nine flows must share the allocation of a single Reno flow, and for the remainder of the experiment, each service obtains approximately 50% of the bandwidth.

5 FlowPolicy

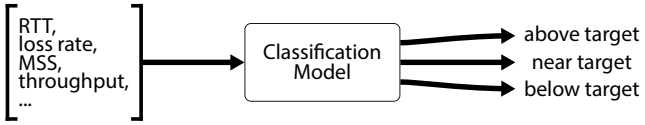
ServicePolicy shows enforcement of different rate policy granularities: per-flow control by services and per-service control by clients. In this section, we discuss FlowPolicy, which is useful when both endpoints focus on per-flow granularity. Within the scope of per-flow rate allocation, FlowPolicy focuses on differences in allocation algorithm and optimization metric, as introduced in Challenges 2 and 3, respectively. Our approaches to these challenges follow a similar design, so for simplicity we focus on algorithmic differences through the design discussion and return to differing optimization metrics afterwards. As FlowPolicy must support the behavior of a wide range of TCP variants, we cannot rely on the Mathis equation, which models TCP Reno behavior. To address this issue, FlowPolicy supports more expressive rate allocation policies by relying on



(a) Example decision architecture based on the Mathis equation.



(b) Example decision architecture based on a regression model. The input feature set expands to include any receiver-observable statistics. The equation for target rate is replaced with a black-box ML model.



(c) FlowPolicy decision architecture based on a classification model. The rate comparison is incorporated into the black-box ML model.

Figure 5: Comparing decision architectures.

a machine learning (ML) model — which can be retrained as the environment or policy changes — instead of a fixed equation. This section defines FlowPolicy's objectives, explains in detail its underlying ML model, and revisits Examples 2 and 3 to demonstrate its effectiveness.

5.1 FlowPolicy definition

FlowPolicy decrees that an individual flow should not exceed its target rate allocation over its bottleneck link. The definition of target rate depends on the receiver's local policy. For demonstration purposes, FlowPolicy strives for throughput equality: for a bottleneck link with bandwidth B and N flows, each flow should achieve B/N throughput. However, because FlowPolicy does not observe the bottleneck or all of the flows that traverse it, it knows the values of neither B nor N.

Therefore, each flow's target rate according to the receiver's policy is unknown. Associated with this unknown rate is a set of network state that can be used to determine the rate. To understand this intuition, consider ServicePolicy as an example. The Mathis equation defines the target rate for a Reno flow without observing the bottleneck or other flows. The network state includes the RTT, loss rate, and MSS, which form an equation to determine the target rate. The equation was derived by careful study of Reno's dynamics. This is visualized in Figure 5a. In the case of FlowPolicy, the network state includes any information learnable from arriving packets and the equation for the target rate is a black-box ML model. FlowPolicy trains this model in an emulator where B and N are known.

In ServicePolicy, the target rate is directly converted to a window size and divided between a service's flows. FlowPolicy takes a different approach, as shown in Figure 5, and instead compares the target rate to the actual rate to determine whether a flow is above, near, or below its target. These

three classes are defined formally below. A strawman design would be to use a *regression* model to calculate the target rate (Figure 5b). Since the desired output of FlowPolicy's decision module is a classification of above, near, or below the target rate, FlowPolicy instead uses a *classification* model that directly produces the class label without the intermediate rate comparison. This final architecture is shown in Figure 5c. In a process described in Section 5.3, FlowPolicy periodically re-evaluates the model and adjusts RWND tuning to guide a flow to converge to near its target rate.

Formally, FlowPolicy defines policy adherence as a multi-class classification task with three classes. For a flow with throughput T, its class is defined as:

$$\text{class} = \begin{cases} \text{above target} & \text{if } T \geq 1.1 \cdot B/N \\ \text{near target} & \text{if } 0.9 \cdot B/N \leq T < 1.1 \cdot B/N \\ \text{below target} & \text{if } T < 0.9 \cdot B/N \end{cases} \tag{1}$$

The 10% tolerance around "near target" is necessary because flows naturally oscillate around the target rate. T is calculated as the average throughput over the past 8 minimum RTTs.

Note that FlowPolicy does not compare flows to each other, but rather evaluates each flow individually against the model's learned notion of target rate. FlowPolicy does not assume that all flows share a bottleneck, that the bottleneck is at the last hop to the receiver or the ISP gateway, or that the receiver observes all flows over the bottleneck. This is a key difference from CRAB and FQ techniques (both in-network FQ and receiver FQ). FlowPolicy is designed for user devices on the Internet where bandwidth contracts are on the order of 100 Mbps, so the receiver has sufficient cycles per packet to run a simple model.

5.2 FlowPolicy implementation

This section describes the design of FlowPolicy's ML model for classifying whether a flow is above, near, or below its target rate. We describe the network emulator used for training, the model's input features that crucially capture flow history, and the model itself. Referring to Section 3.3, the logic described here comprises FlowPolicy's *decision* module.

5.2.1 Offline training in a network emulator

We train FlowPolicy's model in an *offline* network emulator for two reasons: (1) to gain access to ground truth information (the values of B and N) by observing all flows traversing the bottleneck link, and (2) to avoid the runtime performance overhead of training in an online manner. We use an emulator instead of a simulator to achieve realistic timing behavior and to use real CCA implementations. The output of the emulator is a receiver packet trace.

The emulator, shown in Figure 6, creates a dumbbell topology from three hosts. One host uses the Berkeley Extensible Software Switch (BESS) [34] to emulate the bottleneck queue, its associated bandwidth B, and the round trip delay, for both the data and the ACK directions. The other two hosts emulate the senders and receiver and communicate via the bottleneck host. Table 1 lists the network parameters used to generate training data. Each experiment first launches M flows using an incumbent CCA J. These incumbent flows emulate the desired rate allocation behavior

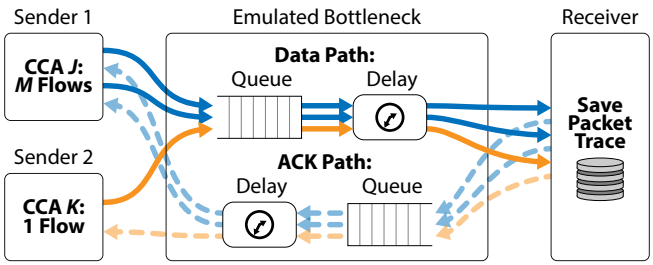


Figure 6: The emulator's topology. Solid lines are data packets and dotted lines are ACK packets. Blue lines are incumbents and orange lines are newcomers. Each delay module emulates half of the RTT.

Table 1: Parameters for training and evaluation.

Parameter	Values
Bottleneck bandwidth (B)	1, 4–100 Mbps in steps of 4
Bottleneck queue size	0.5, 1–4 $\times$ BDP in steps of 1
RTT	20–200 ms in steps of 4
CCA J (incumbent) flows (M)	1–10 in steps of 1
CCA K (newcomer) flows	1
Duration	100 seconds

for the receiver's policy. After 20 seconds, one newcomer flow using CCA K begins. Therefore, $N = M + 1$. The newcomer represents a new algorithm that is expected to adhere to the receiver's policy when interacting with the incumbents. We typically refer to such a configuration as J *vs.* K, e.g., Reno vs. Cubic where Reno is the incumbent and Cubic is the newcomer. All flows are given the same emulated RTT; we do not model RTT unfairness because we focus specifically on behavior inherent to the CCA itself and leave other variables to future work.

We train the FlowPolicy model on combinations of CCA pairs. Returning to Challenge 2, to address the differences in rate control algorithms between Reno and Cubic, we train a FlowPolicy model with Reno vs. Cubic. For Challenge 3, to address the differences in rate control optimization metric between Vegas and Cubic, we train a FlowPolicy model with Vegas vs. Cubic. Furthermore, to show the generalizability of FlowPolicy to broad classes of algorithms and optimization metrics, we train a third model with a combination of Cubic vs. BBR and Cubic vs. Vegas. This third model represents a loss-based incumbent (Cubic) interacting with both rate-based (BBR) and delay-based (Vegas) newcomers.

5.2.2 Feature engineering

In a machine learning task, crafting the input data, i.e., features, that the model operates on is as important as the model itself. FlowPolicy takes a receiver packet trace from the emulator, iterates through the trace, and calculates features for every packet. Examples include packet interarrival time, packet size, loss rate, RTT, and average throughput. The full feature set is described in

Appendix A.1, Table 4.

Dynamic history Deciding if a flow adheres to a policy inherently involves the history of the flow. It is tempting to approach this as a sequence-based prediction task and apply a type of model that incorporates history into its structure, such as a recurrent neural network or long short-term memory (LSTM) network [38]. However, such models are complex to design and train and have high runtime compute requirements. Instead, FlowPolicy uses a technique called *dynamic history* to incorporate history into the features themselves. In addition to basic features like RTT and throughput, FlowPolicy also calculates exponentially weighted moving averages (EWMAs) with multiple decay rates and simple averages over sliding windows of multiple durations. The window duration is a multiple of the RTT, since congestion control behavior evolves on a timescale dependent on the RTT. Refer to Appendix A.1, Table 4 for a list of features that use dynamic history. Appendix A.1, Table 5 lists the dynamic history parameters for these features. The outcome is that FlowPolicy uses multiple versions of each feature, with different amounts of look-back.

Accumulating features with dynamic history takes time, from 1–32 RTTs. Therefore, at the start of a flow, there is a delay before FlowPolicy makes a classification. Consequently, short flows may complete before they are classified. As discussed in Section 3.1, the sender handles fine-grained dynamics, so the receiver has the flexibility to build a view of the sender CCA's behavior across multiple RTTs. Future work could place new flows into a provisional class until the model has sufficient history for a complete decision.

Feature selection to improve performance Including all of the dynamic history features, FlowPolicy uses 209 features in total for each packet. However, many dynamic history features are intuitively redundant because they encode similar data at different time granularities. Calculating these features imposes a high CPU and memory overhead at runtime. To reduce this overhead, FlowPolicy selects a subset of features that are the most informative to the model. Selecting an appropriate subset of features, a process known as *feature selection* [53], yields a model with similar accuracy but many fewer parameters and a shorter runtime.

In FlowPolicy, the feature selection process contains five steps. 1) The model is trained with the complete set of 209 features. 2) The features are clustered according to their Spearman correlation [49], then the clusters are arranged in a hierarchy and a cut of this hierarchy is chosen that yields the desired number of clusters, F [63]. 3) A total order of features is created based on permutation importance [14]. 4) From each cluster, the most important feature is selected according to this total ordering, leaving F features. 5) Finally, the model is retrained from scratch using only the F selected features. We choose $F = 20$, reducing the number of features by an order of magnitude. We measure that the accuracy of the model trained with a set of $F = 20$ features is within 2% of the accuracy of the full model trained with all 209 features, which is an acceptable penalty given the order of magnitude reduction in CPU and memory overheads. Each combination of incumbent and newcomer CCAs yields a unique set of selected features that capture metrics important to a rate policy based on those CCAs. Appendix A.1.2 explores how the selected features change based on the CCAs in the training data.

5.2.3 Model design

Now that we have discussed the model formulation, training emulator, and features, we describe the model itself. Many types of ML model may be appropriate for a multi-class classification task: linear classifiers, deep convolutional neural networks, recurrent neural networks, decision trees, etc. FlowPolicy uses a *decision tree* (DT), for three reasons: (1) DTs are more expressive than a linear model, (2) they are simpler to train than a neural network (fewer design choices and hyperparameters), and (3) they have lower runtime overheads than neural networks. There are likely reasonable designs to be found with these other model types, but in our experience, DTs proved easy to use and sufficiently accurate to demonstrate the benefits of cooperative control.

FlowPolicy uses a subtype of DT called a *histogram-based gradient boosting decision tree* [47]. Breaking down that jargon, "gradient boosting" is a technique that improves the accuracy of a model by training another model to predict the error produced by the first model [28]. This continues iteratively, constructing a chain of models that each fills in a piece of the state space that was missed by the earlier models. Gradient boosting is a common technique to build a complex model from a collection of weak learners. "Histogram-based" refers to a performance optimization by which each input feature, instead of being represented as a continuous value, is first discretized into H buckets, which significantly reduces training time for datasets with many samples (FlowPolicy uses $H = 255$) [47].

Recent work has shown that learning-based CCAs struggle to generalize to scenarios beyond their training data [65]. While FlowPolicy is not a CCA, it likely faces similar limitations. Section 6.2 shows how the FlowPolicy model generalizes to unseen CCAs, unseen network conditions drawn from the same distribution as the training data, and application-limited flows. We leave an evaluation on out-of-distribution network conditions to future work. We believe that the selection of parameters in Table 1 covers a sufficiently broad cross-section of home and mobile network conditions to demonstrate FlowPolicy's capabilities. Refer to Appendix A.3 for additional training details and parameters.

Sage [73] is a recent CCA that uses reinforcement learning to train a neural network that outperforms existing hand-tuned CCAs. Sage uses a similar feature set as RateMon and faces many similar design and training challenges. Sage demonstrates the effectiveness of learning-based techniques in detecting and overcoming the limitations of existing designs. However, Sage is a new CCA itself whereas our contribution is the coexistence of policy controllers at *both* endpoints. Sage's model could likely be reproduced within RateMon at the receiver with minor modifications.

5.3 FlowPolicy mitigation

Now that we have described how FlowPolicy decides whether an incoming flow is adhering to the receiver's rate policy, we explain the actions that the receiver takes in response to this decision — i.e., referring to Section 3.3, this section describes FlowPolicy's *mitigation* module. Succinctly, FlowPolicy continuously evaluates the decision model and adjusts a flow's rate either higher or lower until it converges to near target. This continual adjustment is necessary to avoid overreactions and to adapt to changes in the number of flows over the bottleneck link. If a flow is ever

classified as above target, then it will be cooperatively controlled for its entire lifetime. Flows that are always classified as near or below target are never controlled by FlowPolicy.

FlowPolicy starts by calculating the flow's current throughput T, and then uses a multiplicative increase, multiplicative decrease (MIMD) controller to adjust this throughput over time. We choose MIMD to react rapidly; note that we do not require other CCA properties from this adjustment. A new throughput T' is defined as:

$$T' = \begin{cases} 0.57 \cdot T & \text{if class == above target} \\ T & \text{if class == near target} \\ 1.3 \cdot T & \text{if class == below target} \end{cases} \quad (2)$$

The coefficients were chosen through manual experimentation with the tradeoff between reactivity and underutilization. T' is converted to an advertised window by multiplying by the RTT and is applied via RWND tuning. Note that because RWND tuning is a form of negative control, the receiver cannot explicitly force the sender to increase its rate; if the receiver provides the sender with a larger advertised window, then the sender must ramp up on its own.

5.4 FlowPolicy demonstration

To show FlowPolicy in action, we revisit Challenges 2 and 3 from Section 2.2. Later, Section 6 evaluates FlowPolicy on a broader range of scenarios.

5.4.1 Revisiting Challenge 2 — Differences in algorithm

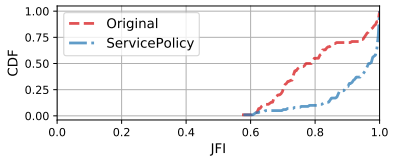
We revisit Example 2 in Figure 2. Three incumbent Reno flows start first, followed by a newcomer Cubic flow, over a 93 Mbps bottleneck. Figure 2c shows FlowPolicy running, using a version of its model trained with Reno as the incumbent and Cubic as the newcomer. After building features (between time = 30–35 seconds), it classifies the Cubic flow as above target and for the remainder of the experiment, the four flows oscillate around 20 Mbps each.

5.4.2 Revisiting Challenge 3 — Differences in optimization metric

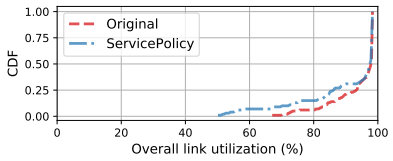
We revisit Example 3 in Figure 3. Three incumbent delay-based Vegas flows start first, followed by a loss-based newcomer Cubic flow, over a 100 Mbps bottleneck. Figure 3c shows FlowPolicy running, using a version of its model trained with Vegas as the incumbent and Cubic as the newcomer. FlowPolicy reacts quicker here, likely because the time to build dynamic history features depends on the RTT, and the RTT is shorter when using delay-based CCAs. FlowPolicy holds the four flows to approximately 25 Mbps each.

6 Evaluation

This section evaluates cooperative rate control more broadly than the three specific scenarios described so far. First, we show how ServicePolicy and FlowPolicy generalize beyond earlier specific

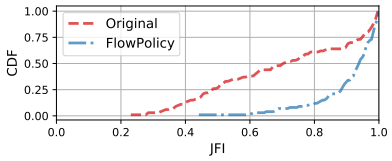


(a) CDF of service-level JFI.

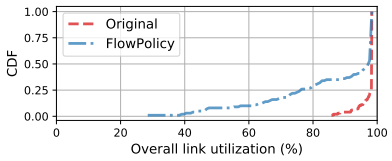


(b) CDF of average utilization.

Figure 7: Challenge 1: Across 100 scenarios, ServicePolicy improves adherence to the receiver's policy for rate allocation at the service granularity.

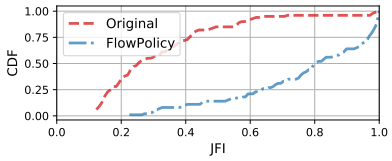


(a) CDF of per-flow JFI.

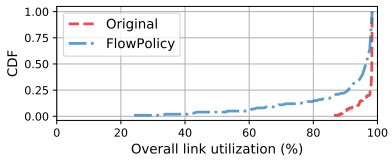


(b) CDF of average utilization.

Figure 8: Challenge 2: Across 100 scenarios, FlowPolicy (Reno vs. Cubic) improves adherence to the receiver's policy for rate control algorithm.



(a) CDF of per-flow JFI.



(b) CDF of average utilization.

Figure 9: Challenge 3: Across 100 scenarios, FlowPolicy (Vegas vs. Cubic) improves adherence to the receiver's policy for optimization metric.

examples and are effective for a broad range of configurations in each of the three challenges. Second, we explore FlowPolicy in depth to show the robustness and generalizability of its model architecture. Finally, we present accuracy and performance microbenchmarks.

In the following results, we run many experiments with different configurations of bandwidth, RTT, queue size, and flow count chosen uniformly at random from Table 1, but avoid any used in FlowPolicy model training, meaning that every network state tested here is a situation that the FlowPolicy model *has not seen before*. We report Jain's fairness index (JFI) [44], which produces a score in the range [0, 1] where 1 represents equal sharing. For ServicePolicy, we calculate JFI between services and for FlowPolicy we calculate JFI between flows. We calculate JFI using the average throughput of each flow over the entire experiment, so each configuration produces one JFI sample in the following distributions. We also evaluate link utilization to show that deploying cooperative control, which reduces sender rates, does not cause a disproportionate decrease in overall utilization.

6.1 Broad evaluation of the three challenges

The goal of this section is to show that cooperative rate control addresses a broad set of network configurations in each category of the challenges discussed earlier.

6.1.1 Challenge 1

Figure 7 evaluates ServicePolicy on 100 random configurations of bandwidth, RTT, queue size, and flow count. Figure 7a shows that ServicePolicy improves the median JFI when calculated at the service granularity by 23.9%, a significant improvement. The low JFI shown in the "Original"

lines is, in effect, a byproduct of the CCAs striving for high bandwidth. By improving the JFI significantly, ServicePolicy forces some flows to back off before they use all available bandwidth, resulting in a slight utilization penalty: the average link utilization at the 10^{th} percentile decreases by 11.4%. Figure 7b shows that penalties at higher percentiles are less, with no impact at the median. This penalty is acceptable because ServicePolicy focuses primarily on rate allocation and does not specifically optimize for high utilization. Many systems argue that a small utilization penalty is acceptable to improve other metrics like low delay [6] or policy compliance [67].

6.1.2 Challenge 2

Figure 8 evaluates FlowPolicy, trained with Reno as the incumbent and Cubic as the newcomer, on 100 random configurations. Figure 8a shows that FlowPolicy improves the median per-flow JFI by 32.7%. Figure 8b shows the utilization penalty, which is higher than for Challenges 1 and 3. On further investigation, FlowPolicy is successful at restraining the newcomer Cubic flows to their target rate, but sometimes identifies unfairness in the Reno flows and restrains them as well.

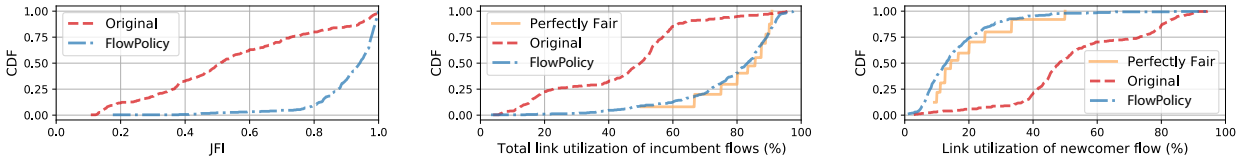
Each FlowPolicy model is sensitive to specific hyperparameters. For the bulk of the evaluation, we choose parameters that work well for all situations, but are likely not optimal for any. Consider the following example. To reduce the utilization penalty shown above at the expense of worse fairness, we hand-tuned FlowPolicy's MIMD coefficients to back off less quickly: we increased the rate reduction coefficient if a flow is above target from $0.57\times$ to $0.8\times$. This cost 1.6% in median accuracy, but improved 10^{th} percentile utilization by 14.8%, yielding the results shown here. Additional tuning would likely improve utilization further, and other models would likely benefit from their own fine-tuning.

6.1.3 Challenge 3

Figure 9 evaluates FlowPolicy, trained with Vegas as the incumbent and Cubic as the newcomer, on 100 random configurations. Figure 9a shows that FlowPolicy improves the median per-flow JFI by 220.8%. Figure 9b shows the utilization penalty, which is notable for only 25% of configurations. Note that since incumbent Vegas optimizes for short queues, it inherently invites the risk of underutilization, as shown by the "Original" line's tail below the 25^{th} percentile. Constraining the Cubic flow serves to increase the likelihood that the Vegas flows cannot fill the bottleneck.

6.2 Robustness and generalizability

The previous FlowPolicy examples focus in isolation on endpoint policy differences regarding either algorithm (Reno vs. Cubic) or optimization metric (Vegas vs. Cubic). However, in any real-world deployment, it is unlikely that the system will know what newcomer CCAs to expect. It is useful for a FlowPolicy model to be trained for a particular incumbent CCA policy and to support a wide variety of newcomer algorithms. To these ends, we present a model that enforces Cubic-style rate allocation on the receiver's flows and is trained on three newcomers: BBR, Vegas, and other Cubic flows. Together, these three CCAs cover a broad range of rate allocation behavior. In summary, we show:



(a) CDF of per-flow JFI. (b) CDF of total incumbent util. (c) CDF of total newcomer util.

Figure 10: FlowPolicy constrains a rate-based newcomer (BBR) to prevent harm to loss-based incumbents (Cubic).

1. **Supports complex metrics:** FlowPolicy improves flow-level fairness in the presence of rate-based BBR flows by up to 85.1% with an acceptable 9.7% utilization penalty.

2. **Benefits for all CCAs in Linux:** FlowPolicy improves the JFI for all CCAs in Linux, despite being trained on only a subset thereof.

3. **Benefits for flows over the real Internet:** FlowPolicy's model, despite training in an emulator, supports flows across the Internet, increasing median JFI by 102.9%.

4. **Generalizes to application-limited flows:** Despite training on infinite-demand flows, FlowPolicy does not overreact in the presence of application-limited flows.

5. **Partial deployments:** To enforce a rate policy over a network domain, as in Example 3, FlowPolicy need only be deployed on hosts that may receive offending flows.

6.2.1 Complex metric — BBR

BBR [17] is an algorithm that was unveiled by Google in 2016 and is the default CCA for YouTube. In just three years, BBR grew to be responsible for 11% of Internet traffic [71]. In this work, we consider BBRv1. Despite careful design and testing, BBR variants can claim a much higher bandwidth fraction than existing loss-based CCAs like Reno and Cubic [71]. BBR is a rate-based CCA that estimates the BDP by both probing for more bandwidth and backing off to drain queues and measure the minimum RTT. These dynamics combine the bandwidth probing seen in loss-based CCAs with the latency sensitivity of delay-based CCAs. However, BBR proves to be more bandwidth-hungry than either approach: BBR's multiplicative bandwidth probing and ambivalence to loss cause it to incrementally claim bandwidth from loss-based CCAs [71].

Figure 10 shows results for various numbers of Cubic incumbents interacting with a BBR newcomer in 300 random network conditions. Figure 10a shows a CDF of the flow-level JFI. BBR is known to take more bandwidth than loss-based CCAs like Cubic, as shown by the uniform distribution of JFI in the "Original" line. FlowPolicy significantly improves the median JFI from 0.51 to 0.94, an increase of 85.1%.

Figure 10b shows a CDF of the combined average utilization of only the incumbent Cubic flows. Without FlowPolicy, the Cubic flows achieve much lower utilization than their fair share (the "Perfectly Fair" line). With FlowPolicy, their median utilization improves by 33.9% and is

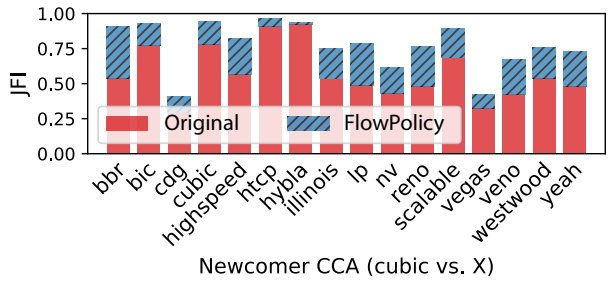
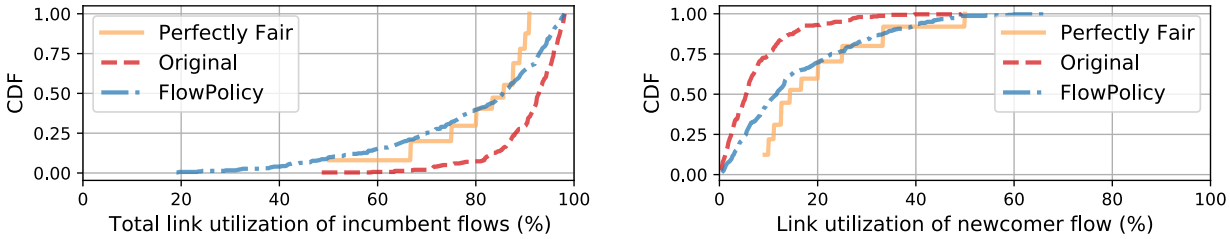


Figure 11: When competing with Cubic, all CCAs in Linux see a JFI improvement when using FlowPolicy.



(a) CDF of total incumbent utilization (Cubic).

(b) CDF of total newcomer utilization (HighSpeed).

Figure 12: A FlowPolicy model can improve fairness even when a CCA (e.g., HighSpeed) was not part of the training data.

only 2.3% below perfectly fair. Figure 10c shows that without FlowPolicy, the BBR flow achieves significantly higher utilization (median 48.4%) than its fair share (median 14.3%) by stealing bandwidth from the Cubic flows. FlowPolicy forces the BBR flow to back off, reducing its median utilization to 12.9%.

6.2.2 All CCAs in Linux

Next, we consider cases where the newcomer CCA is completely foreign to the model. This demonstrates the ability of cooperative rate control to evolve over time as the set of CCAs on the Internet changes. Performing well on unseen CCAs is crucial to practicality on the Internet because services use a wide and ever-changing range of TCP variants that are not available for training. This experiment uses the same FlowPolicy model as above on 100 random configurations. Figure 11 visualizes Cubic competing with all CCAs installed on our Linux test hosts (minus DCTCP [5], since it is not designed for the Internet). Only Cubic, BBR, and Vegas are known to the model. In this experiment, a single incumbent Cubic flow compete with M newcomer flows. In all cases, FlowPolicy improves the JFI regardless of whether the CCA was represented in the training data. Across all configurations, the utilization penalty at the 10^{th} percentile and median are 6.0% and 0.95%, respectively.

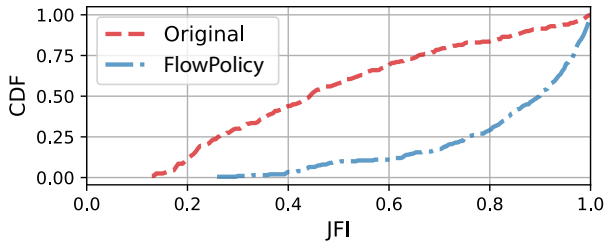


Figure 13: FlowPolicy, despite being trained in an emulator, significantly improves the JFI for Cubic vs. BBR even when flows traverse the Internet.

Unseen Highspeed Next, to show FlowPolicy's generalizability we look in detail at a newcomer that is not present in the training data. We consider a newcomer flow using TCP Highspeed [25], a modification of AIMD that adapts to high bandwidth-delay product networks by scaling the response function as the congestion window grows. Here, M Cubic flows compete with 1 HighSpeed flow. Figure 12a shows that, similar to Vegas, HighSpeed claims less bandwidth than Cubic. FlowPolicy is effective in rate-limiting the Cubic flows to allow HighSpeed to ramp up, reducing their median utilization from 92.7% to 85.5%, which is below the target of median 88.9%. As shown in Figure 12b, HighSpeed's median utilization increases to 2.1$\times$ its value when not constrained, and largely tracks the target fair share.

6.2.3 Flows over the real Internet

Next, we demonstrate that the FlowPolicy model, trained in an emulator, generalizes to flows over the Internet, where there is much greater congestion complexity. This experiment uses the same FlowPolicy model as above, but now the sender is in a different CloudLab region from the emulated bottleneck and receiver (i.e., flows traverse the Internet). Figure 13 shows the CDF of JFI across 400 configurations. Similar to Figure 10, these experiments consider Cubic vs. BBR. The median JFI improvement is actually more pronounced than in the emulator — 102.9% compared to 85.1% — but the distribution has a longer tail, likely due to Internet dynamics interfering with some features, and the improvement is not as significant at other percentiles.

6.2.4 Application-limited flows

FlowPolicy, which was trained on bulk downloads with infinite demand, generalizes to scenarios where flows are limited by the application sending rate (i.e., *app-limited*), as in video bitrate adaptation. We configure an experiment with two senders across the Internet from the receiver and evaluate 100 configuration from Table 1 of Cubic vs. BBR on three scenarios and show that FlowPolicy responds reasonably.

1. When all flows are app-limited to less than their fair shares (i.e., the total demand is less than the bandwidth), FlowPolicy's policy is unnecessary because all demand is satisfied. We observe that FlowPolicy correctly does not react, leaving the JFI and utilization distributions unchanged.

Table 2: Analysis of partial deployments of FlowPolicy.

Which receivers use FlowPolicy	Median JFI	10^{th} %-tile util. (%)
Neither	0.68	95.9
Rec. 1: Incumbents only	0.66	98.0
Rec. 2: Incumbents & newcomer	0.84	97.6
Both	0.87	97.3

2. When the newcomer BBR flow is app-limited to less than its fair share, e.g., as in low-quality YouTube traffic, the median JFI is higher than in Section 6.2.1 at 0.93 because the BBR flow is now tame and the fairer Cubic flows dominate. FlowPolicy reacts occasionally: it does not change the JFI distribution (already good) but sees a small median utilization penalty of 4.3%.

3. When the incumbent Cubic flows are all app-limited to less than their fair shares, e.g., as in a self-throttled file sharing service downloading many objects, the BBR flow seeks more bandwidth and the median JFI is consequently lower, at 0.40. FlowPolicy helps address this greater unfairness and improves the median JFI to 0.57 (improvement of 44.5%), but this is not as significant as the improvement when flows have infinite demand (Section 6.2.1). Median utilization drops by 5.4%.

6.2.5 Partial deployments

Cooperative rate control is useful both to enforce a single receiver's rate policy, as shown in Example 1, but also to enforce a policy that applies to a network domain, like the company in Example 3 that chooses to deploy a delay-based CCA on all of its hosts. In this case, a natural question that arises is whether all hosts must deploy FlowPolicy for it to be effective in enforcing a domain's policy. FlowPolicy is amenable to partial deployments: only hosts that may encounter flows that do not comply with the network's policy must deploy FlowPolicy.

To illustrate this, we conduct an experiment where one sender transmits to two receivers that share a bottleneck link. The sender transmits Cubic flows (the incumbents) to both receivers, but also sends a BBR flow (the newcomer) to the second receiver. Table 2 measures the per-flow JFI and utilization over the bottleneck link depending on whether neither, either one, or both of the receivers deploys FlowPolicy. As expected, deploying FlowPolicy on neither receiver or only the first receiver (with only Cubic flows) results in a low JFI. However, deploying FlowPolicy on only the second receiver (with the BBR flow) is practically as effective as deploying it on both. Mapping this to Example 3, it is sufficient to deploy FlowPolicy only on hosts that may receive traffic that does not comply with the receiver's policy — in this case, inbound traffic from the Internet. Hosts that only interact with compliant traffic do not require FlowPolicy.

Table 3: FlowPolicy model accuracy on test data.

Training data	Model accuracy (%)
Reno vs. Cubic	82.6
Vegas vs. Cubic	94.1
Cubic vs. {BBR, Vegas}	74.8

6.3 Microbenchmarks

This section explores FlowPolicy's model accuracy and runtime performance in detail.

6.3.1 Model test accuracy

We analyze the accuracy of FlowPolicy's histogram-based gradient-boosted decision tree and dynamic history feature selection process in isolation, on its test data, to gain insights into the learnability of fairness for different CCA pairs. Table 3 reports the accuracy for a variety of data compositions. Model accuracy is proportional to the difficulty of the classification task. Cubic claims more bandwidth than Vegas in most scenarios, so modeling their interaction is comparatively easy (94.1% accuracy). Alternatively, Reno and Cubic are both loss-based with related algorithms, so rooting out unfairness there is more challenging (82.6% accuracy). The combined model deals with three optimization metrics, and its accuracy is the lowest at 74.8%. However, this model generalizes well to other CCAs, as shown earlier.

6.3.2 Runtime performance

Our implementation does not optimize performance (RateMon in written in Python), and instead serves as an example of the types of policies that are possible with cooperative rate control. The main performance bottlenecks are generating features and running model inference, which are each given one CPU core. Feature generation time depends on the longest dynamic history sliding window, which in our model is 32 minimum RTTs. Inference time depends on the depth and number of decision trees. As configured, FlowPolicy can process packets at an average rate of 21,413 packets per second, which for 1500 B packets is 257 Mbps. This is sufficient to demonstrate the viability of cooperative rate control for typical Internet clients. We could achieve a higher rate by integrating feature generation into the TCP stack by making more extensive use of eBPF and using a more efficient inference implementation.

7 Related Work

This section discusses related work not covered earlier pertaining to service-level rate control, protocols beyond TCP, and other potential receiver rate control policies.

Techniques for service-level rate control A variety of systems have considered rate allocation at the granularity of parallel flows between two endpoints. TCP control block sharing [70] leverages existing flows' state information during new connection initialization. TCP-int [9] shares a congestion window between flows and uses packet delivery on one flow to detect loss on another, as does TCP sessions [58]. The Congestion Manager (CM) [10, 11] proposes a separate module that uses notifications about data transmission and receipt to provide "TCP-friendly" [36] congestion control at the granularity of a macroflow, and also supports protocols beyond TCP. The rate control policies provided by these techniques are stylistically related to those we explore in cooperative rate control. However, while this earlier work aggregates TCP information at the *sender* for flows to the same receiver and directly integrates with TCP (except for CM), cooperative rate control operates at the *receiver* and does not modify the TCP stack on either endpoint.

Other protocols We focus on TCP, but the QUIC [41] protocol is gaining popularity and some multimedia services use UDP and RTP [62]. Extending cooperative control to QUIC or RTP flows that use flow control is primarily an implementation issue. Raw UDP flows do not obey flow control and would therefore require another form of rate limiting, such as dropping packets.

Other policies While this paper describes ServicePolicy and FlowPolicy, other receiver policies are possible as well. Policies fall into two categories. First, policies arise due to differing receiver *objectives*, such as granularity, algorithm, and optimization metric. ServicePolicy and FlowPolicy both address this category. Other examples include sender-level fair queuing, RTT fairness, proportional fairness [48], or enforcing user-specified weights on traffic classes as in CRAB [67]. Second, policies arise due to differing receiver *visibility* into congestion. Examples include policies based on receiver resource constraints like CPU load and PCIe contention [2, 3, 16], and pathological traffic patterns such as incast, similar to ICTCP [72].

8 Conclusion

The debate over rate control on the Internet — what policies to enforce and which entities should enforce them — is alive and well. Our hope with this work is to propose a vision where the receiver has a stake in this process, without stripping the sender and network of their control. We have described the philosophy of cooperative rate control, and a proof of concept that shows two practical policies. The efficacy of these approaches opens the door for future work that incorporates more specific and dynamic algorithms that take inspiration from the rich literature on traffic scheduling and prioritization.

References

[1] Transmission Control Protocol. RFC 793, Sept. 1981.

[2] AGARWAL, S., AGARWAL, R., MONTAZERI, B., MOSHREF, M., ELMELEEGY, K., RIZZO, L., DE KRUIJF, M. A., KUMAR, G., RATNASAMY, S., CULLER, D., AND VAHDAT, A. Understanding host interconnect congestion. In *ACM HotNets 2022* (New York, NY, 2022), ACM, p. 198–204.

[3] AGARWAL, S., KRISHNAMURTHY, A., AND AGARWAL, R. Host congestion control. In *ACM SIGCOMM 2023* (New York, NY, 2023), ACM, p. 275–287.

[4] AKELLA, A., SESHAN, S., AND BALAKRISHNAN, H. The impact of false sharing on shared congestion management. In *11th IEEE International Conference on Network Protocols, 2003. Proceedings.* (2003), pp. 84–94.

[5] ALIZADEH, M., GREENBERG, A., MALTZ, D. A., PADHYE, J., PATEL, P., PRABHAKAR, B., SENGUPTA, S., AND SRIDHARAN, M. Data Center TCP (DCTCP). *ACM SIGCOMM CCR 40*, 4 (Aug. 2010), 63–74.

[6] ALIZADEH, M., KABBANI, A., EDSALL, T., PRABHAKAR, B., VAHDAT, A., AND YASUDA, M. Less is more: Trading a little bandwidth for ultra-low latency in the data center. In *USENIX NSDI 2012* (San Jose, CA, Apr. 2012), USENIX Association, pp. 253–266.

[7] BAGNULO, M., GARCIA-MARTINEZ, A., MONTENEGRO, G., AND BALASUBRAMANIAN, P. rLEDBAT: receiver-driven Low Extra Delay Background Transport for TCP. Internet-Draft draft-irtf-iccrg-rledbat-09, Internet Engineering Task Force, Nov. 2024. Work in Progress.

[8] BAKER, F., BLACK, D. L., NICHOLS, K., AND BLAKE, S. L. Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers. RFC 2474, Dec. 1998.

[9] BALAKRISHNAN, H., PADMANABHAN, V. N., SESHAN, S., STEMM, M., AND KATZ, R. H. TCP behavior of a busy Internet server: analysis and improvements. In *IEEE INFOCOM* (1998), vol. 1, pp. 252–262 vol.1.

[10] BALAKRISHNAN, H., RAHUL, H. S., AND SESHAN, S. An integrated congestion management architecture for Internet hosts. *ACM SIGCOMM CCR 29*, 4 (Aug. 1999), 175–187.

[11] BALAKRISHNAN, H., AND SESHAN, S. The Congestion Manager. Internet-Draft draft-balakrishnan-cm-03, Internet Engineering Task Force, Mar. 2000. Work in Progress.

[12] BORMAN, D., BRADEN, R. T., JACOBSON, V., AND SCHEFFENEGGER, R. TCP Extensions for High Performance. RFC 7323, Sept. 2014.

[13] BRAKMO, L. S., O'MALLEY, S. W., AND PETERSON, L. L. TCP Vegas: New techniques for congestion detection and avoidance. In *ACM SIGCOMM 1994* (New York, NY, 1994), ACM, p. 24–35.

[14] BREIMAN, L. Random forests. *Machine Learning 45*, 1 (2001), 5–32.

[15] BRISCOE, B. Flow rate fairness: Dismantling a religion. *ACM SIGCOMM CCR 37*, 2 (Mar. 2007), 63–74.

[16] CAI, Q., CHAUDHARY, S., VUPPALAPATI, M., HWANG, J., AND AGARWAL, R. Understanding host network stack overheads. In *ACM SIGCOMM 2021* (New York, NY, 2021), ACM, p. 65–77.

[17] CARDWELL, N., CHENG, Y., GUNN, C. S., YEGANEH, S. H., AND JACOBSON, V. BBR: Congestion-based congestion control. *ACM Queue 14, September-October* (2016), 20 – 53.

[18] CHIU, D.-M., AND JAIN, R. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Computer Networks and ISDN Systems 17*, 1 (June 1989), 1–14.

[19] CHO, I., JANG, K., AND HAN, D. Credit-scheduled delay-bounded congestion control for datacenters. In *ACM SIGCOMM 2017* (New York, NY, 2017), ACM, p. 239–252.

[20] CHOWDHURY, M., ZHONG, Y., AND STOICA, I. Efficient coflow scheduling with Varys. In *ACM SIGCOMM 2014* (New York, NY, 2014), ACM, p. 443–454.

[21] CLARK, D. D., WROCLAWSKI, J., SOLLINS, K. R., AND BRADEN, R. Tussle in cyberspace: Defining tomorrow's Internet. In *ACM SIGCOMM 2002* (New York, NY, 2002), ACM, p. 347–356.

[22] DEMERS, A., KESHAV, S., AND SHENKER, S. Analysis and simulation of a fair queueing algorithm. In *ACM SIGCOMM 1989* (New York, NY, 1989), ACM, p. 1–12.

[23] EBPF. eBPF - introduction, tutorials & community resources. `https://ebpf.io`, Sept. 2022.

[24] EDDY, W. Transmission Control Protocol (TCP). RFC 9293, Aug. 2022.

[25] FLOYD, S. HighSpeed TCP for Large Congestion Windows. RFC 3649, Dec. 2003.

[26] FLOYD, S. Metrics for the Evaluation of Congestion Control Mechanisms. RFC 5166, Mar. 2008.

[27] FLOYD, S., AND FALL, K. Promoting the use of end-to-end congestion control in the Internet. *IEEE/ACM Transactions on Networking 7*, 4 (Aug. 1999), 458–472.

[28] FRIEDMAN, J. Greedy function approximation: A gradient boosting machine. *The Annals of Statistics 29* (Nov. 2000).

[29] GAFNI, E., AND BERTSEKAS, D. Dynamic control of session input rates in communication networks. *IEEE Transactions on Automatic Control 29*, 11 (1984), 1009–1016.

[30] GAO, P. X., NARAYAN, A., KUMAR, G., AGARWAL, R., RATNASAMY, S., AND SHENKER, S. PHost: Distributed near-optimal datacenter transport over commodity network fabric. In *ACM CoNEXT 2015* (New York, NY, 2015), ACM.

[31] GROUP, T. T. tcpdump & libpcap. `https://www.tcpdump.org`, June 2023.

[32] GUPTA, R. WEBTP: A user-centric receiver-driven web transport protocol. Tech. Rep. UCB/ERL M98/77, EECS Department, University of California, Berkeley, Dec 1998.

[33] HA, S., RHEE, I., AND XU, L. CUBIC: A new TCP-friendly high-speed TCP variant. *ACM SIGOPS Operating Systems Review 42*, 5 (July 2008), 64–74.

[34] HAN, S., JANG, K., PANDA, A., PALKAR, S., HAN, D., AND RATNASAMY, S. SoftNIC: A software nic to augment hardware. Tech. Rep. UCB/EECS-2015-155, EECS Department, University of California, Berkeley, May 2015.

[35] HANDLEY, M., RAICIU, C., AGACHE, A., VOINESCU, A., MOORE, A. W., ANTICHI, G., AND WÓJCIK, M. Re-architecting datacenter networks and stacks for low latency and high performance. In *ACM SIGCOMM 2017* (New York, NY, 2017), ACM, p. 29–42.

[36] HANDLEY, M. J., PADHYE, J., FLOYD, S., AND WIDMER, J. TCP Friendly Rate Control (TFRC): Protocol Specification. RFC 5348, Sept. 2008.

[37] HE, K., ROZNER, E., AGARWAL, K., GU, Y. J., FELTER, W., CARTER, J., AND AKELLA, A. AC/DC TCP: Virtual congestion control enforcement for datacenter networks. In *ACM SIGCOMM 2016* (New York, NY, 2016), ACM, p. 244–257.

[38] HOCHREITER, S., AND SCHMIDHUBER, J. Long short-term memory. *Neural computation 9* (Dec. 1997), 1735–80.

[39] IO VISOR. BPF compiler collection (bcc). `https://github.com/iovisor/bcc`, Sept. 2022.

[40] IYENGAR, J., AND THOMSON, M. QUIC: A UDP-Based Multiplexed and Secure Transport. RFC 9000, May 2021.

[41] IYENGAR, J., AND THOMSON, M. QUIC: A UDP-Based Multiplexed and Secure Transport. RFC 9000, May 2021.

[42] JACOBSON, V. Modified TCP congestion avoidance algorithm. Email to end2end-interest Mailing List. Obtain via ftp://ftp.ee.lbl.gov/email/ vanj.90apr30.txt, Apr. 1990.

[43] JAIN, R., AND RAMAKRISHNAN, K. K. Congestion avoidance in computer networks with a connectionless network layer: concepts, goals and methodology. In *[1988] Proceedings. Computer Networking Symposium* (1988), pp. 134–143.

[44] JAIN, R. K., CHIU, D.-M. W., AND HAWE, W. R. A quantitative measure of fairness and discrimination. *Eastern Research Laboratory, Digital Equipment Corporation, Hudson, MA 21* (1984).

[45] JEYAKUMAR, V., ALIZADEH, M., MAZIÈRES, D., PRABHAKAR, B., KIM, C., AND GREENBERG, A. EyeQ: Practical network performance isolation at the edge. In *USENIX NSDI 2013* (USA, 2013), USENIX Association, p. 297–312.

[46] KALAMPOUKAS, L., VARMA, A., AND RAMAKRISHNAN, K. K. Explicit window adaptation: a method to enhance TCP performance. In *IEEE INFOCOM* (1998), vol. 1, pp. 242–251 vol.1.

[47] KE, G., MENG, Q., FINLEY, T., WANG, T., CHEN, W., MA, W., YE, Q., AND LIU, T.-Y. LightGBM: A highly efficient gradient boosting decision tree. In *NIPS 2017* (Red Hook, NY, 2017), Curran Associates Inc., p. 3149–3157.

[48] KELLY, F. Charging and rate control for elastic traffic. *European Transactions on Telecommunications 8* (Feb. 1997).

[49] KOKOSKA, S., AND ZWILLINGER, D. *CRC Standard Probability and Statistics Tables and Formulae (1st ed.)*. CRC Press, 2000.

[50] KUMAR, P., DUKKIPATI, N., LEWIS, N., CUI, Y., WANG, Y., LI, C., VALANCIUS, V., ADRIAENS, J., GRIBBLE, S., FOSTER, N., AND VAHDAT, A. PicNIC: Predictable virtualized nic. In *ACM SIGCOMM 2019* (New York, NY, 2019), ACM, p. 351–366.

[51] MATHIS, M., SEMKE, J., MAHDAVI, J., AND OTT, T. The macroscopic behavior of the TCP congestion avoidance algorithm. *ACM SIGCOMM CCR 27*, 3 (July 1997), 67–82.

[52] MEGA. Home - mega. `https://mega.io`, Feb. 2024.

[53] MIAO, J., AND NIU, L. A survey on feature selection. *Procedia Computer Science 91* (Dec. 2016), 919–926.

[54] MONTAZERI, B., LI, Y., ALIZADEH, M., AND OUSTERHOUT, J. Homa: A receiver-driven low-latency transport protocol using network priorities. In *ACM SIGCOMM 2018* (New York, NY, 2018), ACM, p. 221–235.

[55] NAGLE, J. On Packet Switches With Infinite Storage. RFC 970, Dec. 1985.

[56] NATHAN, V., SIVARAMAN, V., ADDANKI, R., KHANI, M., GOYAL, P., AND ALIZADEH, M. End-to-end transport for video QoE fairness. In *ACM SIGCOMM 2019* (New York, NY, 2019), ACM, p. 408–423.

[57] NIELSEN, H., MOGUL, J., MASINTER, L. M., FIELDING, R. T., GETTYS, J., LEACH, P. J., AND BERNERS-LEE, T. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, June 1999.

[58] PADMANABHAN, V. N. Addressing the challenges of web data transport, 1998.

[59] PERRY, J., OUSTERHOUT, A., BALAKRISHNAN, H., SHAH, D., AND FUGAL, H. Fastpass: a centralized "zero-queue" datacenter network. *ACM SIGCOMM CCR 44*, 4 (Aug. 2014), 307–318.

[60] POPA, L., YALAGANDULA, P., BANERJEE, S., MOGUL, J. C., TURNER, Y., AND SANTOS, J. R. ElasticSwitch: Practical work-conserving bandwidth guarantees for cloud computing. In *ACM SIGCOMM 2013* (New York, NY, 2013), ACM, p. 351–362.

[61] SAEED, A., DUKKIPATI, N., VALANCIUS, V., THE LAM, V., CONTAVALLI, C., AND VAHDAT, A. Carousel: Scalable traffic shaping at end hosts. In *ACM SIGCOMM 2017* (New York, NY, 2017), ACM, p. 404–417.

[62] SCHULZRINNE, H., CASNER, S. L., FREDERICK, R., AND JACOBSON, V. RTP: A Transport Protocol for Real-Time Applications. RFC 3550, July 2003.

[63] SCIKIT-LEARN. Permutation importance with multicollinear or correlated features. `https://scikit-learn.org/stable/auto_examples/inspection/plot_permutation_importance_multicollinear.html`, Sept. 2022.

[64] SHIEH, A., KANDULA, S., GREENBERG, A., AND KIM, C. Seawall: Performance isolation for cloud datacenter networks. In *USENIX HotCloud 2010* (USA, 2010), USENIX Association, p. 1.

[65] SIVARAMAN, A., WINSTEIN, K., THAKER, P., AND BALAKRISHNAN, H. An experimental study of the learnability of congestion control. In *ACM SIGCOMM 2014* (New York, NY, 2014), ACM, p. 479–490.

[66] SPRING, N. T., CHESIRE, M., BERRYMAN, M., SAHASRANAMAN, V., ANDERSON, T., AND BERSHAD, B. Receiver based management of low bandwidth access links. In *IEEE INFOCOM* (2000), vol. 1, pp. 245–254 vol.1.

[67] TAHIR, A., AND MITTAL, R. Enabling users to control their Internet. In *USENIX NSDI 2023* (Boston, MA, Apr. 2023), USENIX Association, pp. 555–573.

[68] THAPETA, V. S., SHINDE, K., MALEKPOURSHAHRAKI, M., GRASSI, D., VAMANAN, B., AND STEPHENS, B. E. Nimble: Scalable TCP-friendly programmable in-network rate-limiting. In *ACM SIGCOMM SOSR 2021* (New York, NY, 2021), ACM, p. 27–40.

[69] THOMSON, M., AND BENFIELD, C. HTTP/2. RFC 9113, June 2022.

[70] TOUCH, J. D., WELZL, M., AND ISLAM, S. TCP Control Block Interdependence. RFC 9040, July 2021.

[71] WARE, R., MUKERJEE, M. K., SESHAN, S., AND SHERRY, J. Modeling bbr's interactions with loss-based congestion control. In *ACM IMC* (New York, NY, 2019), IMC '19, ACM, p. 137–143.

[72] WU, H., FENG, Z., GUO, C., AND ZHANG, Y. ICTCP: Incast congestion control for TCP in data center networks. In *ACM CoNEXT 2010* (New York, NY, 2010), ACM.

[73] YEN, C.-Y., ABBASLOO, S., AND CHAO, H. J. Computers can learn from the heuristic designs and master internet congestion control. In *ACM SIGCOMM 2023* (New York, NY, USA, 2023), ACM, p. 255–274.

[74] YOUTUBE. Youtube. `https://www.youtube.com`, Feb. 2024.

[75] ZAPLETAL, A., AND KUIPERS, F. Slowdown as a metric for congestion control fairness. In *ACM HotNets 2023* (New York, NY, USA, 2023), ACM, p. 205–212.

A Implementation details

A.1 FlowPolicy features

A.1.1 All features

The tables in this section describe the FlowPolicy model's input features in detail. Table 4 lists all features and marks whether each is used as an instantaneous (i.e., raw) value or with dynamic history (either EWMA or average over a sliding window). Table 5 shows the parameter ranges for dynamic history (EWMA decay rates and window sizes).

A.1.2 Selected features

Examining the set of features that dynamic history (Section 5.2.2) selects for each model provides insights into the characteristics that are important to the receiver's rate control policy. Table 6 shows the top 10 (out of 20 total) features selected by the three models evaluated in this paper. The chosen features are determined by two properties: (1) feature correlation in the training data determines feature clusters, and to ensure a diverse set of features, one feature is chosen per cluster; (2) the importance of features to the trained model determines which feature is selected from each cluster, as well as the ranking in this table. See Table 4 for the definition of each feature.

Three categories of features occur in all three models, albeit with different dynamic history parameters: $1/\sqrt{\text{loss rate}}$, RTT ratio, and interarrival time. The inverse of the square root of the loss rate, which is a component of the Mathis equation, provides insight into typical TCP-friendly, loss-based protocol throughput. The ratio of the current RTT to the minimum RTT is a measure of relative queuing, and therefore congestion, in the network; the two models with delay-sensitive Vegas have more RTT-based features. The interarrival time is a useful measure of the sustained bit rate over the averaging window. The first two models consider interarrival time over long durations like 32 RTTs, whereas the third model focuses on short durations. Given the combination of loss, delay and rate-tracking protocol designs that these models focus on, this set of parameters seems very appropriate.

A.2 RateMon implements RWND tuning using eBPF

eBPF, the extended Berkeley Packet Filter, is a safe runtime for inserting verified code into the Linux kernel by registering callbacks at various hook-points [23]. We selected eBPF for this implementation because it allows low-level access into the networking stack without the complexity of writing a custom TCP variant, a kernel module, or modifying the networking stack itself. It also provides primitives for sharing data between user and kernel-space. Our implementation will run on any stock Linux kernel, v5.10 or newer.

The RWND tuning module implements two eBPF programs. First, calculating the receiver advertised window requires the receiver's TCP window scale value, which is specified by each side of the connection as a TCP option during the TCP handshake [12]. We use an eBPF `sock_ops` program that reads the window scale from the receiver's outgoing SYN-ACK packet and records it in an eBPF map. Second, we implement RWND tuning itself as an eBPF `tc` egress filter. The

filter processes every outgoing packet, retrieves a flow's custom RWND value from the decision store, applies the flow's window scale value, and encodes the minimum of the custom RWND value and TCP's existing RWND value (determined by available receive memory) into the TCP header's advertised window field. Registering these eBPF programs and sharing the decision store between the userspace RateMon controller and eBPF is accomplished using the BPF compiler collection (`bcc`) [39].

A.3 FlowPolicy training

A.3.1 Training data

Our core training data includes many configurations from Table 1, chosen uniformly at random without replacement. Table 7 shows the number of network configurations in the training data. However, we found that selecting configurations uniformly at random underrepresents situations where the per-flow fair rate is high, so we add additional configurations with a high fair rate to help balance the data. In all models, FlowPolicy mixes in training data with only incumbent flows (e.g., Reno vs. Reno) to improve accuracy. We select only the last 20% of each training experiment, randomly select 10% from this portion, and then randomly divide this into 70% training and 30% testing sets. The training, testing, and evaluation samples are chosen from separate network configurations to avoid polluting the results.

A.3.2 Training hyperparameters

Learning CCA behavior under diverse network conditions is a broad task, so we allow the decision tree to grow quite large: we constrain the DT to a depth of 100 levels and a width of 10,000 leaf nodes. Intermediate node splitting is allowed above 10 samples. Samples are weighted inversely proportional to their class's popularity. We stop training at 100 iterations, but in practice the models converge before this. 10% of training data is set aside for validation-based early stopping, which occurs when the validation loss has not improved by at least 10^{-6} for 10 iterations. These parameters were chosen based on manual fine-tuning of the training process and hyperparameter optimization. Perfecting the model is a task for future work; our goal is simply to demonstrate the feasibility of cooperative rate control.

Table 4: All potential model features. The last three columns indicate whether a feature is used as its raw value and/or with dynamic history. Additional features may be appropriate as well, such as ECN marks if available. See Table 5 for dynamic history parameters.

Feature	Description	Raw	Dynamic History	
			EWMA	Window
Latest RTT estimate	Latest RTT estimate from the TCP timestamp option	✓	✓	✓
Minimum RTT	Smallest RTT estimate seen so far by this flow	✓	-	-
Ratio of RTT to min RTT	An implicit measure of the current queuing in the network	✓	✓	✓
Age of flow	Time since receiving the first packet from this flow	✓	-	-
Sequence number	Sequence number from the TCP header	✓	-	-
Bytes received for flow	Total number of payload bytes received by this flow	✓	-	-
Packet interarrival time	Time between arrival of current and previous packets	✓	✓	✓
Instantaneous throughput	1 / packet interarrival time	✓	✓	✓
Average throughput	Average throughput over a window	-	-	✓
Loss rate	Loss rate since the last received packet	-	✓	✓
Loss event rate (LER)	Treat multiple losses in the same RTT as one loss event [36]	-	-	✓
Mathis model throughput, using loss rate	Throughput that the Mathis model predicts a Reno flow should achieve given the current RTT and loss rate [51]	-	✓	✓
Mathis model throughput, using loss event rate	Same as above, but using the loss event rate	-	-	✓
$1/\sqrt{\text{loss rate}}$	Hint to the training process because in the Mathis model, bandwidth is proportional to $1/\sqrt{\text{loss rate}}$	-	✓	✓
$1/\sqrt{\text{loss event rate}}$	Same as above, but using the loss event rate	-	-	✓

Table 5: Dynamic history parameters.

Parameter	Values
EWMA decay rate	0.001–0.01 in steps of 0.001, 0.1–1 in steps of 0.1
Sliding window size	1, 2, 4, 8, 16, 32 $\times$ min RTT
Sliding window size when using LER	4, 8, 16 $\times$ current RTT

Table 6: Top 10 of 20 features (by permutation importance) chosen by dynamic history for the three FlowPolicy models.

Rank	Reno vs. Cubic		Vegas vs. Cubic		Cubic vs. {BBR, Vegas}	
	Feature	Decay rate or window size	Feature	Decay rate or window size	Feature	Decay rate or window size
1	$1/\sqrt{\text{loss rate}}$	0.001	$1/\sqrt{\text{loss event rate}}$	$8\times$	$1/\sqrt{\text{loss rate}}$	0.005
2	$1/\sqrt{\text{loss rate}}$	0.5	$1/\sqrt{\text{loss rate}}$	0.001	$1/\sqrt{\text{loss rate}}$	0.1
3	$1/\sqrt{\text{loss rate}}$	1.0	$1/\sqrt{\text{loss rate}}$	0.4	$1/\sqrt{\text{loss rate}}$	$8\times$
4	$1/\sqrt{\text{loss rate}}$	$32\times$	$1/\sqrt{\text{loss rate}}$	1.0	RTT ratio	$16\times$
5	RTT ratio	-	$1/\sqrt{\text{loss rate}}$	$32\times$	RTT ratio	$8\times$
6	RTT ratio	$32\times$	RTT ratio	$1\times$	RTT	$8\times$
7	interarrival time	0.1	RTT ratio	$16\times$	age of flow	-
8	interarrival time	$32\times$	RTT ratio	$32\times$	interarrival time	-
9	interarrival time	$8\times$	interarrival time	0.3	interarrival time	$8\times$
10	instantaneous throughput	0.001	interarrival time	$32\times$	instantaneous throughput	0.001

Table 7: The number of network configurations in a FlowPolicy model's training data, as chosen from Table 1.

FlowPolicy Model (J vs. K)	J vs K; Random	J vs. K; High Fair Rate	J vs. J
Reno vs. Cubic	500	100	200
Vegas vs. Cubic	300	300	200
Cubic vs. {BBR, Vegas}	4000	2500	6500