

Dual-Endpoint Congestion Control

Christopher Canel

CMU-CS-26-128

August 2026

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

Thesis Committee:

Srinivasan Seshan, Chair
Justine Sherry
Peter Steenkiste
Neil Spring (Meta)

*Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.*

Copyright © 2026 **Christopher Canel**

This work was supported in part by the National Science Foundation through award #2212390. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

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

For Jessica

Abstract

Current techniques for rate control in computer networks are not aligned with the policies and considerations of both endpoints in a connection. Historically, fine-grained rate control has been the sender’s responsibility through the Transmission Control Protocol (TCP) and its congestion control algorithm (CCA). However, the receiver, either due to conflicting priorities or greater visibility into congestion, would often be better served by a different rate allocation than the sender. A simple example is a many-flow incast in a data-center network: each sender independently seeks to transmit as quickly as possible, while the receiver can observe that each flow should converge to a small fraction of the last-hop link rate. Likewise, on the Internet, each service provider balances between its customers, whereas a user may desire fairness across services.

To bridge the gap between endpoint objectives, we argue for a dual-endpoint approach to congestion control that incorporates the receiver into the rate decision by providing lightweight hints to senders about the rate regime in which they should operate. We use these techniques to build expressive receiver-assisted policies based on application traffic patterns, observed network congestion, and receiver resource constraints. Dual-endpoint congestion control differs from fully receiver-based techniques because the sender, with its own inherent observations and objectives, maintains control over the packet stream. To implement receiver assistance, we revisit the well-known technique of TCP flow control and show it to be a powerful primitive that does not require modifications to TCP, the sender’s CCA, or applications.

This dissertation explores two environments with differing endpoint objectives to show that dual-endpoint congestion control gives both endpoints a stake in bandwidth allocation decisions. First, we consider datacenter incast bursts, where the receiver’s observability into the traffic pattern enables dual-endpoint control to schedule hundreds or thousands flows into a healthy regime that both improves network utilization and reduces receiver packet processing overheads. Second, we turn to the Internet at large and show the expressivity of dual-endpoint control by resolving common challenges that arise due to conflicting receiver policies regarding rate control granularity, algorithm, and optimization metric.

Acknowledgments

This dissertation would not exist without the support of my colleagues at CMU and Meta.

I would like to thank my PhD advisor, Srinivasan Seshan, for his guidance and patience throughout this journey. I first met Srimi during an interview for CMU and I recall his eagerness to lend a helping hand even during Visit Day and IC. I took a three-year detour into machine learning systems, but when I was ready to return to networking, Srimi welcomed me into his group. I am particularly grateful for his support of my long-running collaboration with Meta, which led to most of my thesis work and a fulfilling job post-PhD.

I also thank my thesis committee, Justine Sherry, Peter Steenkiste, and Neil Spring, for their collaboration over these years. Justine's guidance and friendship have been constants for more than a decade. Co-teaching the computer networks course with Peter was the most personally fulfilling experience of my time at CMU. Neil has been my mentor at Meta for the past five years; he embodies a unique combination of academic rigor and practical engineering that both inspires me and grounds our projects.

I spent my first three years at CMU in the Parallel Data Lab, building machine learning systems as a leader in the Intel Science and Technology Center for Visual Cloud Systems. I would like to thank David G. Andersen and Michael Kaminsky for the opportunity to learn this (at the time) nascent field, which now impacts everyone's lives. Combining systems research with practical systems-building established the model that I want my career to follow. The PDL members, including Thomas Kim, Angela Jiang, Anuj Kalia, Giulio Zhou, Sol Boucher, Daniel Lin-Kit Wong, Conglong Li, Hyeontaek Lim, Huanchen Zhang, Dong Zhou, Michael Kozuch, Padmanabhan Pillai, Kayvon Fatahalian, George Amvrosiadis, and Greg Ganger, formed the most nurturing group of researchers I have seen. Our time preparing talks and posters for PDL and ISTC events taught me how to scope and discuss research.

In the networking group, I would like to thank Sagar Bharadwaj, Shawn Shuoshuo Chen, Mallesham Dasari, Ranysha Ware, Devdeep Ray, Hugo Sadok, Margarida Ferreira, Miguel Ferreira, Francisco Pereira, Isabel Suizo, Yiran Lei, Nirav Atre, and Adithya Abraham Philip for their comradery. The opportunity to manage the group's server racks taught me practical skills in Linux network administration that I still use every day. Thank you to Kevin Lin, Elliot Lockerman, and Ron He for their contributions to Chapters 4 and 5. I would like to give special thanks to Anup Agarwal for his dedication to our friendship.

My collaboration with Meta defined the final years of my PhD. Balasubramanian Madhavan's tireless dedication and ability to unblock any problem set an example that I follow to this day. Thank you to Prashanth Kannan for advocating for my full-time position, to Srikanth Sundaresan for his unending interest in difficult problems, and to Abhishek Dhamija for teaching me to approach systems with an eye towards production. These colleagues, along with Neil Spring and many others, built the static RWND tuning technique analyzed in Chapter 3. I also extend my appreciation to Rajiv Krishnamurthy and Omar Baldonado for their support and feedback on paper submissions. An interview with Rajiv convinced me to join the Host Networking team, a decision that led to Chapters 2–4.

Finally, I must thank my lovely wife, Jessica Canel. When I joined CMU, I expected to be surrounded by brilliant researchers, but from nobody have I learned more than Jess. She stood by my side during all of the unexpected turns in my journey. This dissertation is as much a testament to her support as it is to the technical contributions within. Hidden behind this work is the family that we have built, and that is the true legacy of my PhD.

Contents

1	Introduction	1
1.1	Overview	1
1.1.1	Understanding datacenter incast bursts (Chapter 2)	3
1.1.2	A case study in dual-endpoint control: Receiver window tuning (Chapter 3)	3
1.1.3	Receiver flow scheduling (Chapter 4)	4
1.1.4	Dual-endpoint control for Internet fairness (Chapter 5)	5
1.2	Background	6
1.2.1	Congestion control in a nutshell	6
1.2.2	Steps towards dual-endpoint control	7
1.3	Leveraging TCP Flow Control to Enable Receiver Assistance	8
1.3.1	Flow control overview	8
1.3.2	Receiver window tuning	8
1.3.3	Existing uses of RWND tuning	9
1.3.4	Types of policies in dual-endpoint control	9
2	Understanding Datacenter Incast Bursts	11
2.1	Introduction	11
2.2	Measurement Environment	12
2.3	Measuring Incast Bursts	12
2.3.1	Utilization and loss rate	12
2.3.2	Burst frequency and duration	14
2.3.3	Incast degree distribution is stable	16
2.3.4	Queuing and ECN marking	17
2.3.5	Retransmissions	17
2.4	Understanding Incast Bursts	17
2.4.1	DCTCP modes of operation	19
2.4.2	Short bursts, the common case	21
2.4.3	Divergence at burst boundaries	21
2.5	Discussion	22
2.5.1	Interactions across bursts	22
2.5.2	Limits to window control	22
3	A Case Study in Dual-Endpoint Control: Receiver Window Tuning	25
3.1	Introduction	25
3.2	Proof of Concept in Simulation	26
3.3	Design and Implementation	28
3.3.1	eBPF tc egress	28

3.3.2	SACK adjustment	29
3.3.3	Window scaling	29
3.3.4	Choosing an advertised window clamp	29
3.4	Production Evaluation	30
3.4.1	Example application	31
3.4.2	Services in depth	31
3.4.3	RWND tuning prevents negative interactions with heterogeneous traffic	33
3.5	Conclusion	35
4	Receiver Flow Scheduling	37
4.1	Introduction	37
4.2	A Host Perspective on Incast	38
4.2.1	Role of senders vs. receivers	39
4.2.2	Network stack optimizations	39
4.2.3	Measurement study of receiver efficiency	40
4.3	Scheduling enables large incasts and improved receiver efficiency	43
4.3.1	Introduction to receiver flow scheduling	43
4.3.2	Benefits of changing the network service model	44
4.3.3	Scheduling algorithms	45
4.3.4	Proof of concept in simulation	45
4.4	Related Work	46
4.4.1	Challenges with sender-only incast solutions	46
4.4.2	Scheduling techniques	46
4.4.3	Non-CCA techniques	47
4.4.4	Host bottlenecks	47
4.5	Design and Implementation	48
4.5.1	Incast burst generator	48
4.5.2	Design overview	48
4.5.3	Implementation details	51
4.5.4	Assumptions	52
4.6	Evaluation	53
4.6.1	Evaluation Setup	53
4.6.2	Overview: Benefits for network utilization and receiver efficiency	55
4.6.3	Burst duration and average throughput	57
4.6.4	CPU utilization	57
4.6.5	Socket Buffer Length	57
4.6.6	Cache behavior	59
4.6.7	Instructions per cycle	59
4.7	Conclusion	61
5	Dual-Endpoint Control for Internet Fairness	63
5.1	Introduction	63
5.2	Background & Motivation	65
5.2.1	Rate control and fairness	65
5.2.2	Challenges with bandwidth allocation today	66
5.3	Design Overview	71
5.3.1	Design constraints	71

5.3.2	Limitations of receiver window tuning	72
5.3.3	The Ratemon platform	72
5.4	ServicePolicy	73
5.4.1	ServicePolicy definition	73
5.4.2	ServicePolicy implementation	73
5.4.3	ServicePolicy demonstration	74
5.5	FlowPolicy	74
5.5.1	FlowPolicy definition	75
5.5.2	FlowPolicy implementation	76
5.5.3	FlowPolicy mitigation	79
5.5.4	FlowPolicy demonstration	79
5.6	Evaluation	80
5.6.1	Broad evaluation of the three challenges	80
5.6.2	Robustness and generalizability	82
5.6.3	Microbenchmarks	87
5.7	Related Work	87
5.8	Implementation Details	88
5.8.1	FlowPolicy features	88
5.8.2	Ratemon implements RWND tuning using eBPF	88
5.8.3	FlowPolicy training	90
5.9	Conclusion	91
6	Conclusion	93
6.1	Takeaways	93
6.2	Limitations and Alternative Designs	94
6.2.1	Limitations of flow control	94
6.2.2	Involving the application layer	96
6.3	Other Relevant Trends	97
6.3.1	Involving receivers in congestion control	97
6.3.2	Internet fairness studies	98
6.4	Additional Future Directions	98
6.4.1	A novel Internet environment	99
6.4.2	In-network policies	99
6.5	Closing Remarks	100
	Bibliography	101

List of Figures

2.1	Example of incast bursts, measured at a receiver	13
2.2	Incast burst characteristics across five production services.	15
2.3	Distribution of flow count during a burst	16
2.4	Negative effects of incast bursts on the network	18
2.5	DCTCP operating modes during 15 ms incast bursts	20
2.6	Queue behavior during 2 ms incast bursts	21
2.7	Per-flow in-flight data is highly skewed	22
3.1	Benefits of RWND tuning with various clamp sizes for 15 ms incast bursts	27
3.2	Queue length for various RWND clamps applied to 2 ms incast bursts	28
3.3	Retransmission rate for various RWND clamps	31
3.4	Impact of RWND tuning on key metrics across all five services	32
3.5	Impact of various RWND clamps on key metrics for the aggregator service	32
3.6	Bursts on in-region flows cause retransmissions for cross-region flows	34
3.7	Benefits of RWND tuning applied to a bursty cross-region storage service	35
4.1	Throughput and CPU use for increasing flow counts	40
4.2	Total CPU utilization increases with flow count	41
4.3	Throughput efficiency per unit compute decreases with more flows	41
4.4	Segment size decreases as the flow count increases	41
4.5	L1, L2, and L3 cache access counts per 1 MB of received data	43
4.6	Bottleneck queue occupancy during a 2000-way incast	46
4.7	The architecture of receiver flow scheduling	49
4.8	CloudLab incast testbed	53
4.9	Dual-endpoint control benefits both the network and the receiver host	54
4.10	Burst duration CDF	56
4.11	Average burst throughput CDF	56
4.12	CPU utilization CDF	56
4.13	Socket buffer length CDF	58
4.14	L3 cache hit ratio per 1 MB CDF	58
4.15	Instructions per cycle CDF	58
4.16	L1, L2, and L3 cache access count CDFs	60
5.1	Example 1: A service using many flows claims a higher rate	67
5.2	Example 2: Two CCAs with different algorithms do not converge to equal rates	68
5.3	Example 3: Two CCAs with different congestion metrics do not converge to equal rates	69
5.4	The Ratemon platform	72
5.5	Comparing decision architectures	75

5.6	The emulator's topology	76
5.7	ServicePolicy improves fairness at the service granularity	81
5.8	FlowPolicy improves fairness under different rate control algorithms	81
5.9	FlowPolicy improves fairness under different optimization metrics	82
5.10	FlowPolicy prevents a rate-based newcomer from harming loss-based incumbents	84
5.11	FlowPolicy improves fairness between Cubic and all CCAs in Linux	84
5.12	FlowPolicy improves fairness on CCAs not in its training data	85
5.13	FlowPolicy improves fairness for flows that traverse the Internet	86

List of Tables

2.1	Five example services	14
3.1	Static RWND clamps for the five services in Table 2.1	30
5.1	Parameters for training and evaluation	77
5.2	Analysis of partial deployments of FlowPolicy	86
5.3	FlowPolicy model accuracy on test data	87
5.4	All potential model features	89
5.5	Dynamic history parameters	89
5.6	Top 10 of 20 features for the three FlowPolicy models	90
5.7	The number of network configurations in a FlowPolicy model’s training data	90

Chapter 1

Introduction

In computer networks, many considerations play a role in determining the rate at which data is transmitted. Each entity along a path through the network contributes to the end-to-end rate allocation decision. Sending hosts offer load into the network at a rate of their choosing. Network switches, routers, and middleboxes move and shape the aggregate load contributed by tens to millions of users. At the other end of the network, receivers assemble the data into a coherent buffer and deliver it to applications. At each of these devices, hardware capabilities, such as line rate, processing speed, and memory capacity, interact with application traffic patterns, software algorithms for rate control, and user policies for resource sharing.

This dissertation examines the role of different network entities and how their individual considerations impact rate allocation. Much existing work has focused on rate control at all points of the network. However, we find that current techniques for rate control in computer networks both depend on, but are not aligned with, the objectives of *both of the endpoints* in a connection. Historically, a *sender* is responsible for converging to a fair share of the bottleneck bandwidth on a stable network shared with other flows using the same algorithm. However, both on the Internet and in datacenters, flows often interact in an environment where the *receiver*, either through differing visibility into congestion, resource constraints, or priorities, would be better served by a different rate allocation. A simple example is a many-flow incast in a datacenter network: each sender only has visibility into its own outgoing flows and seeks to balance across them, while the receiver can observe the whole scale of an incast and knows that each flow should converge to a small fraction of its incoming link rate. Similarly, on the Internet, each service provider arbitrates *across its customers*, whereas a user may optimize for fairness *across their services*. Instead of focusing on a single entity, we consider how to bring their disparate information together in a lightweight manner to enable a dual-endpoint rate decision.

1.1 Overview

Traditionally, fine-grained rate control has been the responsibility of the transport layer through the Transmission Control Protocol (TCP) and a sender-based congestion control algorithm (CCA) [14], such as Cubic [51] or Data Center TCP [5]. The sender controls much information relevant to determining a rate, including available application data, the capabilities and current users of transmit resources, and state learned about the network via ongoing probing. Since the sender controls the application data, it is ultimately responsible for putting bytes on the wire. However, the sender is fundamentally limited in its observability into the network and the information it can glean to make a rate control decision. Its state is either learned from afar via probing, and thus inherently an approximation, or the result of shipping network state back to the sender, such as by marking packets [5] or appending network device telemetry [72].

Receivers also possess relevant information, such as the presence of other connections to the same

endpoint that might cause congestion, operating system and hardware capabilities such as the available processing cycles and memory for ingesting packets, higher-level directives such as whether the application is ready to consume the data, and human user policies around traffic prioritization and fairness. Past approaches have considered how to unify the information of both endpoints, and typically entail the receiver sending *either* information *or* a computed rate decision to the sender. The sender then either computes on this state to produce a rate decision or immediately uses the provided decision to control transmission. An example of relaying state to the sender is acknowledgments with extra information, such as explicit congestion notification (ECN) marks [5, 43] or in-network telemetry (INT) [1, 72, 117]. The sender then uses this state to grow or shrink a window of in-flight packets or set an explicit transmit rate [5]. An example of relaying computational results is a centralized or receiver-centric controller, such as Homa [85] and NDP [53], where all flow scheduling is performed on another entity which then dictates to the sender when and how fast to transmit.

There are two drawbacks with a design where one endpoint makes the *full* rate control decision. First, transmitting all information to one endpoint is not practical. Each network device has a plethora of state from which it could base its own decision on target rate, and these vary by device. Second, each entity has its own guiding policies, such as prioritization and resource bounds, that determine how to distill that information into a rate decision. No one entity can hope to capture these device-specific nuances. Therefore, a better option is to combine *decision-making*. However, this is challenging because of the nature of incorporating multiple separate control loops [86]. This dissertation studies how to do so in practical settings and what benefits it provides.

We advocate for a cooperative approach to rate control that incorporates both the sender and receiver in the rate decision. Instead of centralizing state on one entity, our approach has the receiver share both state *and* its own control decision with the sender. Through this approach, which we term *dual-endpoint congestion control*, the receiver provides valuable rate guidance to the sender based on receiver-specific knowledge of flow objectives and traffic patterns, and the sender still learns the state it needs to make its own decision as well. The sender combines the incoming state with its internal state to create its own rate decision, which it then combines with the rate decision provided by the receiver. Many ways exist to combine these decisions, but for the sake of respecting both endpoints' resource constraints, in this work the resulting rate is the minimum of both endpoints' decisions.

Dual-endpoint control makes the objectives of both endpoints first-class entities. To accomplish this, we propose a set of lightweight mechanisms to enable explicit coordination of the sender and receiver rate decisions where senders previously relied on implicit feedback alone. These techniques enable both endpoints to enforce expressive policies based on resource constraints, congestion observations, or user and application objectives. In earlier work, Spring *et al.* [107] also proposed a cooperative approach to congestion control, focusing on the context of bandwidth management of access links. They argue that the receiver is responsible for congestion on the last-hop link while the sender is responsible for congestion in the rest of the network. Inspired by their proposal, we generalize this cooperation and show example endpoint objectives for scenarios involving pathological incast in datacenter networks and flow fairness on the Internet.

Thesis statement: *When sender objectives, such as fairness across clients, differ from receiver objectives, such as fairness across services, dual-endpoint congestion control provides a lightweight mechanism for incorporating both endpoints' objectives into the rate control decision.*

This dissertation focuses first on a well-controlled environment where sender and receiver objectives

differ: datacenter incast bursts. In an incast, tens to thousands of senders each transmit to a single receiver. These flows collide at the last-hop switch before the receiver and each sender must independently reduce their rate to a small fraction of the receiver’s total bandwidth. Senders probe the network and observe the resulting implicit impact on their flows, such as increasing queuing latency or switch-based congestion marks. This allows the sender to build a model of the network path, including bottlenecks that are independent of the incast, which it combines with local information such as its resource constraints and the other requests it is serving. However, the sender lacks explicit knowledge of the ongoing incast. On the other hand, receivers naturally observe all flows in an incast and the impact of that incast on local resource contention on the receiver itself, but currently lack the opportunity to incorporate that information into the rate control decision. With existing single-endpoint solutions (whether based on the sender or the receiver), the local observations and considerations of one endpoint are neglected.

We begin with a measurement study to characterize the properties of incast bursts in workloads today and motivate the need for a solution beyond single-endpoint or in-network control. Second, we consider an initial solution for receiver-assisted control built on TCP’s receiver window tuning and made possible by a set of realistic workload assumptions. Third, we relax these assumptions and build a dynamic receiver flow scheduling mechanism to support incast bursts with thousands of flows, an order of magnitude more than existing TCP-based techniques, while also improving receiver efficiency.

Finally, we expand beyond the datacenter to a less tightly controlled environment, the Internet, and explore the implications that differing endpoint objectives have on flow fairness policies. We conclude with a discussion of the limitations of our approach, implications of recent related work, and directions for future study. The following four sections outline the contributions of this dissertation.

1.1.1 Understanding datacenter incast bursts (Chapter 2)

Chapters 2–4 examine the incast burst traffic pattern in datacenters. To set the stage, we start with a detailed characterization of incast bursts in a production datacenter at a major Internet content provider. In particular, we focus on *bursty* incast, which is a divergence from the *long-running* incast studied by most other incast-focused proposals [5, 71]. Bursty incast traffic patterns are challenging because they violate the basic premise of bandwidth stability on which TCP congestion control convergence is built. In addition to overwhelming shallow switch buffers, the on-off paradigm prevents TCP from stably converging to an appropriate rate allocation. This chapter first provides an introduction to the datacenter environment and follows with an in-depth investigation into the limits of sender control during high-degree incast bursts. We explore those properties via measurements at scale and show that real-world incasts often include an order of magnitude more flows than previous studies. In-depth simulations reveal that independent senders lack the coordination necessary to converge to a safe rate. This demonstrates a key drawback of sender-only rate control: disparate senders that rely purely on probing the network are incapable of coordinating to resolve this traffic pattern. A dual-endpoint technique, on the other hand, opens a source of additional observation (the receiver) and provides a method for explicit coordination. Finally, we show that hosts running an application exhibit similar and predictable burst properties, which paves the way for the sender-uniform service model in Chapter 3.

1.1.2 A case study in dual-endpoint control: Receiver window tuning (Chapter 3)

Building on the measurement study above, this chapter demonstrates the practicality of dual-endpoint control to mitigate incast bursts by studying a solution that was deployed at scale in production datacenters. Two factors combine to enable a simple example of dual-endpoint control in action. First, the receiver’s visibility into the incast traffic pattern provides a unique opportunity to provide a useful rate policy. Since the receiver observes the last hop link, it naturally has direct insight into the number of flows in an incast. Second, we observe in Chapter 2 that the scale of an incast is uniform across both time

and the hosts running a particular application. This enables the receiver to leverage a predetermined burst profile to determine a rate decision.

Our approach here is analogous in service model to fair queuing in network switches [33, 108]. In fair queuing, a switch observes the aggregate traffic volume over a link and establishes a model of operation where each flow is entitled to the same fair share. Using dual-endpoint control, we perform a similar operation over the flows in an incast burst. Based on the application’s burst profile, the receiver determines a fair allocation and delivers that decision uniformly to all senders in the incast. We refer to this as a sender-uniform service model, similar to fair queuing. This approach is simple to implement at runtime, but relies on assumptions about the predictability of application behavior. Once the receiver has computed its rate decision, we implement the dual-endpoint control by reusing existing features in TCP, namely the receiver’s advertised window (RWND), to inform the sender of the receiver’s rate (in practice, encoded as a window size). While a backwards-compatible mechanism like RWND in TCP is not strictly required for dual-endpoint control, a benefit of this implementation is that it does not require changes to the protocol or sender.

Together, this case study demonstrates that dual-endpoint congestion control, implemented by controlling RWND, provides the coordination necessary to help independent sender CCAs reduce the volume of in-flight data to keep queues short. Building on our observation of the predictability of application burst behavior enables us to avoid computationally expensive online adaptation and instead adopt a sender-uniform RWND tuning approach with low overheads. We draw on three years of production deployment experience to demonstrate that RWND tuning for incast bursts reduces retransmissions by $3.8\times$, leading to a $2\text{--}8\times$ reduction in service tail latency.

1.1.3 Receiver flow scheduling (Chapter 4)

The sender-uniform strategy for receiver rate allocation is illustrative of the capabilities of dual-endpoint control, but it has two practical drawbacks.

The first drawback to a sender-uniform approach is that network congestion is not the only consequence of an incast: receiver packet processing efficiency suffers under high degrees of multiplexing as well. For every packet that arrives, the receiver’s network stack must read that packet, update data structures tracking stream reliability, and potentially send an acknowledgment packet. In addition, the receiving application continuously polls its input buffer to fetch each segment of new data. This processing imposes a per-packet overhead, and if the receiver’s CPU cannot keep up, then input buffers grow and overflow. Although the network has delivered a packet to the receiver, it might be dropped before being delivered to the application. More importantly, CPU cycles that the receiver burns in this packet processing are cycles not spent servicing its applications. In a datacenter, wasting CPU cycles in this way increases the number of hosts required to serve user traffic, increasing infrastructure expense.

To reduce the impact of packet processing overheads, receivers implement a raft of optimizations that aggregate data across sequential packets from the same flow. Features such as generic receive offload, interrupt coalescing, and ACK coalescing combine the processing of multiple contiguous packets into a single operation. However, these depends on receiving multiple packets from the same flow in quick succession. In an incast with hundreds or thousands of flows, a single flow does not receive enough contiguous packets per unit of time to take advantage of these aggregations. This manifests as smaller coalesced packet sizes, increased CPU utilization, and poor cache performance. This chapter begins with an investigation into the impact of incast on these receiver mechanisms to show that as the number of flows increases, the per-byte host processing overhead also increases. A line of recent work has focused extensively on controlling congestion *within* the receiving host [2, 3, 20, 103, 116]. Their investigations address the reasons congestion occurs within the receiver stack; our work complements these findings by focusing on the impact of excessive multiplexing.

The second drawback to a sender-uniform approach is that by conveying a rate that maintains fairness across all flows, similar to fair queuing, the receiver signals that all flows should be active at once. This means that large incasts encounter a rate control limitation (as discussed in Section 2.5.2): each flow must have at least one packet in flight. However, for incasts with hundreds or thousands of flows, even one packet in flight per flow is, in aggregate, more data than switch buffers can hold. Once senders reduce their windows to a single packet, congestion control has reached a degenerate state where it cannot decrease further. And should a flow increase its window to two packets, that corresponds to a doubling of the offered load, which across all flows leads to large oscillates in congestion.

Together, the poor receiver efficiency under high degrees of multiplexing combined with the limitations that one-packet-per-flow places on the network’s ability to absorb incast motivate a novel solution that redefines the network service model. Instead of a sender-uniform policy where all flows make progress at the same time, albeit slowly with one or a few packets in flight, the receiver will enforce *flow scheduling* where only a subset of flows are active at once. Much work has focused on clean-slate congestion control techniques that provide centralized [28, 94] or receiver-based [85] flow scheduling, but, as throughout this dissertation, we propose a dual-endpoint design that builds this scheduling as a loose extension atop existing sender control mechanisms instead. The sender will continue to perform its typical congestion control algorithm, but dual-endpoint control will dictate, again using receiver window tuning, which flows are scheduled to be active and how much data each is allowed to send. Each active flow operates in a healthy window regime with many packets in flight. This allows us to trade off fine-grained flow fairness for improved efficiency across both the network and the receiver stack. By keeping the set of active flows small (≈ 20), receiver packet aggregation techniques deliver improved performance, reducing CPU overheads and improving cache performance. By reducing the scale of an incast to a manageable scope, this technique allows the receiver to coordinate arbitrarily large incasts, breaking the incast scale limitation of the sender-uniform approach.

Our results on a 100G testbed show that flow scheduling through dual-endpoint control improves both network utilization and receiver efficiency during incast bursts up to thousands of flows compared to sender-side congestion control alone. Our techniques reduce the P99 flow completion time for 1000–5000-flow incasts by up to 74.4%. In terms of receiver performance, average CPU utilization decreases by 59.6%. Looking closer, we see that this reduction is due to an increase in the median coalesced packet size of $6.9\times$, showing more effective use of receiver aggregation techniques due to the availability of more packets from each flow at once. In terms of the memory hierarchy, better access locality for kernel data structures due to having a smaller working set leads to a 63.0% reduction in average cache accesses. These combine to produce an 8.8% increase in the average instructions per cycle. Receiver flow scheduling is inherently a mechanism for decreasing sender rate, and these improvements come at the cost of an up to 12.0% burst duration penalty for fewer-flow incasts.

1.1.4 Dual-endpoint control for Internet fairness (Chapter 5)

For the final chapter, we leave datacenter incast bursts behind, relaxing our design constraints and expanding our notion of dual-endpoint control to focus on receiver policies in a more general manner. We turn instead to the Internet at large, which is inherently more heterogeneous and less tightly controlled than a datacenter serving well-understood production traffic. In terms of receiver policy, we consider primarily end users and look broadly at objectives for rate allocation between their applications, which is typically considered under the umbrella of “fairness”.

On the Internet, fairness objectives differ between endpoints. Even through both endpoints seek to maximize throughput, senders (typically content provider servers) often prioritize fairness across their *customers*, perhaps weighted by each customer’s purchased level of service. Receivers, on the other hand, must balance the many services which a user interacts with at a given time, likely weighted by

the utility of each service to the user. This disconnect leads to conflicting expectations about what a fair rate allocation should be. In this fairness battleground, we use dual-endpoint congestion control to allow both endpoints to enforce their own *fairness policies* cooperatively, leading to a natural union of their individual objectives. As examples of this approach in the Internet environment, we consider two receiver-side policies: (1) ServicePolicy, a policy to enforce per-service fair shares rather than per-flow fair shares, and (2) FlowPolicy, a policy to override unfair congestion control behavior. To enable these policies, we present Ratemon, a modification to a receiver’s TCP stack that monitors arriving packets, infers whether a flow is behaving unfairly according to the receiver’s policy, and enforces the policy-defined rate. Ratemon avoids modifications to the sender, network devices, or the TCP header by only using information available to the receiver and again leveraging TCP’s built-in flow control to transparently signal the sender to back off. We show in emulation and with Internet experiments that ServicePolicy increases Jain’s Fairness Index (JFI) [63] by 19% when calculated on a per-service basis and FlowPolicy increases JFI by 85% on a per-flow basis.

Our example fairness policies, while illustrative of the capability of dual-endpoint control, are not meant to be the end of the story on Internet rate allocation. Even after forty years of TCP congestion control, the debate over the design [23] and real-world properties [97] (including fairness [118]) of CCAs on the Internet is alive and well. We conclude with reflections on how incorporating the policies and constraints of both endpoints impacts future networking research.

1.2 Background

This section provides background about rate determination and congestion control in networks today. We describe existing multi-entity mechanisms that point the way towards our proposal for dual-endpoint rate control. Next, we provide an overview of TCP flow control and the receiver window tuning technique that the case studies in this dissertation use to implement dual-endpoint control. Context-specific details, such as application techniques for rate control, are addressed as needed in the following chapters. We end with a discussion of the different types of policies that a dual-endpoint approach could achieve, which provides a map of the remainder of this dissertation.

1.2.1 Congestion control in a nutshell

Traditionally, congestion control is implemented at a single endpoint, usually the sender. That endpoint exercises complete control over the packet stream. The CCA’s rate control algorithm, which governs how it probes for bandwidth and backs off under congestion, determines its interactions with other flows and its converged rate. A simple example is TCP Tahoe and its descendants, which use the additive increase, multiplicative decrease algorithm to converge to a fair allocation [50, 60, 61].

CCAs fall into broad categories based on the signals they use to detect congestion. Early CCAs such as Tahoe [60], Reno [61], NewReno [50], and Cubic [51] use packet loss as a sign of congestion, but losses only happen when congestion has already occurred and queues are already full, leading to increased latency through the network. Later techniques like Vegas [17], BBR [23], and TIMELY [84] seek to address high latency by adapting to packet delay itself as a measure of growing congestion. In the datacenter space, network-assisted algorithms like DCTCP [5] and HPCC [72] react to switch information marked into or attached to packets to build a fine-grained model of congestion.

Regardless of the congestion signal, sender-side CCAs typically follow an objective tied to their own throughput and latency performance: ramp up to use the available bandwidth, preferably while inducing as little queuing as possible. This may be combined with application-level considerations, such as balancing the offered load for multiple different users. Importantly, this is a solitary process. Even if a sender receives information about network state, each sender acts separately, and network-level properties emerge due to the distributed interaction of tens to millions of senders all simultaneously

adapting according to their own algorithms.

1.2.2 Steps towards dual-endpoint control

Over the years, alternatives have emerged to sender-side congestion control that point the way towards a model where multiple network entities combine their policies.

1.2.2.1 In-network techniques

Researchers have considered in-network bandwidth allocation 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 operator knows about but are invisible to the sender and receiver. The canonical example is fair queuing (FQ) [33, 88], 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 [41]. 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 dual-endpoint control provides similar benefits to in-network techniques: expressive bandwidth allocation neither tied to a particular CCA nor application, nor dependent on the sender observing all necessary state.

1.2.2.2 Network-assisted techniques.

Rich queue measurement enabled by in-network telemetry (INT), available in some modern switches, has led to a new class of CCAs. Poseidon [117], PowerTCP [1], and HPCC [72] offer precise convergence by directly measuring congestion at the queues in which it occurs. However, INT features are not yet universally available across switch vendors and hardware generations and require design changes to the sender-side CCA to take advantage of these new signals. Furthermore, the use of INT prevents the use of Generic Receive Offload (GRO) [20, 35] in the Linux kernel network stack, which is key to reducing host CPU overhead (explored in Chapter 4), because each header is different (due to different INT telemetry) and therefore cannot be coalesced [56].

1.2.2.3 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 [107], specifically for web content [49], or at the boundary of rate-controlled networks [65]. More recent proposals such as ExpressPath [27], Homa [85], NDP [53], and pHost [46] 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 dual-endpoint control would be to take an existing receiver CCA and deploy it in parallel with a sender-side CCA, creating full 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 dual-endpoint control on the Internet is to specifically avoid integrating the objective of dual-endpoint 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) [110] 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. Dual-endpoint control, on the other hand, directly communicates a receiver rate decision instead of inducing congestion to force the sender to reconverge. Our approach does not introduce additional latency and does not waste network resources by dropping packets that have already arrived at the receiver.

1.3 Leveraging TCP Flow Control to Enable Receiver Assistance

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 dual-endpoint congestion control, as an extension of the abstraction that the transport layer already provides to applications. This section provides an introduction to receiver window tuning, the simple transport-level technique that we use to enable dual-endpoint control.

1.3.1 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. 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 congestion window (CWND) 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. Memory for TCP sockets is typically not scarce in modern machines (unless an application is slow to read received data), so congestion control, not flow control, typically constrains a flow’s rate. Under normal operating conditions, it is desirable to have congestion control and not flow control constrain a flow’s rate.

1.3.2 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. Our case studies leverage RWND tuning to implement dual-endpoint congestion control. 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 rate control decision. 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 the receiver is overwriting RWND, then the value communicated to the sender may shrink the advertised receive buffer capacity even though a corresponding amount of data has not been received. RFC 9293 [37] 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 adjust the advertised window progressively, at the same rate as data arrives, in the event that the receiver’s rate decision calls for a lower value than what is otherwise transmitted.

In these case studies, we assume that the cooperatively-chosen rate is a minimum of the preferences of both endpoints. Since we are primarily dealing with respecting resource constraints (Chapters 2–4) and holding services and flows to their fair rate (Chapter 5), a minimum is an appropriate example. This is a limitation of implementing dual-endpoint control via flow control, not a design restriction. Section 6.2.1.2 describes alternative approaches to combining the sender and receiver rate decisions.

Since the combination function is a minimum, the final rate is naturally lower than one of the endpoints' suggested rates. Therefore, from the point of view of one endpoint, the final rate decision may yield underutilization. Similar to Customizable Receiver-driven Allocation of Bandwidth (CRAB) [110], we believe that a minor degree of underutilization is acceptable to better preserve both endpoints' objectives, and we evaluate this penalty in each case study.

1.3.3 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) [121] 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 [55] 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 [8] 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 our dual-endpoint architecture, aside from those described in this dissertation.

We take inspiration from work by Spring *et al.* [107] 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. This bears some similarities to our incast case studies, but our general proposal instead ensures more broadly that traffic through the network adheres to the receiver's local policies and constraints. Our techniques do not require knowledge of or observability into the bottleneck and, in fact, each flow may have a different bottleneck.

1.3.4 Types of policies in dual-endpoint control

While we presented the dissertation outline in Section 1.1, another way to view this work is through the lens of policy categories. Dual-endpoint control is a framework for incorporating the receiver into the rate control decision, not one defined policy. A variety of policy categories are possible (the list here is not exhaustive).

1.3.4.1 Resource constraints

The simplest receiver policy is that the rate must adhere to the receiver's resource limitations. An example of a resource-based policy already exists in TCP today: flow control is designed to signal the sender to back off if the receiver runs out of memory to store data that has not been retrieved by the application. This policy was crucial during the early days of TCP, when host memory was scarce, so it is natively supported. Some resource constraints are exposed by certain workloads: if an incast causes congestion, then there is inherently more data to deliver than the receiver can ingest. The resource of network bandwidth must be divided between flows, a process that is typically the responsibility of the CCA through a probing and back-off algorithm. Chapters 2–4 use incast as a motivating example because of its prevalence and the receiver's ability to accurately observe the bottleneck resource. Recently, researchers have explored other receiver resources that become congested and should be part of the rate control policy, such as PCIe I/O pressure [3]. In Chapter 4 we focus on a policy that improves the receiver's per-packet CPU efficiency through flow scheduling.

1.3.4.2 User and application considerations

As typically the client, a receiver is inherently sensitive to the objectives and policies of end users, whether as a service running in a datacenter resolving user requests or Internet users themselves, potentially interacting with multiple services. Users and their applications natural place demands on a rate allocation decision. For example, users want their requests to complete with low tail latency and predictable performance. If users are interacting with multiple services (such as a file sharing service and a video call), they may desire fairness between those services, or prioritization of one over the others. Typically, applications control for these goals by how they map demand to flows, but this is indirect and coarse-grained. At the transport layer, the TCP implementation typically strives for flow-level fairness [54]. In the network, switches might divide bandwidth between users to avoid a noisy-neighbor problem. Chapter 5 explores user-level policies in more detail and shows how dual-endpoint control can represent these policies in an expressive manner. Chapter 6 reflects on opportunities to directly incorporate application-layer policies into dual-endpoint control, as well as the potential for a novel Internet environment created by giving clients a stake in the rate control decision.

Chapter 2

Understanding Datacenter Incast Bursts

2.1 Introduction

To understand the benefits of dual-endpoint control, we study a workload with which traditional single-endpoint control has struggled. While there are multiple workloads for which sender-only (or receiver-only) techniques are ill-suited, we choose to study incast because it is a widely encountered phenomenon that many past designs consider, yet still plagues datacenter operators today.

Modern datacenter networks face many-to-one traffic patterns known as *incast* [5, 25, 87]. Applications (i.e., *services*) are often organized into a partition/aggregate traffic pattern where a coordinator server dispatches up to thousands of sub-tasks to worker servers and waits for their replies [5, 70, 95]. In an effort to improve machine utilization, service architects determine the degree of fan-in based on the processing capabilities of the host CPU, often creating situations in which hundreds or thousands of workers interact with a single coordinator. The roughly synchronized responses from the many workers cause congestion in the coordinator’s top-of-rack (ToR) switch, which absorbs excess packets in queues, increasing end-to-end latency and delaying congestion feedback to senders. Packet loss occurs when switch queues eventually overflow, causing high tail latency that directly impacts service-level performance. At end hosts, these delays and losses interact poorly with TCP’s congestion control and loss recovery algorithms [37], resulting in poor end-to-end performance.

We first present a measurement study that makes two contributions: 1) while incast is well studied, we revisit this behavior for modern workloads in a large datacenter operated by Meta, a major Internet content provider; 2) our observations reinforce conventional wisdom but also uncover trends and characteristics of more recent incast patterns that impact the performance of current solutions. We show that incast often occurs on the scale of hundreds of flows, which is an order of magnitude higher flow count than most TCP-based protocol designs [5, 25, 73, 113, 121] have focused on, and is *bursty*, dissipating in milliseconds, yet still induces significant queuing, packet delay, and retransmissions. This reinforces recent work [47] showing that traffic patterns can experience large-scale incast bursts at millisecond timescales. Our study reveals that for each service, the flow count distribution in an incast is stable, and therefore predictable, both over time and across the hosts in the service. Therefore, rather than *reacting* to incast bursts, as in TCP congestion control, hosts could *predict* the scale of congestion and adjust their rates proactively.

Second, we show using detailed simulations *why* congestion control breaks down in such situations; specifically, we study Data Center TCP (DCTCP) [5], a congestion control algorithm (CCA) widely deployed in datacenters. We show that DCTCP cannot converge to a sufficiently small congestion window (CWND, the volume of data in flight) as the degree of incast increases, resulting in persistently high buffer utilization. We find that the combination of high flow count and the cyclic nature of bursts invites oscillations and unfairness across flows that prevent protocols like DCTCP from converging stably. In-

stead, divergence at the burst boundaries exacerbates the next burst. These measurements imply that sender CCAs are ill-equipped to address incast on their own.

This chapter proceeds as follows. Section 2.2 describes the datacenter environment. Section 2.3 presents a measurement study of production workloads, finding that incast bursts routinely experience hundreds of connections and overwhelm the reactivity of sender-based DCTCP. Section 2.4 uses simulations to diagnose DCTCP’s shortcomings, finding that divergence at burst boundaries exacerbates the impacts of incast. Section 2.5 concludes with reflections on existing techniques and design considerations for future techniques.

The simulation tools and raw data in Section 2.4 are available at github.com/cmu-snap/incast-bursts commit 259f0aa. This chapter describes work that was published in Canel *et al.* [22]

2.2 Measurement Environment

In order to better understand our measurements and observations, we provide some high-level details about our datacenter environment. The Meta datacenter network is described in [47]. Machines are organized into a three-layer topology. Each rack consists of a set of servers connected to a ToR switch using 25–100 Gbps Ethernet links. The ToR switches in our network employ dynamically shared buffers and support active queue management techniques such as Explicit Congestion Notification (ECN) [43] marking.

The servers we measure use DCTCP [5], a network-assisted CCA that has been studied extensively. DCTCP leverages ECN [43], which enables switches to inform endpoints of congestion by setting the Congestion Experienced (CE) bits in the IP header. The receiver reflects these marks back to the sender by setting the ECE bit in the TCP header of acknowledgment (ACK) packets. DCTCP senders adjust their rate proportionally to the fraction of ACKs that are marked. We consider DCTCP for three reasons. First, its focus on keeping queues short reduces drops and improves sharing in shallow-buffered switches. Second, among datacenter CCAs, DCTCP stands out as a mature design that has been implemented in the Linux kernel since 2014. Third, the required switch features for ECN marking are well supported by switch vendors. Recent work describes our DCTCP deployment in more detail [34].

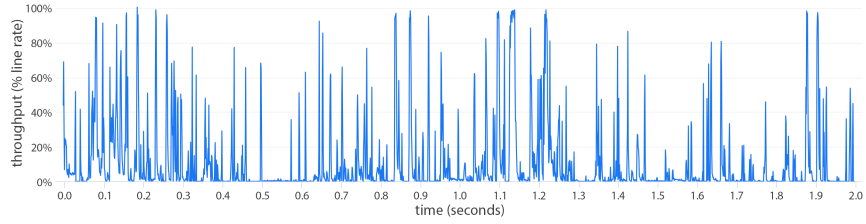
DCTCP uses two tunable parameters, the ECN marking threshold in switches, which controls the amount of queuing allowed before a switch is considered congested, and the g parameter on senders, which controls how quickly its congestion model evolves. For the ECN threshold, we use a higher value than the DCTCP paper [5] recommends to avoid underutilization when faced with host burstiness. Although we avoid reporting a specific value, the chosen ECN threshold is 6.7% of queue capacity. We set the parameter g to $1/16$ based on Equation 15 in [5].

2.3 Measuring Incast Bursts

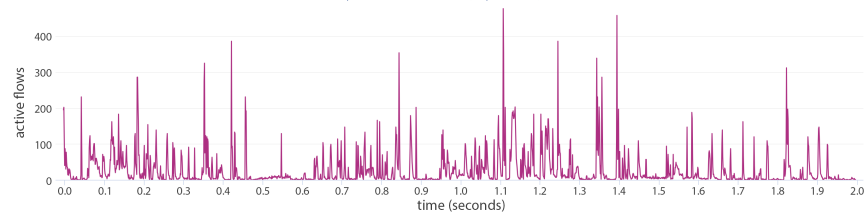
To characterize the behavior of incast bursts in modern workloads, we first show an illustrative example, then present a measurement study of five services at scale to make observations about loss rate, incast frequency and duration, queue length, and ECN marking.

2.3.1 Utilization and loss rate

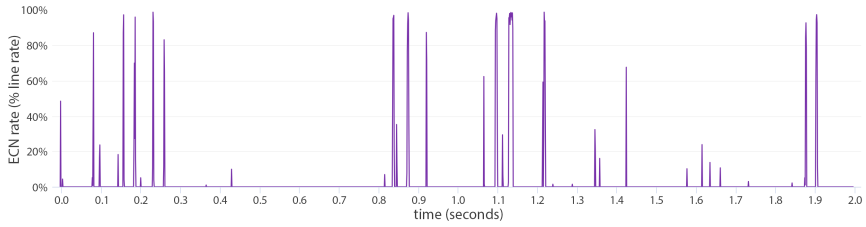
We define a “burst” as any contiguous time span where the average aggregate ingress data rate, measured at the *receiver* at 1 ms intervals, is greater than 50% of the network interface card (NIC) line rate. Figure 2.1 shows a sample two-second traffic trace collected from a popular service (“aggregator” below). Figure 2.1a shows the ingress throughput at the receiver. We see that the traffic has many sharp bursts at line rate, with each burst lasting only a few milliseconds. Average link utilization is low, only 10.6%, with essentially all traffic being part of a burst. This low overall utilization is consistent with recent measurement work [47]. Bursts are associated with high active flow counts (Figure 2.1b), which jump to



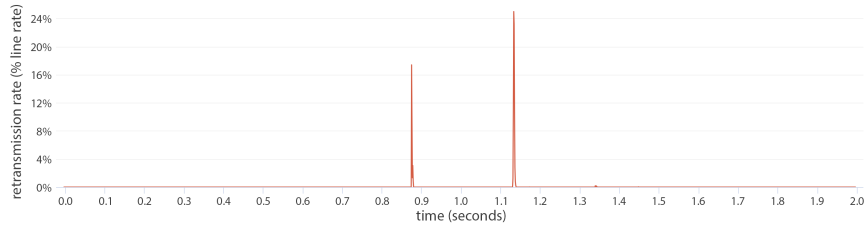
(a) Throughput is dominated by 1–5 ms bursts.



(b) The number of active flows spikes to 100–400.



(c) Packets are ECN-marked, showing queue buildup.



(d) Retransmissions are significant, but rare.

Figure 2.1: Example of incast bursts, measured at a receiver. Figures are time-series showing two seconds, measured every millisecond.

Table 2.1: Five example services.

Service	Description
Storage	Distributed key-value store
Aggregator	Collects content to display on a page
Indexer	Indexing service for recommendations
Messaging	Distributed real-time messaging system
Video	Video analytics service

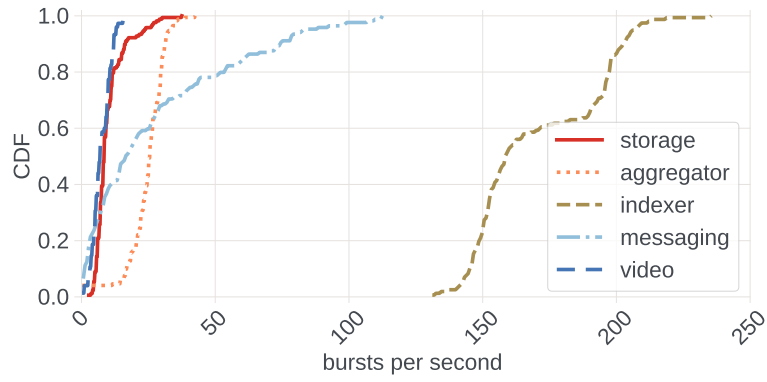
200 or more, signifying that these are incasts. The observation that most traffic is incast bursts motivates why congestion control behavior during incast bursts is important for overall network performance.

Figure 2.1c measures the ingress rate consisting of ECN-marked packets. Recall from Section 2.2 that switches mark packets with ECN when queues are 6.7% full. Here, we see that if traffic is marked, then essentially all packets are marked (marking rate roughly equals line rate). This means that during bursts, sender-side DCTCP will observe either no marking or high marking, leading to oscillations between ramping quickly up and ramping quickly down. Figure 2.1d shows retransmitted packets; those which are dropped by a full ToR queue. Not all bursts cause retransmissions, but when it does occur, loss can be catastrophic and reach 24% of line rate. While isolated, these losses are concerning because they impact application tail latency and therefore user experience.

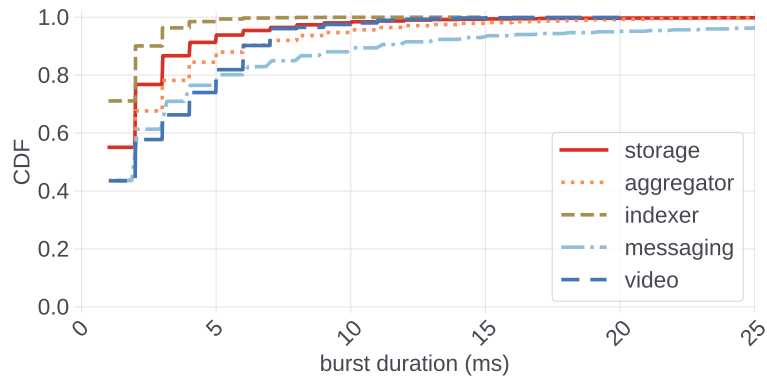
2.3.2 Burst frequency and duration

We show that the traffic patterns described above are common across many services. We observe that compared to existing work, these workloads experience incast on the scale of hundreds, not tens, of flows. We instrument five popular services with different characteristics, described in Table 2.1 and chosen because of their high retransmissions. We collect a two-second trace (measured at 1 ms granularity) from 20 hosts in each service, nine times throughout a day. We collect this data using Millisampler [47, 109], a lightweight host-side measurement tool that runs as an eBPF tc filter [36, 99]. We then identify all bursts, as defined above. We consider the behavior during bursts (ignoring inter-burst periods, where utilization is low), so in the following cumulative distribution functions (CDFs), each sample corresponds to one burst.

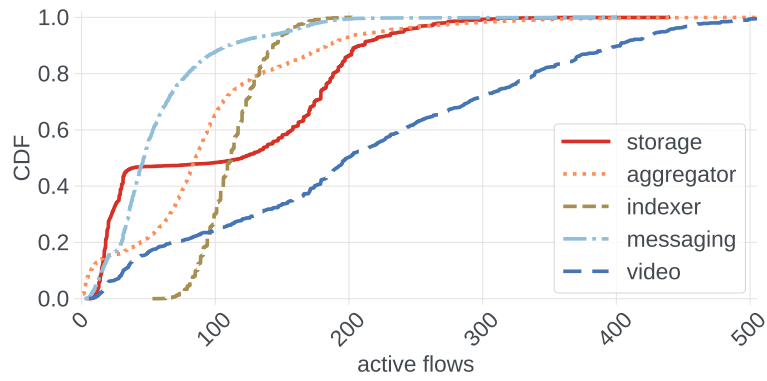
Figure 2.2 measures the extent of incast bursts. We look first at burst frequency in Figure 2.2a: depending on the service, the receiver experiences between tens and as many as 200 bursts per second. Each burst lasts 1–20 ms, as shown in Figure 2.2b, with about 60% of bursts being either 1 or 2 ms. Note that these measurements are at 1 ms granularity, so we cannot detect bursts shorter than 1 ms. Burst duration is in line with recent work measuring bursts in datacenters [47], and is shorter on average than those discussed in the DCTCP paper, which considers queries that complete in 2–40 ms. Importantly, given that a burst of several milliseconds corresponds to tens or hundreds of datacenter round trips, each DCTCP sender should be able to converge to an appropriately low rate. We discuss later why this is not the case. Burst duration informs the type of techniques that may be used to address it: for example, while Swift [71] supports thousand-flow incasts, these short bursts would be challenging for its pacing mode (Section 2.5.2). The bursts observed here are distinct from the *microsecond*-scale incasts described in [125], which are often absorbed by ToR buffers and often complete faster than congestion control can react to. An earlier study of the Meta (Facebook) datacenter network [100] noted millisecond-scale burstiness, but this study goes further to directly quantify the duration, frequency, and impacts of this burstiness.



(a) Frequency: 10s–100s per sec.

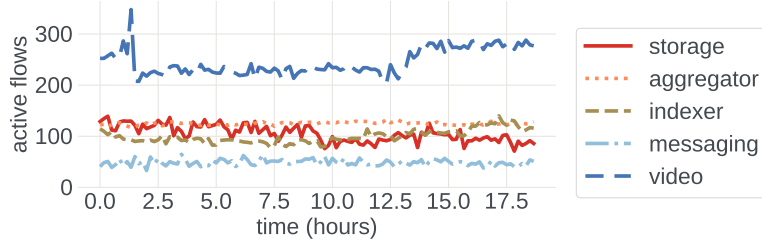


(b) Duration: Bursts last 1–20 ms.

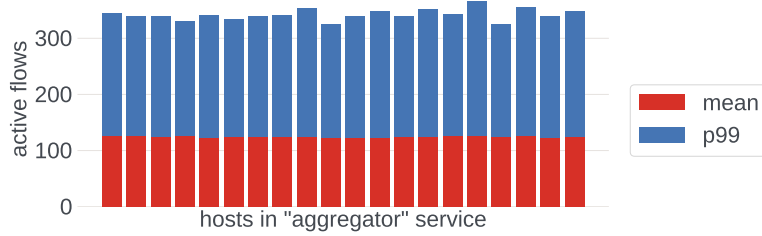


(c) Flows: Incast of up to 500 flows.

Figure 2.2: Incast burst characteristics across five production services.



(a) The average flow count is stable over hours.



(b) 20 sampled hosts in “aggregator” see similar average and P99 flow counts; other services are also stable.

Figure 2.3: Within a service, the distribution of flow count during a burst is stable across time and hosts.

2.3.3 Incast degree distribution is stable

Figure 2.2c shows the number of *active* flows during each burst. Historically, “incast” has referred to any situation where multiple flows converge, however because using multiple flows is standard practice on high-bandwidth datacenter networks and modern CCAs perform well with a few dozen flows, we choose to define an incast with a higher threshold of 25 flows. The majority of bursts in these distributions are incasts (more than 25 flows), with some reaching as high as 200–500 flows in the 99th percentile. The number of active flows is measured over a 1 ms interval, so across the entire multi-millisecond burst, more flows may have been active, just at non-overlapping times. Figure 2.2c also shows cases where a burst is not an incast: “storage” and “aggregator” both show a cliff where between 10% and 45% of bursts experience less than 20 flows. These bursts likely correspond to a bimodal workload where the service performs both a high-flow task like aggregating responses and a low-flow task like checkpointing.

The majority of these incasts consist of many more flows than the $O(50)$ -flow scenarios considered by most existing approaches, such as DCTCP [5], ICTCP [121], D²TCP [113], On-Ramp [73], and [25]. Swift [71] and recent receiver [27, 46, 53, 85] and INT-based [1, 7, 72, 117] techniques do consider hundreds or thousands of flows, but are challenging to deploy due to their requirements for fine-grained timestamping, endpoint stack modifications, or switch features. Other work measuring datacenter bursts either focuses on burst duration but not flow count [13] or considers at most 10 flows [64]. The earlier Meta (Facebook) study [100] revealed that *senders* communicate with up to 300 destination racks (*outcast*), but we investigate the number of flows experienced by each *receiver* (*incast*). The earlier study remarked that measurement overheads prevented the evaluation of incast or microbursts. These challenges were overcome for this work (due in part to the tools described in [47]).

Since flow count defines the scale of an incast, we conduct a more extensive characterization. For each service, we measure active flows on 20 hosts for two seconds at 10 minute intervals over 18 hours. Earlier, Figure 2.2c showed a distribution of flow counts for each service. Figure 2.3 examines how these distributions change over time and across hosts. We see in Figure 2.3a that over time, the *average*

flow count for each service oscillates around steady (albeit different) operating points. An exception is “video”, which shifts between two operating modes of ≈ 225 and ≈ 275 flows, likely due to the service scheduler spooling up more worker hosts in response to load. Figure 2.3b shows that for “aggregator”, selected because it causes particularly high queuing (Figure 2.4a), the average and P99 flow count are stable across hosts as well. While averages hide the variance in Figure 2.2c, the stability in Figure 2.3 suggests that the distributions as a whole change little on the timescale of hours. Stability in the P99 is particularly useful because it indicates the worst-case incast that a service can expect, and these worst-case bursts can have a significant impact on other traffic due to induced queuing delay and drops. Incast solutions can leverage this stability as predictability into how to prepare for future incasts.

2.3.4 Queuing and ECN marking

Figure 2.4 shows the impact of incast bursts on the network. Queue length is a proxy for increased network latency, which applications (and users) often perceive as high response time. The median burst occupies a peak of 20–100% of ToR queue capacity, as shown in Figure 2.4a. To reduce measurement overheads, switches record queue utilization as a “high watermark” over the last minute, which is the highest occupancy the queue achieved over that minute. ToRs enforce a per-queue capacity limit, but the capacity available at runtime may be lower because total memory is shared between ports. The DCTCP paper [5] discusses how flows traversing other ports reduce buffer capacity available to absorb bursts.

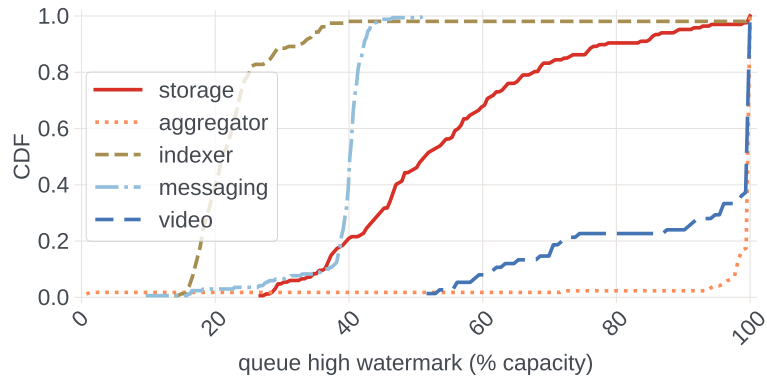
Keeping the ECN marking rate low is crucial for controlling network latency and is also a measure of CCA effectiveness: DCTCP strives for the queue length to oscillate around the marking threshold, so a high marking percentage indicates that DCTCP struggles to converge. Figure 2.4b shows that at the 95th percentile of bursts, between 2.5% and 80% of traffic is marked with ECN. “Aggregator” and “video” experience a P90 marking rate above 60%, suggesting that DCTCP struggles to keep queues short for some bursts. Note that these two services show similar marking rate distributions despite their flow counts differing greatly (Figure 2.2c). In this figure the y-axis starts at the 50th percentile, showing that approximately 50% of bursts do not experience any marking at all, which matches the example in Figure 2.1c where only some bursts cause queues to build. As described in [47], simultaneous burst events to other hosts on the same rack (i.e., rack-level contention) can consume shared switch memory and likely exacerbates a subset of incast bursts.

2.3.5 Retransmissions

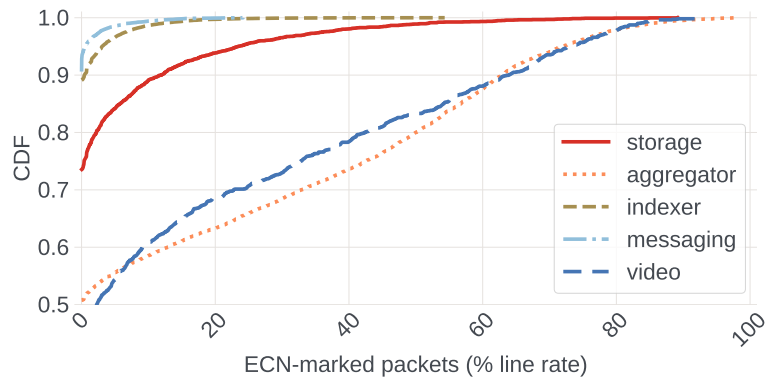
In general, retransmissions occur in only a small fraction of bursts, but when they do occur, they can be significant. Figure 2.4c shows the volume of retransmitted data as a percentage of the NIC line rate. For the profiled services, only 5% of bursts see retransmissions (the y-axis starts at the 95th percentile). The CDF is dominated by its tail: in the top 0.1% of bursts, drops result in up to 8% of data volume being retransmitted traffic. The bandwidth consumed by these retransmissions would be of better use in delivering new data. For services, this long tail of retransmissions is challenging because it leads to high tail latency for queries, although we do not present service-level latency measurements here. These results match with the observations in Figure 2.1d, where most bursts do not cause retransmissions, but when they do occur, they are up to 24% of line rate. These high retransmissions indicate that the CCA struggles to control congestion during incast bursts.

2.4 Understanding Incast Bursts

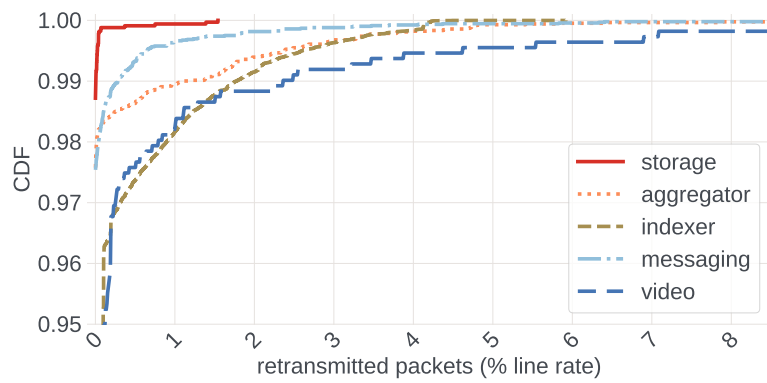
The high retransmission and ECN marking rates observed above suggest that DCTCP struggles during high-flow incast bursts. To understand these shortcomings and how to address them in future techniques, we use a simulator built atop NS3 [92] to reproduce these incasts. In Section 2.4.1, we examine 15 ms bursts, despite this being longer than 90% of bursts (Figure 2.2b), to more clearly visualize DCTCP’s



(a) Queuing: Incasts build long queues.



(b) ECN: 50% of bursts experience marks.



(c) Retransmits: 8% of utilization can be wasted.

Figure 2.4: Negative effects of incast bursts on the network. Note different y-axis ranges.

behavior during incast. We also present results with more common 2 ms bursts in Section 2.4.2. Finally, Section 2.4.3 examines behaviors at burst boundaries that are key to understanding the shortcomings of independent senders using window-based CCAs.

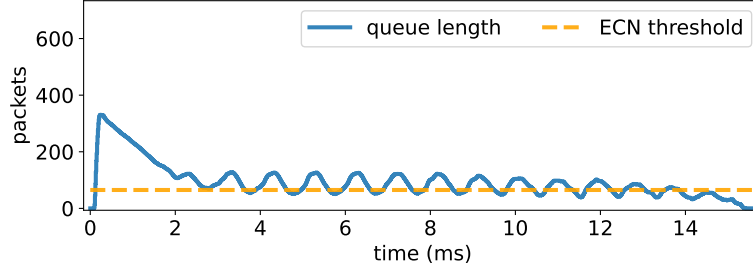
First, we briefly describe the simulation environment. N senders are connected in a dumbbell topology to a single receiver. Incast congestion occurs at the downlink on the receiver’s ToR, so we primarily investigate queue length there. We configure a desired burst duration, then configure all flows with equal demand. Flows use DCTCP and their start times are jittered between 0–100 μ s to model variations in processing time. Each host-ToR link is 10 Gbps and the ToR-ToR link is 100 Gbps. We examine 10 Gbps links because this creates a greater incast potential due to a 10:1 oversubscription with the inter-ToR link, whereas 400 Gbps uplinks and 100 Gbps downlinks, for example, create a smaller 4:1 oversubscription. The maximum transmission unit (MTU) is 1500 bytes. The round-trip time (RTT) is 30 μ s, which is realistic for modern datacenters. The bandwidth-delay product (BDP), the amount of data required to be in flight to keep the network fully utilized, is $10 \text{ Gbps} \times 30 \mu\text{s} = 37.5 \text{ KB}$. We configure switch queues to have a capacity of 2 MB (1333 packets) and use an ECN marking threshold of 65 packets, as recommended by DCTCP [5]. To create the figures in this section, we simulate 11 bursts and report the average performance of the final 10 bursts. We discard the first burst because the more aggressive behavior during TCP slow start [14] results in significant packet losses and timeouts that are not representative of steady-state performance, where persistent connections are used. We disable the commonly used aggregation feature of TCP delayed ACKs [14] because it exacerbates burstiness and masks the impact of DCTCP’s congestion control algorithm.

2.4.1 DCTCP modes of operation

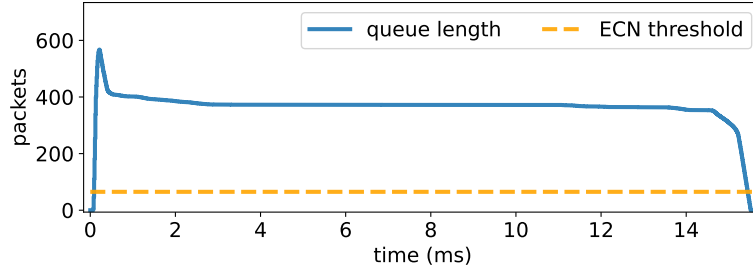
We investigate how DCTCP responds to incasts of different flow counts. While these bursts are short, they do last for tens to hundreds of RTTs, and there is also high ECN signaling, so theoretically there should be sufficient time for DCTCP to reduce its congestion window. However, in practice this is only effective up to a certain number of flows (≈ 150 flows in this configuration). Beyond this, the queue depth is always above the ECN marking threshold because flows cannot back off any further in response to ECN marks. Their CWND is at its minimum value of 1 maximum segment size (MSS). Figure 2.5 shows three distinct operating modes differentiated based on the queuing behavior at the receiver’s ToR.

2.4.1.1 Mode 1: Figure 2.5a | 100 flows | Healthy; periodic

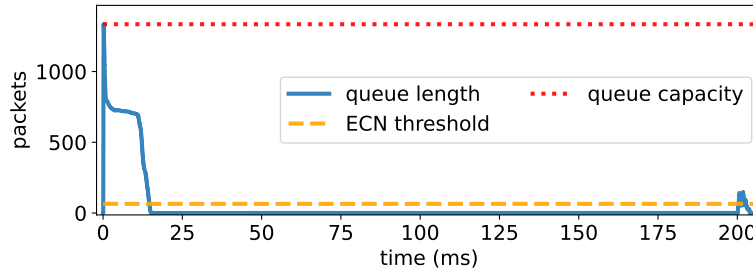
With 100 flows, the queue oscillates around the ECN marking threshold of 65 packets. With a BDP of 25 packets, it takes ≈ 90 packets in flight to trigger ECN marks. Queue depth does not approach capacity (1333 packets) and often falls below the ECN threshold, so DCTCP observes periods of no marking that trigger it to ramp up. Congestion control is functioning, so this mode is healthy. The burst completion time (BCT) is near the optimal of 15 ms. However, DCTCP still causes high queuing and loss in the production network, as shown in Section 2.3, even during Mode 1. First, we observe that queue overflows occur in practice because switch buffers are typically shared between ports, reducing their effective capacity, which the simulations do not model. While each port has a capacity limit, at runtime if multiple ports are sending traffic, then the limited overall buffer memory will be expended long before each queue reaches its individual limit. If the simulations modeled a shared switch buffer, the effective queue capacity would be lower and bursts would experience loss at lower flow counts. Second, we see more queue buildup at the start of the burst due to a subset of flows ramping up. We discuss this further in Section 2.4.3.



(a) Mode 1: 100 flows.



(b) Mode 2: 500 flows.



(c) Mode 3: 1000 flows.

Figure 2.5: DCTCP operating modes during an incast burst that should last 15 ms, in terms of ToR queue length (capacity = 1333 packets). Note different axes ranges.

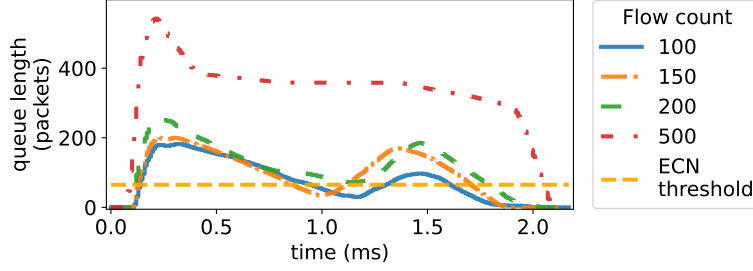


Figure 2.6: Queue behavior during 2 ms incast bursts.

2.4.1.2 Mode 2: Figure 2.5b | 500 flows | Congestion control breaks down; non-periodic

With 500 flows, congestion control breaks down. Even if each flow reduces its CWND to 1 MSS (a single packet per flow), the total data in flight (500 packets) is always higher than the ECN marking threshold (≈ 90 packets, including the BDP). We refer to this state where all flows converge to a CWND of 1 as the *degenerate point*. Beyond this state, senders have no recourse to further control congestion because they cannot back off further, so the queue depth is simply equal to the number of flows minus the BDP. We still see a spike at the start of the burst caused by straggler flows that were able to ramp up at the end of the previous burst (Section 2.4.3). To generate losses in steady state, we must simply increase the flow count K until $K > \text{queue length} + \text{BDP}$. The BCT is still near optimal, but queuing delay is now significant ($\approx 480\mu\text{s}$). We chose a deep queue (1333 packets) for illustrative purposes, but the measurements in Section 2.3 show that with 500 flows losses do occur in practice (e.g., due to buffer sharing).

2.4.1.3 Mode 3: Figure 2.5c | 1000 flows | Timeouts

Increasing the flow count to 1000 flows results in packet losses that incur timeouts due to the small TCP windows in use. TCP’s normal triple-dupACK fast retransmit [14] does not function and losses can only be detected via timeouts, which increase the BCT significantly (≈ 200 ms in this case). Note that the y-axis in this figure extends higher to show the overflow that causes drops. The DCTCP paper [5] notes that its algorithm breaks down during incast once there are so many flows that the first window of data is enough to cause loss, similar to the 1000-flow case described here.

2.4.2 Short bursts, the common case

We illustrated these modes with 15 ms bursts, but this behavior is similar during shorter bursts. Recall from Section 2.3.2 that 60% of bursts are 2 ms or less. Figure 2.6 shows the queue length over time for 2 ms bursts with various degrees of incast. Short bursts are dominated by the queuing spike caused by the flows initially transmitting an entire window at once, and there is insufficient time for the oscillatory steady state behavior shown with 15 ms bursts to occur. This higher fraction of time with deep queues combined with less reaction time before a burst completes means that this common case creates a greater challenge than long-lived bursts.

2.4.3 Divergence at burst boundaries

The cyclic nature of these bursts is particularly challenging for DCTCP. Persistent connections mean that congestion control state is reused from one burst to the next. Unfairness develops within the large set of flows, as follows: Some flows finish early. The stragglers then ramp up their CWND to take advantage of the available capacity at the end of the burst. At the start of the next burst, these stragglers start with a much higher CWND, dominating the other flows and causing a spike in queue length that can lead to loss. We see this queue spike in Figure 2.5, at the start of the burst. Then a new set of straggler flows

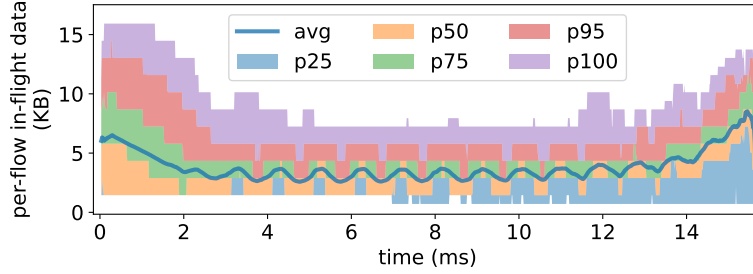


Figure 2.7: Per-flow in-flight data during a 100-flow incast is highly skewed. Stragglers ramp up needlessly, causing high queuing at the beginning of the next burst.

develops, and the process repeats.

Figure 2.7 shows the distribution of per-flow in-flight data over time for the *active* flows in a 100-flow incast (Mode 1 in Section 2.4.1). We see a long tail (P95 and P100) of flows that transmit several times the average and median. At the end of the burst, the average per-flow in-flight data increases as the stragglers ramp up to claim newly available bandwidth. These stragglers have, in effect, *unlearned* the correct CWND for use during an incast. Even though in Mode 1 DCTCP converges to a low CWND and performs well in steady-state operation, the bursty nature of these workloads causes divergence that can build queues, leading to loss and high delay.

2.5 Discussion

This section reflects on key takeaways from these measurements that we believe should influence future incast solutions.

2.5.1 Interactions across bursts

In Mode 1 in Section 2.4.1, where flow count is low and each flow has many packets in flight, DCTCP performs well except for divergence between bursts that arises because stragglers *unlearn* observations made during the burst. While we focus on DCTCP, these challenges apply to independent, window-based CCAs in general. We could modify TCP behavior to explicitly “remember” such observations during incast workloads, e.g., based on the flow count distributions of past bursts. However, this may cause TCP to react less quickly to actual changes in network conditions. Another approach is to tune the CCA’s parameters, such as g in DCTCP, to react more quickly to congestion, but this is brittle and does not address the root cause. Other proposals for centralized [28, 94] and receiver-based [27, 46, 53, 85] flow scheduling address incast with thousands of flows, but necessitate replacing TCP, a significant deployment hurdle. An alternative would be to provide simple guardrails that prevent TCP from ramping up excessively during incast, maintaining responsiveness but limiting TCP’s ability to use available bandwidth. Such guardrails would also limit queue growth during slow start, a key source of loss even at low flow counts. Chapter 3 examines how dual-endpoint control can provide these guardrails by incorporating the receiver’s knowledge of incast properties into the rate control decision using a simple technique that takes advantage of the stable incast properties observed earlier.

2.5.2 Limits to window control

Window-based TCP’s rate control, loss recovery, and efficiency struggle at high flow counts that necessitate small window sizes, as shown in Modes 2 and 3 above. An alternative is Swift [71], which enables $O(10k)$ incast by using a pacing mode that sends one packet every several RTTs. While powerful, this approach has three downsides. First, sending packets infrequently may harm receiver CPU efficiency by

interfering with optimizations such as Generic Receive Offload (GRO) [35] and TCP ACK coalescing [14] that require many packets from the same flow. Second, infrequent probing robs senders of congestion feedback, so a CCA's internal model may be stale by the time it sends a packet. Consequently, pacing is useful only for long incasts: e.g., for 5000 flows, the Swift paper [71] presents a 20-second experiment, whereas our incast bursts complete in milliseconds. Finally, Swift is challenging to deploy because it requires software changes to both endpoints and accurate timestamps from the NIC, which behave inconsistently across NIC models. INT-based [1, 7, 72, 117] and programmable hardware [7] approaches also face deployment challenges. Instead of chasing high flow counts, an alternative approach is to divide, or *schedule*, a large incast into a series of smaller incasts where only a manageable number of flows are active at once. With fewer flows, each would operate in a healthier CWND regime, both for TCP and the receiving host. Such a scheduling technique is reminiscent of earlier work [27, 46, 53, 85], but, due to DCTCP's reasonable steady-state behavior, need only serve as an enhancement rather than a replacement to TCP. Chapter 4 uses dual-endpoint control to build rich receiver flow scheduling that both enables high-degree incast and improves receiver packet processing efficiency.

Chapter 3

A Case Study in Dual-Endpoint Control: Receiver Window Tuning

3.1 Introduction

Now that we understand the scale and properties of incast bursts in production workloads and the limitations of sender-side approaches like DCTCP (Chapter 2), we explore an example of how dual-endpoint congestion control can use the receiver’s observability to signal to senders an appropriate rate for this traffic pattern. Since incast typically produces congestion where flows converge at the receiver’s top-of-rack (ToR) switch, the receiver observes all flows in the incast, has context to understand the likely incast properties of the application, and can better coordinate an appropriate per-flow rate. Instead of senders adapting to ToR queuing from afar, involving the receiver provides the opportunity for senders to react to explicit signaling. Receiver control alone is insufficient, however, as each flow is still beholden to sender-side considerations such as the availability of application data and transmit resources, and crucially depends on the end-to-end model of the network path that the sender CCA has built, which may include bottlenecks unrelated to the incast.

In this case study of dual-endpoint control, each receiver enforces a sender-uniform rate allocation, meaning that it provides each sender in an incast with the same rate decision. This sender-uniform model is analogous to fair queuing (FQ) in network switches [33]. In fair queuing, a switch observes the aggregate traffic volume over a link and provides each flow with a fair share. Here, each flow in an incast is mapped to its corresponding application and given the same rate, similar to an FQ fair share.

The exact fair rate for an application is based on the scale of the incast, the properties of the network, and empirical measurements. The receiver knows its own link speed and (due to other management infrastructure) the location of a flow’s sender (i.e., in rack, in datacenter, and cross-region, each with a separate RTT regime). Therefore, it can estimate the bandwidth-delay product (BDP) of the network. With this information, the receiver can calculate, according to its own policy, a per-flow advertised window that will result in the total in-flight traffic being held to a safe level. The receiver can evolve the per-flow rate over time in response to changing conditions.

However, a key insight is that while it is possible to perform per-flow dynamic window adaptation based on the active flow count and the BDP, our observations about burst properties allow us to adopt a simpler design. Section 2.3.3 shows that the number of active flows is dependent on the service and is consistent over time and across hosts. This uniform service behavior creates an *application burst profile* that maps to the correct rate allocation for the receiver to use, and that allocation is the same across all of its incast senders. Per-flow dynamic tuning may allow for more accurate responses at a fine granularity, but with dual-endpoint control the sender still handles packet-by-packet reactions in response to its model of the network path. Whereas ICTCP [121] dynamically tunes the TCP advertised

window over time in response to the observed flow count, throughput, and RTT, our dual-endpoint approach mitigates incast without this additional control. In contrast to single-endpoint techniques, our dual-endpoint approach demonstrates how the receiver can distill its local observations into a more expressive policy than simply relaying current state to the sender. Our approach is simple to implement at runtime, but relies on assumptions about the predictability of application behavior. In Chapter 4 we relax the sender-uniform model to support larger scale incasts with improved resource efficiency.

Once the receiver has computed its rate decision, we implement dual-endpoint control by overriding the receiver’s advertised window (RWND tuning), a technique described in Section 1.3. By incorporating a receiver rate decision based on the number of flows in an incast, dual-endpoint control provides a lightweight form of centralized coordination that complements existing sender algorithms. Furthermore, the approach of RWND tuning is proactive, not reactionary, so incast bursts are clamped as they occur, avoiding convergence time after a burst has begun. Although we show the effectiveness of this approach with DCTCP, it is independent of the CCA and is specifically *not* a new congestion control algorithm. A benefit of this implementation is that it does not require changes to the protocol or sender. While backwards compatibility with TCP is not a requirement of dual-endpoint control (and in many cases, a clean-slate design would be desirable), this design is amenable to implementation in a production environment. We argue that existing solutions that propose extensive changes to the CCA [71], require switch support [1, 72, 117], or introduce dynamic control systems on the receiver (such as ICTCP [121], which also tunes flow control), are more complex than required and instead propose a simple approach that is practical to deploy at scale.

We implement RWND tuning using eBPF [36], deploy it at scale on up to 7.1% of machines in our datacenters for three years, and show that it is highly effective in our production workloads. By significantly reducing queue buildup during incasts, RWND tuning reduces retransmissions by up to $3.8\times$ and decreases the fraction of bursts that see long queues by 56%. We show that dual-endpoint control is also useful for improving fairness between in-region DCTCP and cross-region Cubic [51] flows, reducing service tail latency by $2\text{--}8\times$. RWND tuning can lead to a 3.5% decrease in average throughput, but this is an acceptable tradeoff given the benefits to retransmissions, queuing, and tail latency.

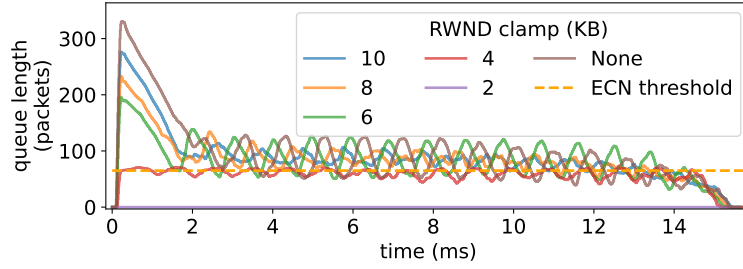
This chapter proceeds as follows. Section 3.2 explores in detail the impacts of a sender-uniform rate policy through simulations of incast bursts, building on the simulations in Section 2.4. Section 3.3 describes practical considerations for implementing receiver window tuning, including how each application’s rate (in practice, a window size) is chosen. Section 3.4 presents a suite of measurements from production workloads showing the benefits of dual-endpoint control for retransmissions and queuing. Section 3.5 concludes with a discussion of the limitations of the sender-uniform rate decision that point the way towards more expressive receiver flow scheduling in Chapter 4.

Note: The RWND tuning system and the sender-uniform policy were independently designed and implemented by our collaborators at Meta.¹ The analysis and evaluation in this dissertation occurred afterwards, during an internship.

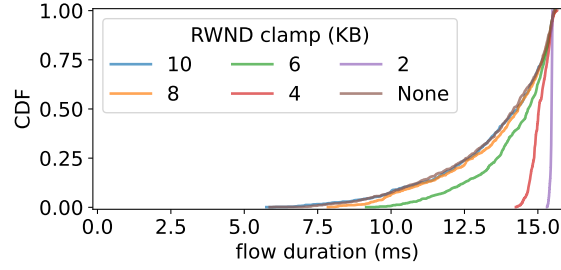
3.2 Proof of Concept in Simulation

We implemented the sender-uniform policy first in a simulator to explore its benefits in detail. In the context of RWND tuning, the receiver’s rate decision is encoded as a window size in bytes that each sender is allowed to have in flight at once. The receiver always overrides the TCP advertised window in outgoing ACKs with a particular policy-defined window size. Figure 3.1 examines the impact of RWND tuning on a 15 ms incast with 100-flows. As in Section 2.4, we discard the first burst to avoid the effects of TCP slow start and average the measurements over the next 10 bursts. Figure 3.1a shows bottleneck

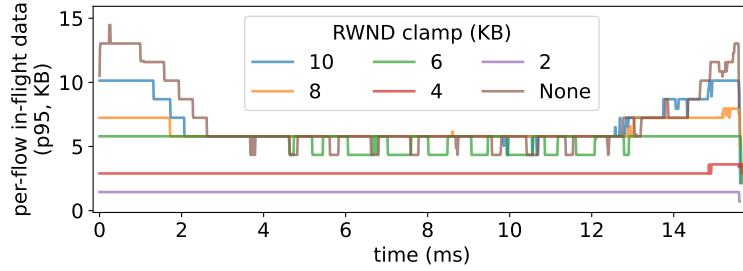
¹Balasubramanian Madhavan, Srikanth Sundaresan, Neil Spring, Prashanth Kannan, Abhishek Dhamija, and many others.



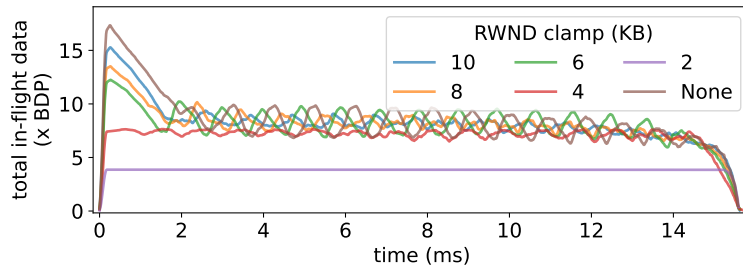
(a) Queue length: Lower clamps reduce queuing.



(b) CDF of FCT: Lower clamps improve fairness.



(c) 95th percentile of per-flow in-flight data: Lower clamps improve fairness.



(d) Total in-flight data: Lower clamps reduce total data in-flight and dampen sender CCA oscillations.

Figure 3.1: Benefits of RWND tuning with various clamp sizes for a 15 ms incast burst with 100 flows.

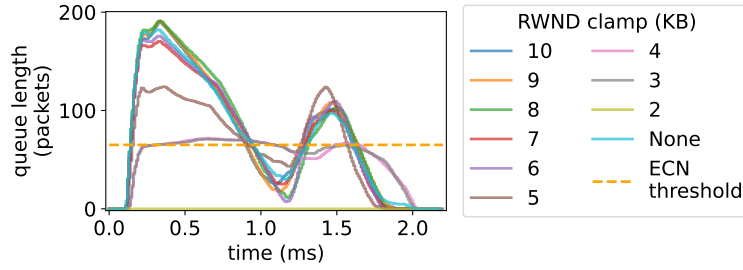


Figure 3.2: Queue length for various RWND clamps applied to 2 ms incast bursts with 100 flows.

queue length over time. Progressively lower RWND clamps reduce queuing, particularly during the first window of data at the beginning of the burst because it prevents the small number of aggressive flows from sending their full window. The implication is improved fairness, since fewer flows get a “head start”. While not pictured, an RWND clamp of 3 KB reduces the queuing to below the ECN threshold, so DCTCP does not receive any ECN marks, meaning it does not detect congestion and does not back off. With a clamp of 2 KB, a single 1500 B packet is in flight at once, which is the slowest rate that a DCTCP sender can achieve.

Figure 3.1d confirms that RWND tuning reduces the total data in flight, in line with the queue results, but also dampens the oscillations caused by sender-side CCAs trying to ramp up due to lack of insight into the incast. At the individual flow level, Figure 3.1c shows that the P95 in-flight data per flow both decreases and becomes more stable, suppressing the inflation at the beginning and end of bursts that leads to unfairness and divergence. Figure 3.1b shows the impact of improved fairness on flow completion time, where we see that low RWND clamps of 2–4 KB lead to almost perfect fairness across flows. A clamp of 6 KB improves fairness as well, but to a lesser extent. Intuitively, a clamp of 4 KB is preferable since it constrains DCTCP senders to approximately the ECN marking threshold, but does not cause the queue to be empty, thus avoiding underutilization.

The queuing reduction becomes more pronounced for short bursts, which are more common (Section 2.3.2) and are dominated by the initial queuing spike. Figure 3.2 shows a 2 ms incast with 100 flows. The overall reduction in queuing with low clamp sizes is more than for longer bursts. Given the burst duration characterization in Section 2.3.2, we believe that the majority of production bursts fall into this short-burst situation. In this example, 3 KB and 4 KB produce similar queuing effects, so we would prefer the higher size to reduce the chance of underutilization.

However, this sender-uniform approach to dual-endpoint control is not a panacea. Beyond the degenerate point where flows already converge to a single packet in flight, as in the 500-flow example in Figure 2.5b, RWND tuning cannot force senders to back off further than one packet per flow, and an alternative solution is needed. (Chapter 4 explores a per-sender allocation scheme that resolves this.) Still, Figure 2.2c shows applicable situations abound in our production network, and Section 3.4 shows that RWND tuning benefits these cases in practice.

3.3 Design and Implementation

This section describes practical considerations for implementing dual-endpoint control via receiver window tuning.

3.3.1 eBPF tc egress

RWND tuning is implemented using eBPF [36], the extended Berkeley Packet Filter, which enables the insertion of verified-safe code into the Linux kernel without modifying the kernel itself. Consequently,

our implementation is portable and not tied to a specific kernel. We use eBPF atop the Linux kernel network stack instead of a user-space networking library due to their robust feature support, ease of development (existing data structures, safety guarantees), and CPU efficiency (eBPF code runs natively in kernel space). At runtime, an eBPF [36] tc [99] egress filter program overwrites the advertised window of all departing ACKs.

All flows to a receiver are subject to the clamp. We configure the clamp on a per-service basis as described in Section 3.3.4, and based on a flow’s scope, which can be *in-rack*, *in-datacenter*, *in-region*, or *cross-region*. In practice, we have yet to find a practical benefit to configuring different rack, datacenter, and in-region clamps, since their RTTs are on the same magnitude. However, when clamping cross-region flows, we must take into account their significantly larger BDP (see case study in Section 3.4.3.2).

RWND tuning is deployed on 24 bursty services in our datacenters, which at peak comprised 7.1% of total hosts.

3.3.2 SACK adjustment

When packets are lost, the receiver likely sends a selective acknowledgment (SACK) [42] specifying that it received, for example, data up to sequence number x , but also y through z . To keep consistent the amount of data permitted to be in flight, we increment the window limit by the amount of data sacked. That is, if we want to keep 64 KB of data in flight for the example SACK above, the receiver will advertise a window of $64\text{ KB} + (z - y)$ with this SACK. We want the sender to continue transmitting new data and keep the flow progressing, as in TCP fast recovery [14]. Not adjusting for SACKs risks (1) making the flow go idle (as the sender has no more data of use to send in the original window) and (2) creating bursts once the retransmission is acknowledged (as the receiver advertises a full window of new packets).

3.3.3 Window scaling

The advertised window field in the TCP header is 16 bits, which would ordinarily only be enough to encode a 64 KB window. To address this limitation, windows are scaled [16]: to be shifted by a value shared with the endpoint during connection setup. Unfortunately, the eBPF tc filter does not have access to the TCP connection state associated with the outgoing handshake segment, and thus cannot tell specifically what window scale value was exchanged with the other endpoint. Fortunately, the value our host sent can be recomputed from the host configuration. Briefly, the window scaling value (typically 11) is computed by the kernel so that the advertised window can advertise the maximum allowable buffer on the host (typically around 11 MB). As long as `rmem_max` is not reconfigured during the lifetime of the connection, it is possible to repeat this calculation from userspace to determine what window scaling value the kernel uses.

3.3.4 Choosing an advertised window clamp

A key question in RWND tuning is how to determine the advertised window size to use, which we refer to as the *RWND clamp size*.

3.3.4.1 Considerations

The goal is not to eliminate all congestion or ECN marking – just prevent excessive congestion – while still relying on the CCA to handle packet-level window control. The intuition is to pick the least restrictive clamp that minimizes packet discards at the ToR without compromising the ability of the senders to saturate the bottleneck link: the ideal clamp is high enough to keep buffers nonempty (i.e., prevent idleness) but low enough to prevent buffer overflow (i.e., prevent drops). Since a relatively low number of services generate the majority of network traffic, the clamp configuration process is performed manually. Configuration takes place once, unless a service’s burst characteristics change. First, we profile a service to determine its flow count over time and its RTT, then calculate an estimated BDP using the

Table 3.1: Static RWND clamps for the five services in Table 2.1.

Service	RWND Clamp Size (KB)
Storage	4
Aggregator	7
Indexer	6
Messaging	6
Video	8

known link bandwidth. Keeping 1 BDP of data in flight is sufficient for full bottleneck utilization, so this becomes the initial target for total in-flight data. The full BDP is divided equally across flows to give a per-flow clamp. Using this as a starting point, an engineer works with the service team to fine-tune the clamp to reduce retransmissions and queue use without sacrificing throughput or negatively impacting service-level metrics like query latency. At runtime, a validation process periodically measures the flow count to make sure that it still corresponds to the profiled range.

3.3.4.2 Heuristic for picking a clamp size

In Figure 3.3, we clearly see a point of diminishing benefits for discard rate as we decrease the RWND clamp from 8 KB to 4 KB. For ease of demonstration, we select a representative clamp size for each service from the parameter sweep of 4–20 KB. To do this, we use a heuristic to find a similar balanced configuration for each of the five services: we score each clamp as the weighted sum of the relative reduction in discards, retransmissions, queuing, and ECN marks, counter-weighted by any loss in throughput due to underutilization, and select the highest scoring clamp. The chosen sizes appear in Table 3.1. These are not necessarily optimal, but they balance the tradeoff between reducing congestion and inviting underutilization and are sufficiently illustrative for an evaluation. These clamps may also differ from those actually deployed at scale on these services: e.g., for the aggregator service the heuristic chose a clamp of 7 KB, whereas in production it uses 8 KB (as stated in Section 3.4.1).

3.4 Production Evaluation

Having discussed the theoretical benefits of RWND tuning in simulation, we present an evaluation on production workloads and demonstrate that this simple technique improves network health with an acceptable throughput penalty. Section 3.4.1 revisits the motivating example in Section 2.3.1 to show RWND tuning in action. Section 3.4.2 examines the benefits of RWND tuning on five bursty services. Section 3.4.3 describes a case study from the deployment process that demonstrates RWND tuning’s effectiveness in preventing incast bursts from harming cross-region flows.

Summary of production results:

1. RWND tuning reduces packet drops by up to 92% and P99 retransmissions by $3.8\times$.
2. RWND tuning reduces peak queuing by 24% on average and reduces the likelihood that a burst experiences ECN marks by 56%.
3. RWND tuning invites an acceptable throughput penalty of 3.5% due to over-clamping when flow count is low.
4. Two examples show that RWND tuning improves the coexistence of in-region and cross-region traffic, causing P99 service latency to improve by $2\text{--}8\times$.

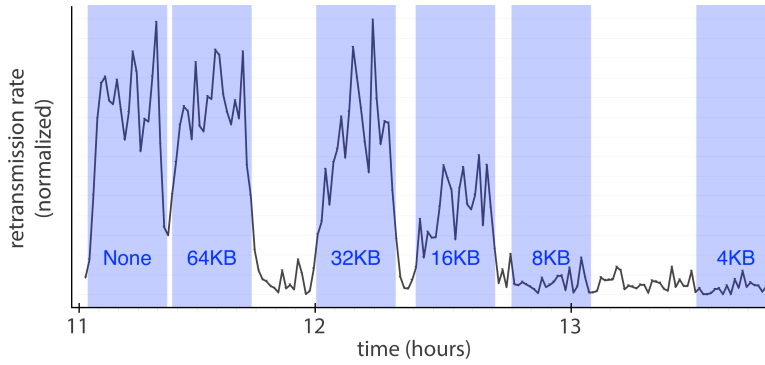


Figure 3.3: Retransmission rate (normalized to max) for various RWND clamps (blue regions). White gaps use the default policy (8 KB) and can be ignored.

3.4.0.1 Network environment

The datacenter environment used for evaluation is the same as described in [22].

3.4.1 Example application

We have seen the benefits of RWND tuning in simulation in Section 3.2, but let us return to the example in Figure 2.1 to see RWND tuning in action for the aggregator service with 100–400 flows. Figure 3.3 visualizes the change in ToR discards over an hour for progressively more restrictive RWND clamps. 64 KB and 32 KB clamps have minimal impact on drops. 16 KB, however, yields a significant reduction, as does 8 KB. 4 KB sees diminishing returns over 8 KB and likely has greater risks of underutilization. In production, this service uses an RWND clamp of 8 KB.

3.4.2 Services in depth

Now that we have seen a simple example of RWND tuning in action, we conduct a more fine-grained analysis across the five example services in Table 2.1. We conduct a parameter sweep of clamp sizes from 4 KB to 20 KB while collecting both network and host-level metrics. Each sample in the following CDFs corresponds to a single burst. Recall that we define a burst to be any contiguous period where the ingress rate at the receiver is at least 50% of line rate.

The following analysis uses two visualizations for each metric of interest: Figure 3.4 shows the overall impact of RWND tuning across the five services, whereas Figure 3.5 shows an in-depth look at how a range of RWND clamp sizes impact a representative service (aggregator).

3.4.2.1 Improved efficiency due to fewer discards

Figure 3.4a shows that across the five services, RWND tuning decreases queue discards at the bottleneck link significantly at the tail: up to 92% at the 90th percentile, from 293 drops/s to 22 drops/s. While forcing senders to back off reduces packet drops, it inherently invites underutilization if the number of active connection falls unexpectedly, which we show in later Section 3.4.2.3 to be an acceptable 3.5% penalty in average throughput. Looking in detail at the aggregator service in Figure 3.5a, simply enabling RWND tuning with any of these clamp sizes leads to a significant reduction in congestion discards at the rack switch. In general, more aggressive (i.e., lower) RWND clamps lead to progressively reduced packets in flight and therefore greater network benefits, with the potential for a higher throughput penalty. Each service experiences a point of diminishing returns that helps us select the appropriate clamp for a service. Note that the ToR drops per second are measured as an average over a minute, so their magnitudes are diluted by the periods between bursts when the link is lightly use (e.g., Section 2.3 gives an example

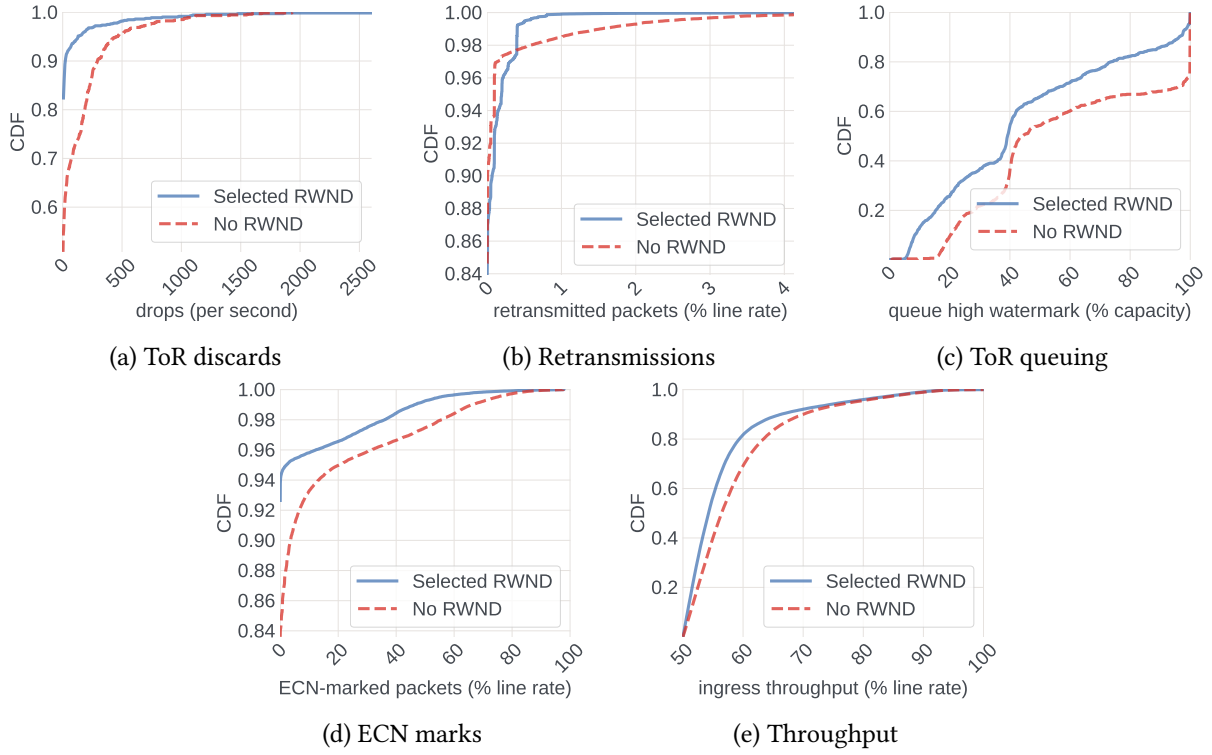


Figure 3.4: Impact of RWND tuning (with Table 2.1 clamps) on key metrics (CDFs) across *all five services*.

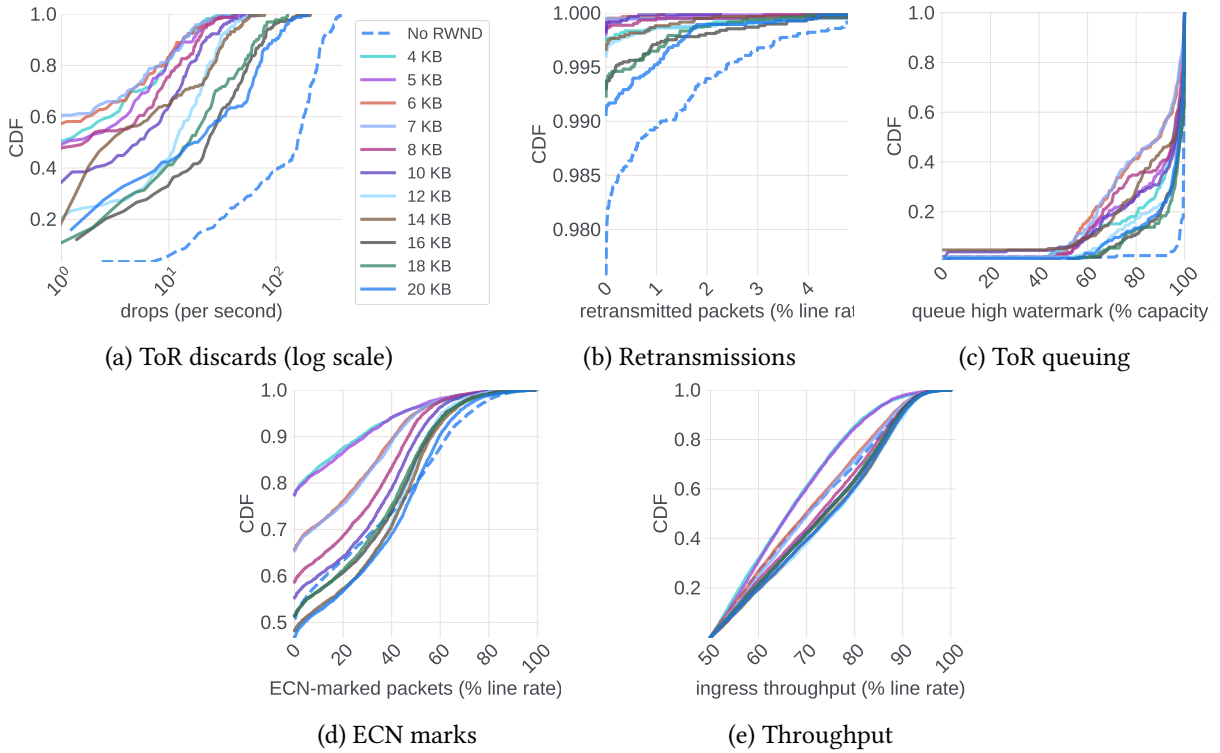


Figure 3.5: Impact of RWND clamps from 4–20 KB on key metrics (CDFs) for the *aggregator service*.

where link utilization is only 10.6%).

Turning to the retransmissions resulting from these drops, Figure 3.4b shows RWND tuning reduces the P99 retransmission rate by $3.8\times$, from 1.52% to 0.40% of line rate. Interestingly, RWND tuning actually increases retransmissions for some bursts at lower percentiles, but significantly reduces the fraction of bursts that experience high retransmissions. For the aggregator service, in Figure 3.5b we see a progressive reduction in retransmissions reflecting the fewer discards in Figure 3.5a.

3.4.2.2 Reduced queuing and congestion signals

Figure 3.4c shows that the RWND tuning reduces the queue high watermark, which is the greatest queue depth measured over a minute. The reduction in average queue high watermark is 13.9%, from 58.2% to 44.3% maximum capacity. Note that drops occur at multiple occupancy points because buffer memory is shared between ports. Shorter queues mean lower latency, and more buffer capacity to absorb (microsecond-scale) bursts. Looking at the aggregator service specifically, Figure 3.5c shows how lower RWND clamps lead to less queuing.

Since high watermark reports a max over one minute, it gives no insight into the behavior over that minute. Instead, ECN marking is a better proxy for queue occupancy. Figure 3.4d shows how the ECN marking rate decreases with RWND tuning. In fact, RWND tuning raises the fraction of bursts that see no ECN marking at all from 84% to 93%. However, the average ECN marking rate across bursts that see marks remains similar, rising from 18.3% to 19.2% line rate with RWND tuning. This means that if a burst experiences ECN marks, in either case, it is likely to see the same marking rate, but with RWND tuning a burst is 56% less likely to see any ECN marks at all. In other words, this more than halves the number of bursts that experience long queues, where “long” is defined as past the ECN marking threshold. For the aggregator service, Figure 3.5d shows that the selected clamp of 7 KB results in a decrease in average ECN marking rate by almost half, from 20% to 11% of line rate.

3.4.2.3 Acceptable throughput penalty

The downside to RWND tuning is that because it causes senders to slow down, it can result in underutilization of the network if the number of connections drops unexpectedly. This leads to a small overall throughput penalty, on average 3.5%, as shown in Figure 3.4e. From the standpoint of our applications, this penalty is acceptable given the tail latency benefits from significantly reducing congestion discards. Interestingly, Figure 3.5e shows that for the aggregator service, RWND tuning can increase the average throughput in many cases. For the selected clamp of 7 KB, the average throughput is unchanged.

3.4.3 RWND tuning prevents negative interactions with heterogeneous traffic

Datacenter networks are not a homogeneous environment. The network must transmit a mix of intra-datacenter (or *in-region*) flows with short RTTs (20–100 μ s), as well as inter-datacenter (or *cross-region*) flows with long RTTs (up to hundreds of milliseconds). Flows with short RTTs can claim more bandwidth because rate-increase logic is triggered per-RTT, and therefore occurs more frequently. Furthermore, in-region and cross-region flows may use different congestion control algorithms, with in-region traffic taking advantage of new approaches like DCTCP [5], Swift [71], or HPCC [72] while cross-region flows use legacy Internet variants like Cubic [51] or Vegas [17]. It is well known that unfairness between CCAs is prevalent [118]. Together, this heterogeneity of RTTs and CCAs presents an additional challenge in mitigating incast congestion.

3.4.3.1 In-region bursts cause loss for cross-region traffic

In-region incasts coexist with long-RTT cross-region flows. In our network, cross-region flows use the Cubic [51] variant of TCP, which uses loss to detect congestion. Due to the inherent nature of TCP favoring flows with smaller RTTs, we observed that the long-RTT Cubic flows can be starved in situations

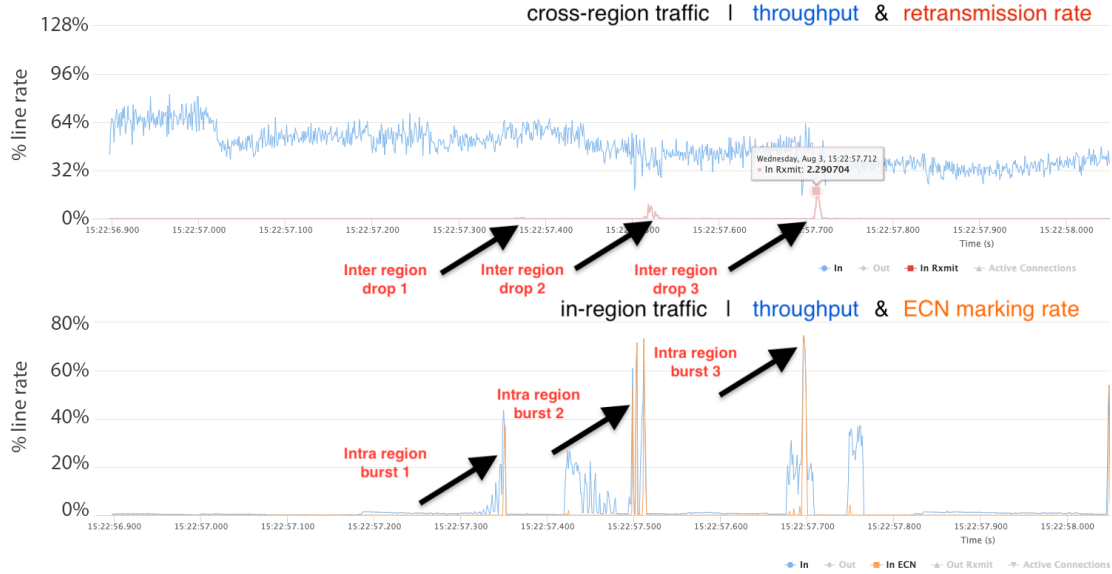


Figure 3.6: Throughput (blue), retransmission rate (red), and ECN marking rate (orange) for in-region (lower graph) and cross-region (upper graph) flows. Bursts on in-region flows cause retransmissions for cross-region flows.

where their demand is low, despite the nature of Cubic to fill buffers and even though the contending short-RTT flows use less aggressive DCTCP. This becomes evident when the same host attempts to read data from its local region as well as multiple remote regions, a traffic pattern that is common in search and storage services where data is read and written to both local shards and remote backups. If the in-region reads are bursty, as shown in the lower portion of Figure 3.6, we observe that their bursts congest the ToR queue (as shown by increased ECN marking) and eventually cause the cross-region flows to experience loss. In the upper portion of Figure 3.6, we see cross-region retransmissions follow shortly after each in-region incast burst.

After profiling the in-region service, we deployed an RWND clamp of 16 KB to the in-region flows and immediately observed a reduction in cross-region latency and drops, which confirmed that these were in fact caused by high buffer utilization from the incast. RWND tuning improved the peak network utilization of cross-region flows by 15%, reduced P99 latency for cross-region requests by $2\times$, and increased read/write reliability from 99.5% to above 99.9%. Used in this way, RWND tuning is a powerful tool for enforcing fairness across scope-based traffic classes.

3.4.3.2 Clamp bursty cross-region Cubic flows to improve service tail latency

Above we focused on the impact of in-region incast on cross-domain traffic; however, we observed that bursty cross-region traffic can also impact service performance. We conducted a preliminary exploration of using RWND tuning to mitigate this issue as well. An important challenge we encountered is that RTT unfairness poses a significant problem for cross-region flows that experience much greater RTT variation (10–100 ms) depending on the geographic distance between datacenters. Therefore, instead of setting a static clamp like with in-region flows, we set a static *target rate* and compute at runtime the appropriate receiver window to achieve that rate based on the observed RTT. This provides rough fairness between cross-region flows with different RTTs.

Use of this preliminary design on a bursty cross-region storage service running Cubic showed promis-

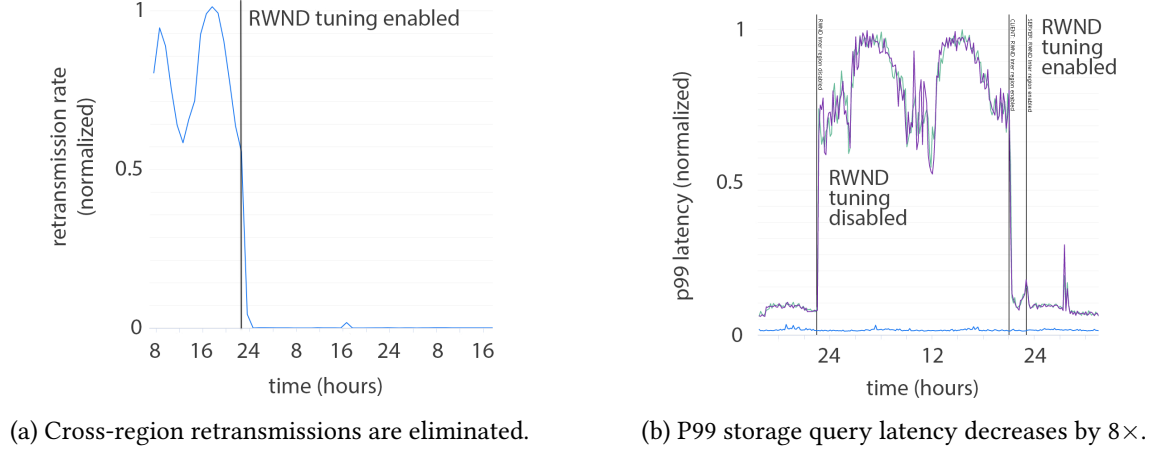


Figure 3.7: Benefits of RWND tuning applied to a bursty cross-region storage service.

ing results. Figure 3.7a shows a normalized timeseries of cross-region retransmissions, and marks the point at which cross-region RWND clamping was enabled. We can see that cross-region retransmissions are almost entirely eliminated. This drop in retransmissions had a positive benefit for application tail latency. Figure 3.7b shows a timeseries of P99 latency measurements for a service that has the cross-region clamp enabled.² In the middle of the figure is a time-span where we disabled the cross-region clamp to observe the baseline P99 latency. We can see that during the periods where RWND tuning is enabled (start and end of the figure range), the service shows an $8\times$ reduction in P99 latency. Since tail latency directly impacts user experience and engagement, this reduction was a significant benefit for the application.

3.5 Conclusion

The combination of evolving datacenter applications and hardware advancement continues to push the limits of congestion control into more demanding situations. Our hope with this work is to propose that for certain cases like bursty incast, dual-endpoint congestion control provides a useful counterpart of existing sender algorithms, without the need for a new CCA. Through an in-depth evaluation of production workloads, we show that simple RWND tuning provides the coordination necessary to keep independent senders in a safe operating regime for sender-side congestion control. Relying on predictable incast properties enables the receiver to reuse the same rate decision for all senders in an application.

However, this approach of applying a single state summary across all senders has two limitations. First, it assumes that all senders behave the same and, in particular, the same across fine-grained time intervals. If an incast contains more or less flows than the application burst profile, then the chosen per-sender rate may lead to congestion or under-utilization, respectively. Even if the incast matches the expected scale, at fine-grained timescales some senders still might not have data available to send, leading to under-utilization. This may be caused either by an application that delivers data intermittently or due to some flows finishing a burst earlier than others. This is an example of sender state that the receiver is not aware of. These effects likely contribute to the 3.5% throughput penalty seen in Section 3.4.2.3.

The second limitation is that using a uniform summary across all senders means that all senders will be active at once. This is reasonable for the scale of incasts observed in Chapters 2 and 3, but at

²Note that Figure 3.7b show a different time range than Figure 3.7a because these measurements come from different experiments.

higher flow counts (e.g., thousands of flows), the uniform rate where all senders can be active is too small to be practical. Typical TCP congestion control has a minimum of one packet in flight, which for thousand-flow incasts may still overwhelm switch buffers. As described in Section 2.5.2, techniques that pace packets at even slower rates face challenges in measuring the network and incur penalties for host efficiency. This is analogous to fair queuing with high numbers of flows: as the number of flows increases and the per-flow rate trends to zero, at some point the amount of useful work accomplished by each flow becomes negligible. Moreover, such a high degree of multiplexing has detrimental effects on other parts of the system, namely the receiver's packet processing efficiency.

In the next chapter, we relax the assumption that the receiver provides the same rate decision to each sender. Instead, the receiver will trade off short-term unfairness for greater efficiency, both in terms of the ability to support much larger incasts but also in the host resources expended to achieve line rate.

Chapter 4

Receiver Flow Scheduling

4.1 Introduction

The previous chapter demonstrated how dual-endpoint control can incorporate the observability and objectives of both endpoints through a case study with a simple receiver policy: a sender-uniform rate allocation that holds hundreds of flows in an incast to an aggregate rate that does not overflow switch buffers. However, the suitability of the sender-uniform policy depends on the workload characteristics at the receiver, and this chapter shows how the characteristics of high-degree incasts reveal two drawbacks with the sender-uniform design. First, to return to the fair queuing analogy, dividing a resource between progressively more entities eventually reduces the per-entity allocation to an extent that each entity no longer makes useful progress. In the case of congestion control, techniques like bandwidth probing and loss detection break down when the sender rate trends to zero. There are practical concerns as well: a TCP flow cannot have fewer than one packet in flight. Second, network bandwidth is not the only resource that the receiver must arbitrate. Its CPU and memory are finite as well, and as the degree of multiplexing increases these resources are put under strain. To demonstrate the benefits of a dual-endpoint approach across a broader set of policies, this chapter proposes an alternative to the sender-uniform model that better supports the full suite of receiver considerations in a high-degree incast.

High degrees of multiplexing have unintended side effects for hosts. Recent papers have brought to light that host processing stacks can be significant performance bottlenecks [2, 3, 20, 103]. In light of these observations, we explore the impact that incast (and other highly multiplexed workloads) have on host efficiency. We find that the Linux network stack and its aggregation features have been carefully optimized over time for high throughput. However, these optimizations, such as Generic Receive Offload (GRO) [35], which combines contiguous packets into large segments, often fail to achieve the same efficiency gains under highly multiplexed workloads because the low per-flow rate does not provide enough locality for aggregations to be effective. In addition, other parts of the system, such as the memory hierarchy, suffer from efficiency issues due to the size and churn of the working set. In this chapter, we describe and quantify these efficiency issues.

This chapter demonstrates how to address these efficiency challenges by using scheduling to reduce the degree of multiplexing. Existing centralized [28, 94] and receiver-based [27, 46, 53, 85] scheduling techniques do not incorporate the objectives of both endpoints: even though we are discussing receiver efficiency, this does not negate the considerations of the senders, which still must balance their many clients and react to congestion other than the incast. Inspired by this scheduling work, we leverage dual-endpoint control to incorporate a coarse-grained receiver scheduling decision with sender-side policies. Receivers can use this technique to decompose a high-degree incast into a series of low-degree incasts that are safer for the network and the host stack. In our receiver flow scheduling design, receivers provide a loosely-scheduled, application-aware service model, whereas senders still control the packet

stream. This model trades off short-term unfairness for improved efficiency, which is suitable for barrier-synchronized applications where all flows must complete before processing the results. Receiver flow scheduling provides benefits both for hosts and for the network. Hosts make better use of offloads like GRO and can achieve lower burst latency. In the network, switch buffers experience less queuing because senders are once again able to react to congestion, and eliminating timeouts improves bandwidth utilization.

To implement receiver flow scheduling, we extend the RWND tuning mechanisms used in Chapter 3 for the sender-uniform rate allocation. A new userspace scheduler observes requests that cause incast and enforces a policy tailored to the application. Within receiver flow scheduling, policies abound regarding the choice of which flows should make progress. The choice of policy is based on the design of the application. Policies such as time slicing, approximate shortest job first, and application priority are all possible within a dual-endpoint approach. As a proof of concept, we adopt a round-robin schedule based on byte grants. A set of flows are each granted a byte allocation; the receiver tracks incoming bytes, and as a flow finishes its grant, another flow is activated in round-robin order. We implement receiver flow scheduling using a combination of a userspace scheduler that sits between applications and the kernel plus a set of eBPF programs that track grants and install customized receiver windows into outgoing ACKs. An evaluation using a new incast burst traffic generator shows that dual-endpoint control can reduce the completion time of 1000–5000-flow incast bursts by up to 74.4% by reorganizing flows into a model where congestion control keeps many packets in flight and is able to react to queue buildup and loss. On top of these network improvements, we show that scheduling takes better advantage of GRO, reducing CPU utilization by 59.6% on average and up to $10.7\times$ for large incasts due to a $6.9\times$ increase in median coalesced segment size. A smaller working set and better cache locality lead to significant reductions in cache access, improving the CPU’s instructions per cycle.

This chapter proceeds as follows. Having already laid the groundwork regarding the incast burst traffic pattern and the mechanisms of dual-endpoint control in previous chapters, Section 4.2 begins immediately with a characterization of the interactions between incast and the Linux network stack. We present a measurement study showing that host aggregations are diminished during incast. Section 4.3 discusses how the combination of network and end host challenges motivates a flow scheduling solution that changes the network service model, and Section 4.4 reviews existing approaches to scheduling for congestion control. Section 4.5 presents the design and implementation of scheduling using eBPF and a userspace scheduler. Section 4.6 evaluates receiver flow scheduling on a practical testbed, showing improved incast scale and efficiency, and Section 4.7 concludes.

The code for the incast burst traffic generator is available at github.com/rware/cctestbed commit **a53a203**. The code for receiver flow scheduling is available at github.com/cmu-snap/ratemon commit **54d468e**.

4.2 A Host Perspective on Incast

Traditionally, proposals have considered incast from the perspective of its impact on the network, such as switch queue buildup, packet loss, and high latency. We explore these effects in Chapters 2 and 3. In this chapter, we focus on the protocol stack perspective as well.¹ We find that the Linux network stack is optimized for high throughput, but these optimizations can be rendered ineffective by highly multiplexed workloads such as incast, motivating our proposal for a change in the transport layer service model.

¹We focus on Linux because it is ubiquitous, leverages multiple types of aggregations to achieve high throughput, and has been the subject of recent work on host bottlenecks [3].

4.2.1 Role of senders vs. receivers

Both endpoints naturally have resource constraints and implement stack optimizations. However, when discussing incast in particular, their challenges are not equal. During an incast, only the receiver must deal with multiplexing, while each sender transmits only a single flow. Assuming hosts are homogeneous, senders are not a resource bottleneck. Furthermore, the sender is fundamentally more capable of managing multiplexing because it inherently controls low-level transmission and packet scheduling on the wire. Therefore, in this work, we focus on the effects at the receiver, which cannot directly control the degree of multiplexing offered by senders.

4.2.2 Network stack optimizations

Developers have long sought to optimize host software stacks to support ever-growing link rates. To achieve a given link rate, a host has a certain amount of time to process each packet. For a 100 Gbps link with 1500 B packets, this is only 120 nanoseconds per packet. Per-packet processing includes updating internal state tracking reliability, assembling out-of-order data, and delivering data to the application. Most host optimizations involve coalescing or aggregating multiple packets so that the same processing is amortized across a larger data volume. Although the features described below are independent of each other, they complement each other and are typically enabled together. These features are described as implemented in the Linux kernel TCP stack for Ethernet networks, but the underlying techniques are general and can be applied to improve efficiency in userspace network stacks, for other datalink technologies, and for other transport protocols.

4.2.2.1 TSO & GRO

TCP supports segments of up to 64 KB, while the maximum transmission unit (MTU) of the network layer is typically 1500 or 9000 bytes. During transmission on the sender, a process called TCP Segmentation Offload (TSO) divides a large segment into small segments depending on the Maximum Segment Size (MSS) after other TCP-layer processing, often in the NIC hardware immediately before packets are sent. Inversely, during reception on the receiver, a process called Generic Receive Offload (GRO) reassembles multiple contiguous TCP segments back into a single large segment before passing the data to the remainder of the stack. Some NICs perform GRO in hardware, before the small segments are ever sent to the CPU and kernel.

GRO is particularly effective at reducing the receiver CPU overhead. Consider a network with an MTU of 1500 B. A full 64 KB TCP segment is 44 packets on the wire. If none are lost, then GRO can reassemble those 44 packets into one segment. This means that GRO can lead to a $44\times$ reduction in the number of TCP segments that must be processed by the Linux network stack. Of course, GRO itself introduces overhead, either in the NIC or software stack, but this is overshadowed by its benefits.

4.2.2.2 Interrupt coalescing

The NIC communicates new packets to the kernel by triggering an interrupt. To reduce the number of interrupts, the NIC waits until multiple packets are ready before triggering the interrupt. Known as *interrupt coalescing*, this amortizes the kernel's context switch overhead across multiple packets. Interrupt coalescing is user-configurable with thresholds based on the number of received frames and a timeout.

4.2.2.3 ACK aggregation

In TCP, acknowledgments (ACKs) are cumulative: an ACK for sequence number S means that all data until (but not including) S has been received. To reduce the number of individual ACK packets it must generate, the TCP stack may delay sending an ACK for a particular data packet to see if a second data packet arrives soon, in which case it can send a single ACK covering both data packets. This feature,

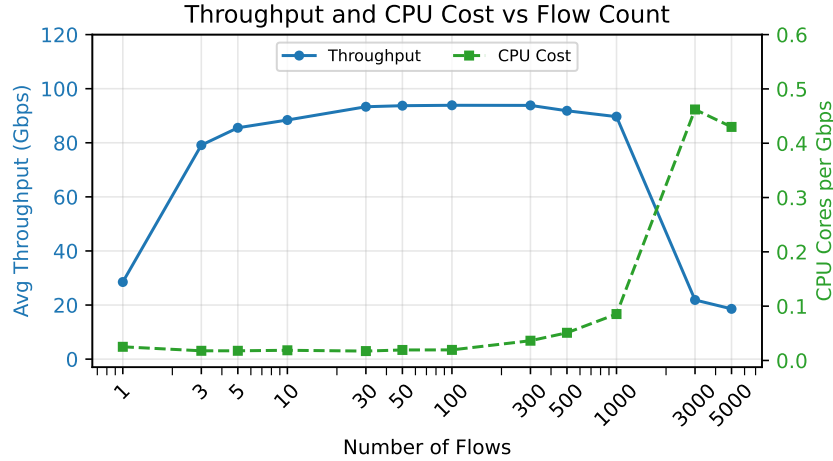


Figure 4.1: Throughput and CPU use per 1 Gbps of receiver throughput, for increasing flow counts.

known as TCP *delayed ACKs*, can reduce the ACK count by half with a small penalty in delaying feedback to the sender. However, this is only effective when a flow receives data packets in quick succession.

4.2.2.4 Jumbo frames

Since host processing typically occurs at the granularity of packets, a simple way to reduce overhead is to make the packets larger. The typical MTU for an Ethernet frame is 1500 B, including the TCP and IP headers and all packet data. This limit is determined by the NIC and the switch hardware on the path between endpoints. Modern NICs and switches typically support MTUs larger than 1500 B, which are called *jumbo frames*, with 9000 B being the most common size. Although jumbo frames do not reduce the total transmission time for a bytestream, which is determined by the total data size and network bandwidth, it reduces the number of data packets (and resulting stack traversals) by six times. Note that for jumbo frames to be effective, the sender must have sufficient data to send to saturate the larger MTU, so this is useful primarily for bulk transfers.

4.2.3 Measurement study of receiver efficiency

This section presents a measurement study of receiver processing overheads. We first look at high-level CPU use and then focus on the efficacy of one of the above techniques: GRO. We end with an investigation of cache performance, another key aspect of host efficiency.

4.2.3.1 Throughput and CPU overhead

Figure 4.1 succinctly summarizes both the network and receiver challenges with large-scale incast. This figure reports both receiver ingress throughput and CPU utilization per 1 Gbps of throughput for incast bursts with increasing flow counts. The network bandwidth is 100 Gbps, the MTU is 1500 B, and the TCP variant is Cubic [51]. In terms of throughput, on a 100 Gbps network, we see that 3–5 flows are typically required for TCP Cubic to reach the line rate. Once the flow count grows large enough that even one packet per flow is too much in-flight data, the throughput drops significantly due to TCP timeouts (here, this cliff occurs between 1000 and 3000 flows).

Let us focus on the utilization of the receiver CPU during incast, which gives a high-level picture of the host efficiency during high degrees of multiplexing. In terms of CPU utilization, this figure shows that with more flows, the CPU “price” of 1 Gbps increases. The rate of increase is more aggressive above 1000 flows, where receiver multiplexing efficiency suffers the most. This points to a fundamental tradeoff

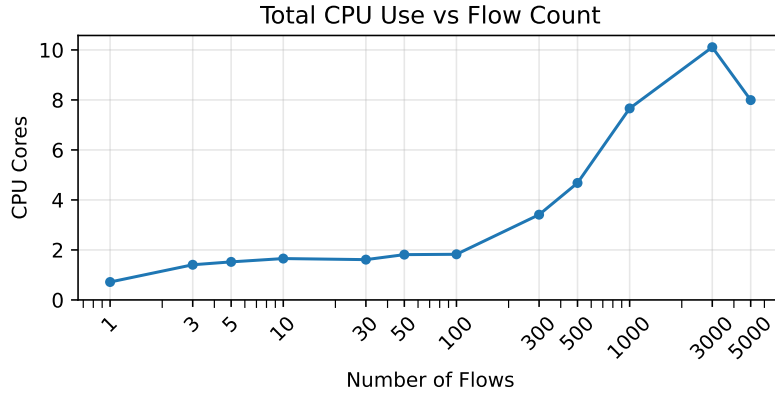


Figure 4.2: Total CPU utilization increases with flow count, but is not a bottleneck (receiver has 64 cores).

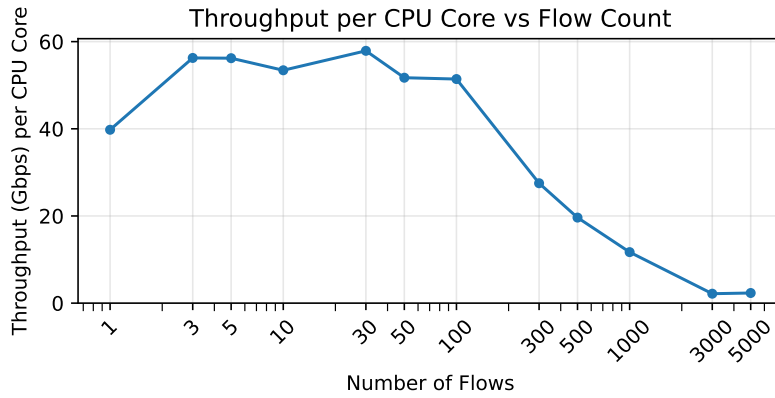


Figure 4.3: Throughput efficiency per unit compute decreases with more flows (inverse Figure 4.1).

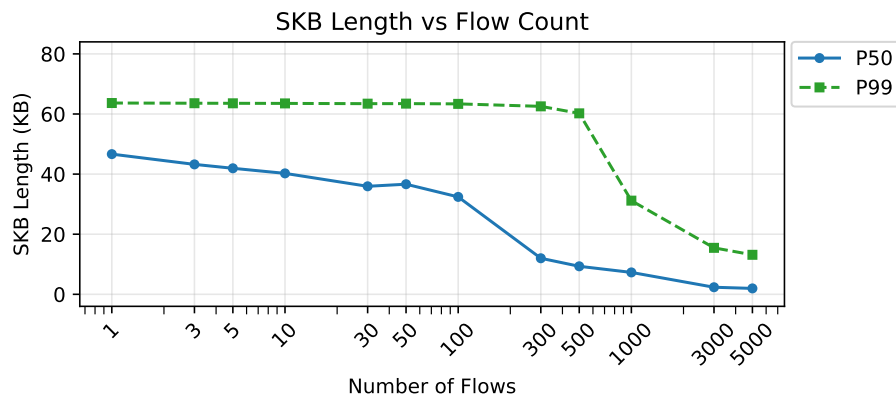


Figure 4.4: The size of the SKBs delivered to the kernel TCP stack decreases as the flow count increases, indicating that GRO is unable to reassemble large segments. Therefore, the receiver must process more individual segments to achieve the same throughput.

between flow count and CPU efficiency. For the same throughput, an incast causes much higher CPU use than a low flow count bulk transfer. As we will show next, this is due to poor aggregation performance that leads to small segments. This being said, the total CPU use is not a bottleneck in this situation. Figure 4.2 shows the total CPU use, which does increase with flow count but at a maximum of 10 cores not a limiting resource on this machine, which has 64 cores (see Section 4.6.1 for testbed details).

The efficiency of the receiver depends not only on the amount of useful work to be done in terms of receiving data and delivering it to applications but also on the distribution of that work across flows, which is a pattern of the workload. In addition to ingress processing, receivers typically distill and organize the data in some way, such as filtering results to return to a user. By using the CPU less efficiently, high-degree incasts rob receivers of resources that are intended for application processing. This is in addition to the network effects of high-degree incast, such as queue buildup and packet drops, as shown in the throughput line and Chapter 2. The next two sections explore host CPU overheads in more detail to uncover the reasons for poor efficiency with high degrees of multiplexing.

4.2.3.2 Efficacy of GRO

Next, we look in depth at one of the most impactful features in terms of managing receiver processing efficiency: Generic Receive Offload (GRO). As described in Section 4.2.2.1, GRO (either in the NIC hardware or in the Linux network stack software) coalesces small but contiguous TCP segments into larger segments, leading to a reduction of up to $44\times$ in the number of segments that the network stack must process. To understand whether GRO is effective at coalescing packets, we measure the size of each segment as it passes through the TCP stack. To accomplish this, we use `bpftool` to attach a `kfunc` callback to the `tcp_rcv_established()` function in the Linux TCP stack. This function is called to process each incoming segment and logs the size of the segment.

Figure 4.4 shows the segment size (median and P99) for increasing numbers of flows. The experiment parameters are the same as above. Looking at the single flow case, the figure shows that the median segment length is 63 KB, meaning that GRO is successfully assembling 44 1500 B packets into near-maximum-sized 63 KB segments (the maximum segment size is 64 KB). Looking at the higher flow counts, the median segment length decreases to 2–7 KB for 1000 or more flows. With one flow, GRO reduces network stack traversals by $44\times$, but with $1000\times$ the number of flows, this reduction is at most $5\times$. The P99 shows that there are still some large segments at moderate flow counts (albeit few), likely due to inherent burstiness in TCP: as described in [103], OS and NIC scheduling combined with window-based control and TSO typically create bursts of packets on each flow. However, at 500 flows and above, the P99 drops towards the median. This correlates with a decrease in the throughput per CPU core, as shown in Figure 4.3, which is the inverse of cores per 1 Gbps of throughput shown in Figure 4.1.

4.2.3.3 Cache performance

Existing work has shown that the performance of the memory hierarchy, both caching and memory bandwidth, can lead to host congestion and be a bottleneck for network traffic [3]. Figure 4.5 shows the number of cache accesses per 1 MB of data received for the L1, L2 and L3 caches as the number of flows increases. We calculated this by measuring both the cache accesses and received bytes, and then correlating the two time series. During this experiment, no other memory-intensive tasks are running. Note the log-scale y-axis. All three caches show an orders-of-magnitude increase in access count as the flow count increases, with a spike at 1000 flows and above. As more flows are active, the total amount of socket state and data that is accessed and, therefore, loaded into cache increases. As the flow count increases, the “cache cost” of each unit of data increases as well. This contributes to the CPU inefficiency shown above, but also by filling the caches with socket state, these incasts rob cache space from other running applications. Note that, in addition to the packet headers and kernel flow data structures, packet

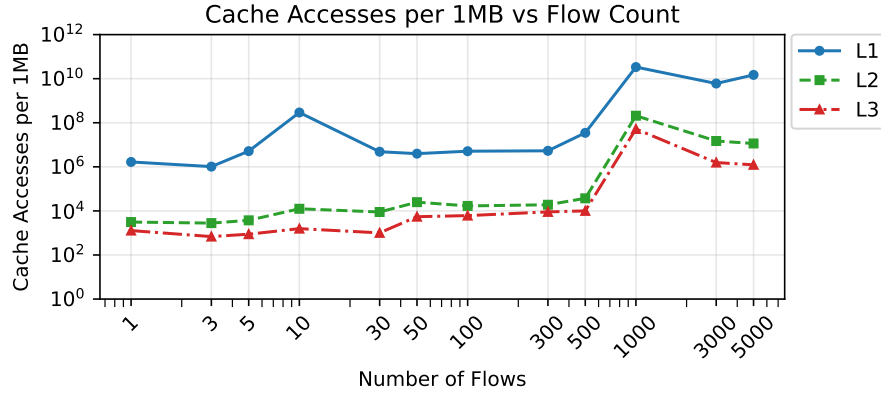


Figure 4.5: L1, L2, and L3 cache access counts per 1 MB of received data.

data must also be handled by the system, resulting in a set of cache accesses associated with the 1 MB of data.

4.3 Scheduling enables large incasts and improved receiver efficiency

As link bandwidths have increased, so has the scale of incast. Early incast work focused on tens of connections [5, 87, 121], whereas modern proposals like Swift [71] and HPCC [72] target thousands of concurrent flows. This trend continues with the rise of heavily parallel applications like training machine learning models on thousands of GPUs, which exchange weights using collective communication paradigms like all-to-all. Our measurements reveal how common techniques designed for efficiency break down in the face of these high degrees of multiplexing. This section discusses how receiver flow scheduling offers a path forward.

4.3.1 Introduction to receiver flow scheduling

This section describes receiver flow scheduling, a case study of dual-endpoint control, and how it addresses the shortcomings in single-endpoint techniques that lack observability into both endpoints' objectives and resource constraints.

The existing network service model faces a key limitation: for incasts with hundreds or thousands of flows, even one packet in flight per flow will overflow switch queues. However, common window-based CCAs cannot send less than one packet per RTT and are not designed to operate with only a few packets in flight. Swift [71] allows senders to converge to an arbitrarily low rate by transitioning from a windowed mode to a pacing mode, but we seek an alternative approach without the CPU efficiency drawbacks described in Section 4.2.

High degrees of multiplexing on a short timescale are fundamentally at odds with optimizations to amortize per-packet processing over large segments. Instead of each flow reducing its rate, we propose the alternative of reducing the number of active flows. To support incasts with thousands of flows and to improve performance even during smaller incasts, a natural solution is to divide an incast into a series of lower-degree, manageable incasts by constraining the number of flows that are active at once. This changes the network service model: instead of the network providing each connection with the opportunity to make progress slowly, we argue that in these scenarios it is useful to adopt a model where (even within the same priority or traffic class) a subset of flows make progress quickly while the rest wait. We describe below how this is beneficial for both the network and the receiving host.

We refer to this technique as *loose* receiver flow scheduling because control is exerted at the gran-

ularity of flows, not packets, and receiver control coexists with the sender CCA, which follows its own algorithm while a flow is active. Instead of explicitly coordinating between senders or adopting a full receiver-side CCA, loose flow scheduling is *complementary* to existing sender-side CCAs. The goal of receiver scheduling is to avoid putting the sender CCA into a regime it was not designed to handle. It is not a full congestion control algorithm, and does not attempt to duplicate the sender’s probing of in-network congestion. The sender CCA is generally built for fairness and preventing starvation and packet loss. Scheduling here is built to make that easier by reducing the number of concurrent flows, introducing temporary unfairness that improves overall completion time and host efficiency. Its purpose is to combine receiver observability with decentralized sender rate decisions that already model the network path. This design distributes congestion responsibility: a sender is responsible for balancing across its clients and “filling the pipe” by growing its CWND, whereas a receiver is responsible for using its visibility into the incast and resource constraints to balance across its senders by dictating which senders should transmit and placing guardrails on their window growth.

4.3.2 Benefits of changing the network service model

Receiver flow scheduling through dual-endpoint control enables the network to support orders of magnitude larger degrees of incast while simultaneously reducing host processing overheads on the receiver.

4.3.2.1 Enabling large incasts

Receiver flow scheduling overcomes the congestion control degenerate point where all flows converge to a CWND of 1 MSS. We previously described this degenerate state in Sections 2.4.1.3 and 2.5.2, and revisit those observations here. Traditional TCP CCAs [5, 23, 50, 51, 71] that rely on probing the network are not designed to operate at extremely low rates with only a few packets in flight, as is necessary to avoid overloading the network during large incasts, for three reasons. First, when CWND is low, any increase in CWND can have a huge impact on total packets in flight in the network: if CWND=1 MSS, a typically conservative additive increase of 1 MSS is actually a 100% increase in the offered load on the network. Across hundreds of flows in an incast, this leads to volatility and oscillations. Second, senders depend on continuous observations of the network to detect congestion and know when it is safe to ramp up, and in standard TCP that feedback (e.g., loss detection, delay measurement, ECN marks) only comes from sending packets and receiving their ACKs. Operating in a low-CWND mode robs CCAs of the signals they need to detect congestion. Moreover, with long switch queues, what feedback senders do receive is stale by the time it arrives. Third, TCP features like triple-dupACK fast retransmit require at least four packets in flight to detect loss, otherwise connections must fall back to timeouts, which cause high tail latency. Section 2.4.1.3 visualizes these pitfalls. By reducing the number of flows, receiver scheduling enables those that are active to operate in a healthy CWND regime with multiple packets in flight, enabling better network probing, fast retransmit, and fine-grained rate adjustments.

4.3.2.2 Improving host CPU performance

Traditional sentiment is that fairness is preferable, i.e., that multiple flows sharing a bottleneck should converge to equal shares of the bandwidth [26, 60]. However, we argue that in a bursty incast, performance may be more desirable than fine-grained fairness. Refactoring the network service model to keep fewer flows active, each with more data in flight, improves receiver CPU efficiency by taking advantage of aggregations like GRO and improving pipelining of network IO and application processing. As network bandwidth increases, receivers rely more on aggregation features in the Linux network stack to reduce CPU overheads. An example is generic receive offload (GRO) [35], which combines packets in the same flow into larger segments, thus reducing the number of network stack traversals necessary to process the same amount of data. For 1500-byte packets aggregated into 64 KB segments, the CPU

overhead reduction can be $40\times$. However, during an incast with hundreds or thousands of flows, each individual flow may only have a small number of packets in flight. With only a few packets per flow, GRO produces small segments (as shown in Section 4.2.3.2), which diminishes its benefits. Receiver flow scheduling, on the other hand, reduces the number of active flows so that those transmitting have sufficient data in flight to take advantage of GRO. This significantly reduces the CPU overhead required to process the same amount of data.

4.3.3 Scheduling algorithms

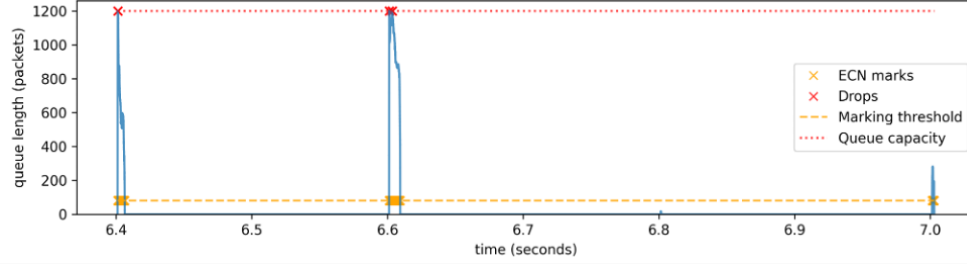
By staggering batches of flows that operate in a healthy CWND regime, the receiver can support incasts with an arbitrarily high number of flows. (Of course, host resource constraints enforce an upper bound on open connections.) How this scheduling plays out depends on the receiver’s policy. Several scheduling algorithms are appropriate for incast bursts, depending on endpoint objectives. Suppose that to balance receiver efficiency, F flows may be active at once.

1. **Round-robin:** F flows are active for an epoch of duration T , where T may be in units of time or bytes. At the end of its epoch, a flow is paused and another is activated. This provides a service model where all flows make progress, but sacrifices fine-grained fairness for improved efficiency.
2. **Run-to-completion:** F flows are active, and when one finishes transmitting (i.e., goes idle), a new flow is activated. This is useful for applications where the sender’s entire message is required before the receiver can start processing it. Allowing some senders to finish quickly improves the receiver’s pipelining of IO and compute. This may cause starvation, so a long-running flow should be preempted to allow others to make progress.
3. **Fewest remaining bytes first:** The F flows with the smallest number of remaining bytes are active. This is useful for lowering the average flow completion time during a bursty incast, but depends on knowing the flow size, which may or may not be available.
4. **Priority:** The F highest priority flows are active. This is useful when there is a stratification in flow priority, such as based on tail latency objectives or traffic class. This entails a cross-layer design in which the application informs the transport layer about flow priority.

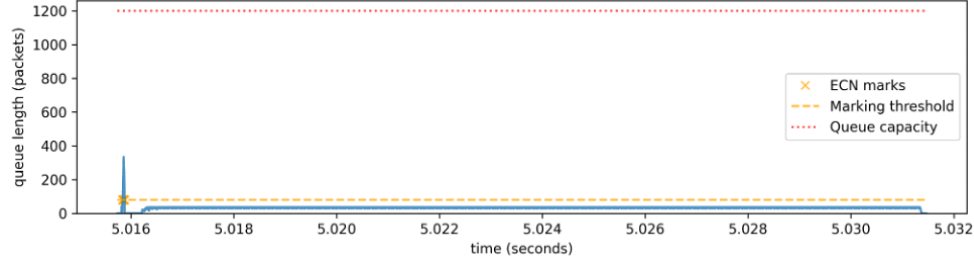
For this case study, we adopt the first option: a round-robin scheduling policy inspired by Homa [85]. During an incast, the receiver allows at most F flows to be active at once, with the epoch duration determined by byte-based grants. When a flow exhausts its grant, a new flow is chosen in round-robin order. The value of F must be large enough so that the utilization remains high during the period between grants. On the other hand, F must be small enough to (1) achieve good receiver processing efficiency and (2) given the bottleneck bandwidth, each flow should be able to grow its CWND to a healthy range for congestion control to adapt to changes in congestion. We use $F = 20$ in our evaluation. T , the per-flow grant size in bytes must be large enough to be efficient but small enough to emulate processor sharing; we use $T = 256$ KB. Section 4.5 provides details regarding the design and implementation of a grant-based round-robin scheduling policy on top of RWND tuning.

4.3.4 Proof of concept in simulation

As a proof of concept, we implemented receiver flow scheduling in simulation. Similar to the analyses in Section 2.4, Figure 4.6 shows bottleneck queue occupancy during a 2000-flow incast burst where the ideal duration is 15 ms. Figure 4.6a shows that with DCTCP, the large number of active flows quickly cause timeouts at the start of the burst. What is worse, even after timeout and loss recovery, enough packets are in flight to cause another round of timeouts. Ultimately, the flows take ≈ 600 ms to finish, but the link is idle for most of that time. Figure 4.6b shows receiver flow scheduling with $F = 20$ flows active at a time. When a flow completes, another is allowed to transmit. Burst time is ≈ 15 ms, and no losses



(a) Without flow scheduling, a subset of flows time out twice and the burst duration is ≈ 600 ms.



(b) With flow scheduling, flows run in small batches and the burst complete in 15 ms. Queuing is minor.

Figure 4.6: Bottleneck queue occupancy during a 2000-way incast. The target burst duration is 15 ms.

occurs. While there is an initial queue spike at the start of the burst, it does not overflow the queue. In this way, the receiver divides an arbitrarily large incast into many small chunks that each execute safely. More queue capacity remains to absorb bursts from other traffic without changing the underlying CCA.

4.4 Related Work

This section reviews the challenges of sender-only approaches to incast control and then provides overviews of two additional topics: scheduling approaches for congestion control, a long-standing research area typically approached through centralized or receiver-oriented control, and studies of host overheads, a field that is growing in importance as high link rates place more demands on host hardware.

4.4.1 Challenges with sender-only incast solutions

During pathological traffic patterns such as bursty incast, the collective mitigation of congestion is a decentralized process where each sender operates independently, without knowledge of the overall traffic pattern, the actions taken by other senders, or the objectives of the receiver. Senders try to balance between their multiple clients, maximizing their individual throughput (e.g., by growing their CWND) until they see a negative signal such as loss [51], ECN marks [5], or high delay [17, 71, 84]. This manifests as oscillations in the load offered, as shown in the earlier simulations (Section 2.4). Although ECN marking in DCTCP and delay measurements in TIMELY and Swift make these techniques more sensitive to queue buildup than purely loss-based algorithms such as Cubic [51], a single flow still strives to maintain a healthy buffer utilization to avoid sacrificing throughput. We recognize the value of these approaches in terms of responding to normal congestion and reducing latency. However, we advocate augmenting them with an additional receiver-assisted mechanism to improve their incast performance.

4.4.2 Scheduling techniques

Researchers have proposed fine-grained flow scheduling techniques from both centralized and receiver-oriented perspectives.

4.4.2.1 Centralized scheduling

In Fastpass [94], a central arbiter controls the transmission time of every packet. Endpoints request capacity from the arbiter, which can precisely control which flows are active. With CoFlow scheduling [28], a set of flows in an incast are jointly optimized by a central scheduler. These techniques give powerful control over pathological traffic patterns. Their algorithms could be adapted to reduce the degree of multiplexing in an incast, similar to our receiver flow scheduling. However, the fundamental limitation of these techniques is that they involve collecting state from multiple entities and making a single decision. Although expressive, the central scheduler inherently cannot capture arbitrary endpoint objectives. We take inspiration from these scheduling techniques, but we advocate for a design where each endpoint makes its own decision and then the decisions themselves are combined. Furthermore, centralized techniques have historically been challenging to deploy because they require a closed network environment in which all entities subscribe to the central scheduler.

4.4.2.2 Receiver-oriented scheduling

Recent proposals such as ExpressPath [27], Homa [85], NDP [53], and pHost [46] involve fine-grained receiver scheduling to achieve high throughput and low latency, but often require switch support like priority queues or traffic shaping logic. These techniques can leverage their visibility into the active flow count as well as receiver resources to throttle permission to send. Similar to centralized techniques, these approaches lack visibility into sender objectives, such as balancing across multiple clients, which is key for holistic workload performance. Incast congestion control for TCP (ICTCP) [121] also advocates for tuning the receiver’s TCP advertised window to force senders to back off during incasts. However, ICTCP adopts a complex, online, and reactive approach that more closely resembles a full CCA, whereas we propose simple, lightweight approaches that sit *alongside* the existing sender CCA.

4.4.3 Non-CCA techniques

Earlier work proposed directly modifying the size of a receiver’s socket buffer to force senders to back off during incast [107]. Since memory is not scarce, we prefer to avoid the risk of discarding received packets or interfering with application behavior. Vasudevan *et al.* [114] focused on shrinking TCP timeouts to detect loss sooner during incast, but this approach involves reacting to loss, while we prefer to prevent loss by keeping queues short. On-Ramp [73] proposes a parallel sender-side mechanism, outside of TCP, that holds packets in response to transient overloads; this separation of concerns is similar to our proposal for independent incast control at the receiver, but On-Ramp operates on the sender (which has poor visibility into incasts) and focuses on only 40 flows.

4.4.4 Host bottlenecks

Recent work has brought to light how the host architecture creates various bottlenecks for network applications. Link bandwidths now approach the throughput of the PCIe interconnect between the NIC and CPU, meaning that PCIe features such as the IOMMU can significantly impact flow performance [90]. To focus specifically on stack optimizations, we configured our testbed so that PCIe would not be the bottleneck by using 100 Gbps NICs with PCIe 4.0, which supports 256 Gbps. Agarwal *et al.* [2] explored how contention in the memory subsystem can significantly reduce the available NIC-to-CPU bandwidth, causing congestion, queuing, and packet drops in the host. The hostCC [3] congestion control algorithm builds on this study and incorporates host congestion signals such as IIO queue occupancy into the rate control decision. These findings are largely complementary and orthogonal to our work on multiplexing.

Cai *et al.* [20] perform host stack analyzes similar to those in this dissertation. Their all-to-all traffic example creates an incast of $24 \times 24 = 576$ flows, which is similar to the scale on which we focus. They also find that GRO efficiency decreases during incast, but observe less of an impact than we find because

they use only one sender host (with many flows), which causes a greater degree of implicit batching than using separate sender hosts. Our work extends these findings to focus on an order of magnitude greater degree of multiplexing, where host performance is significantly degraded by ineffective use of aggregations. They also remark that receiver scheduling techniques can be extended to reduce the degree of multiplexing. Our work makes this real through dual-endpoint control.

4.5 Design and Implementation

This section describes the design and implementation of receiver flow scheduling. We begin by describing a new traffic generator that mimics real-world incast bursts up to thousands of flows. The core of this section is an overview of the flow scheduling design itself, following the lifecycle of a burst. We end with details of the eBPF programs that make up the scheduling infrastructure and review several assumptions made to simplify the scheduling implementation while still showing the benefits of dual-endpoint control.

4.5.1 Incast burst generator

Chapters 2 and 3 evaluate dual-endpoint control using production workloads. To enable a more in-depth evaluation for flow scheduling, we designed and implemented a new traffic generator, named the Incast Burst Generator (IBG), to approximate the incast burst pattern in production workloads. IBG is used for all the experiments in the evaluation in Section 4.6, as well as the motivational studies in Section 4.2.3.

Existing traffic generators such as `iperf` [38] and `flowgrind` [126] face scalability challenges with thousands of flows (e.g., both tools use `select()` [78], which is inherently limited to 1024 flows). We describe IBG here because it provides a concrete application on which to explain the design of receiver flow scheduling. As a proof of concept, certain implementation details are tailored to IBG as described below, but the design of dual-endpoint control is not dependent on a particular application.

The design of IBG is a client-server architecture in which a single receiver (client) sends burst requests, containing a request for a specific data size, to K sender hosts (servers). Each sender host creates F long-lived flows to the receiver, for a total of $K \times F$ flows. When a sender host receives a burst request from the receiver, it transmits the specified data size on each of its flows. This captures the key incast burst pattern of partition-aggregate workloads. IBG will execute a series of repeated bursts, separated by an idle period. This allows us to focus on the effects of bursts, and mimics the incast burst pattern we observe in modern workloads, where a particular host exhibits predictable incast scale and little non-incast traffic (Chapter 2).

IBG allows the user to control the mapping of flows to cores on both the senders and the receiver: the endpoints create T_S send threads and T_R receive threads, and divide their flows evenly between these threads. IBG pins threads to cores on the basis of which NUMA node the NIC is attached to. Both the senders and the receiver use `epoll()` [76] to efficiently manage thousands of threads.

4.5.2 Design overview

Figure 4.7 shows the architecture of receiver flow scheduling. This section describes the design by following the lifecycle of an incast burst. Details about each eBPF [36] program are provided in Section 4.5.3.2.

4.5.2.1 Startup

The receiver maintains a set of flows with current grants, called the *active flows set*, and a queue of waiting flows, called the *paused flows queue*. The receiver monitors a defined port range for new flows. On startup, using dynamic function interposition, the scheduler intercepts the `accept()` `syscall`² [74] and

²In IBG, the senders `connect()` [75] to the receiver, hence the receiver-side scheduler monitors `accept()` [74] for new flows.

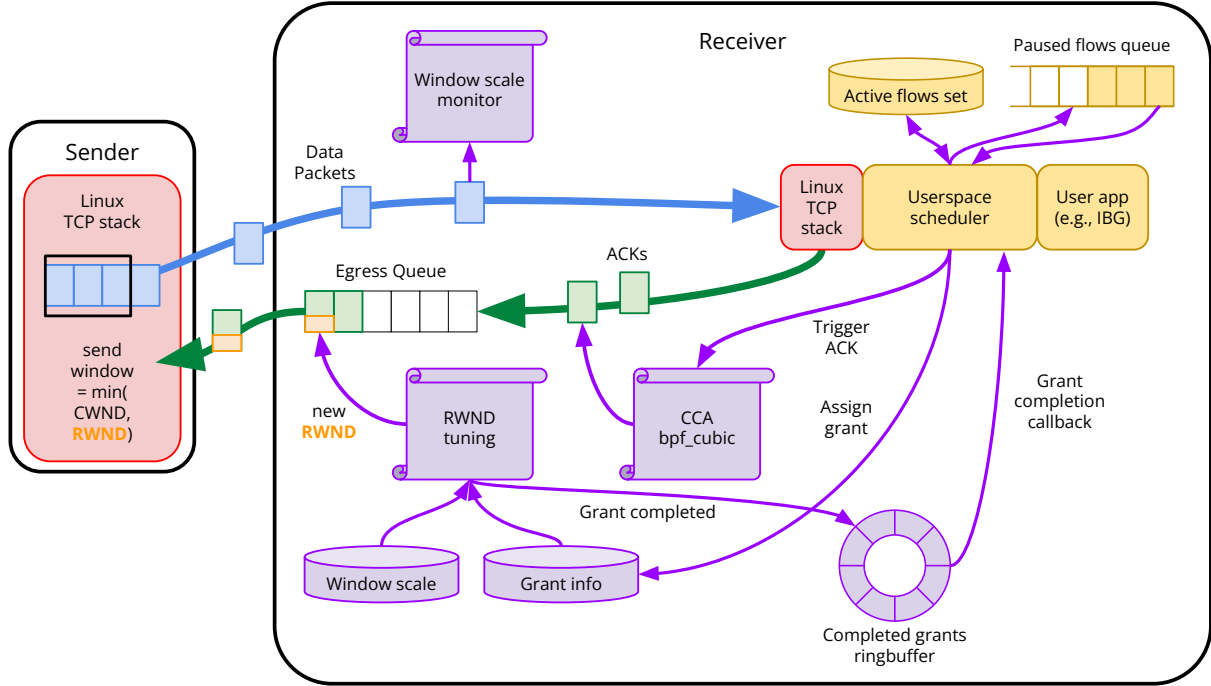


Figure 4.7: The architecture of receiver flow scheduling.

adds the flow to the paused flows queue.

4.5.2.2 New burst request

The scheduler detects when the receiving application issues a request to the senders that will initiate an incast by monitoring outgoing `send()` [79] syscalls. On a burst request, if the size S of the active flows set is less than N , then the scheduler pops $N - S$ flows from the paused flows queue and assigns them grants. On the first burst, $S == 0$ because no pregrants are active (pregrants are described below).

4.5.2.3 Assigning grants

A grant is tracked as the highest TCP sequence number allowed by the grant, called the *grant end sequence number* S_{end} . In this implementation, all grants are the same size: G bytes. If the sequence number of the highest cumulatively-ACKed byte is S_{ack} , then $S_{end} = S_{ack} + G$. Careful bookkeeping is required to handle wraparound of 32-bit sequence numbers. To initiate a grant, the receiver leverages the eBPF CCA to trigger an ACK to communicate a new RWND value. The new RWND value is $D = S_{end} - S_{ack}$, which the RWND tuning eBPF program installs in the outgoing ACK. A separate eBPF program already recorded the receiver window scale W of each flow, and the window scale is taken into account when inserting the new RWND value into the outgoing ACKs – the actual RWND value is $D' = D \gg W$. As new data arrives and is ACKed, S_{ack} advances and $S_{end} - S_{ack}$ shrinks. When $S_{ack} == S_{end}$, then the grant is complete.

4.5.2.4 Pipelining new grants

Since it takes an RTT for the sender to transmit the last packet in a grant, the packet to reach the receiver, and the receiver to issue a new grant to a different flow via a new ACK, there is inherently downtime between grants. The receiver reduces this downtime by preemptively considering a grant done when the ACK sequence number passes a threshold called the *grant end buffer*, B . Therefore, the true grant end

condition is $S_{ack} == S_{end} - B$. This pipelines the completion of one grant with the startup of another, increasing network utilization.

4.5.2.5 On grant completion

When a grant is completed, the RWND tuning eBPF program adds the flow to a ringbuffer of completed grants. Inserting into the ringbuffer triggers a callback in the userspace scheduler, which triggers the next scheduling event. The scheduler removes the flow whose grant completed from the active flows set and pushes it to the end of the paused flows queue. The scheduler pops a flow from the head of the paused flows queue, adds it to the active flows set, issues it a grant, and triggers an ACK, as described above.

4.5.2.6 Tracking burst progress

A flow is only given a grant if it has bytes remaining in the current burst. The receiver tracks the number of bytes remaining in the current burst, R , for each flow and chooses the true grant size G' as $G' = \min(G, R)$. If $R == 0$ when a flow is popped from the paused flows queue, then it is reinserted at the end of the queue. If the scheduler searches the entire queue but finds no flows with $R > 0$, then no grant is assigned and the size of the active flows set drops below N .

4.5.2.7 Pregrants prepare for the next burst

A challenge remains regarding the start of bursts. Previously, we described the behavior for the first burst: select N flows to activate and trigger ACKs. However, in steady state, a burst must stall while the receiver issues these grants and sends the corresponding ACKs. This increases the burst duration. Instead of assigning grants at the start of every burst, the scheduler instead assigns *pregrants* at the end of the previous burst to prepare a set of flows to send immediately when the next burst is requested. Pregrants are applied as follows. For the last N flows in a burst with $R > 0$, when each flow is assigned its final grant in a burst (i.e., when $R < G$), a pregrant of P bytes is applied to increase the size of the grant: $G' = \min(G, R) + P$. In the current burst, the flow will consume the portion of G' corresponding to $\min(G, R)$. The remaining grant of P provides excess RWND that will allow this flow to immediately send the next burst request. Until the full pregrant is completed after the next burst starts, these flows remain in the active flows set. This is why, in the “new burst request” section above, there may already be $S \leq N$ flows already active at the start of a burst.³

Pregrants are designed to improve performance when bursts repeat. If a burst is a one-time event, or if bursts are intermixed with non-burst traffic on the same flows, then an approach that assigns grants at the start of a burst is appropriate instead. If the incast is long-lived, then the scheduler can continually cycle grants with no need for pregrants.

4.5.2.8 Grant bookkeeping

To coexist with standard TCP as implemented in the Linux kernel, the scheduler must take care regarding the size of the advertised window, in particular when the remaining grant is small. The first consideration pertains to avoiding silly window syndrome [30]. If a sender with more than 1 MSS of pending data receives a grant (i.e., an advertised window) of less than 1 MSS, then the sender will refrain from sending until it can transmit a full MSS, meaning that the sender waits for the receiver to send a new grant (in traditional silly window syndrome avoidance, waiting for the receiver to free up buffer space).

The second consideration pertains to the TCP window scaling. TCP uses a window scale factor to support advertised windows larger than the range of the field in the ACK header used for this purpose, which is 16 bits (largest advertised window is 64 KB). The window scale is communicated separately

³ $S \leq N$ instead of $S == N$ because the total number of flows may be less than N .

by each endpoint during the SYN and SYN-ACK exchanges as a TCP header option. The window scale value W is a bit shift, so the smallest window size representable under window scaling is 2^W . Therefore, grants can only be assigned in units of 2^W bytes, and a grant smaller than 2^W cannot be represented. The value of W is chosen by the kernel based on the amount of receive memory assigned to the socket. For this reason, we need to be careful when assigning receive memory: too little runs into obvious limitations with flow control, but too much receive memory leads to an overly large window scale factor that prevents fine-grained grant bookkeeping. We set `SO_RCVBUF` on each IBG socket to 8 MB, which we observe yields a window scale value of $W = 8$, so we can represent grants in units of 256 B.

With these two considerations, if the original RWND value in an outgoing ACK as determined by flow control is D_F , then the full formula for the RWND installed by receiver flow scheduling is:

$$D' = \min(D_F, \max(MSS, 2^W, S_{end} - S_{ack}) \gg W)$$

This calculation is done by the RWND tuning eBPF program at `tc egress`, for every ACK. At the end of a grant, it is possible that the true remaining grant $S_{end} - S_{ack}$ is less than the MSS or 2^W . In that case, this formula will allow “extra” grant to the sender. To keep grant bookkeeping precise, we account for this additional grant with a corresponding increase to S_{end} (if the sender has sufficient bytes left to send).

4.5.3 Implementation details

The implementation relies on eBPF [36] programs and data structures to insert additional logic into the kernel datapath.

4.5.3.1 eBPF data structures

We use three eBPF maps to exchange state between eBPF programs and the userspace scheduler.

- **Grant info.** This hashmap stores a mapping from flow four-tuple to the sequence numbers defining that flow’s current grant, such as S_{end} . The appropriate RWND value for each outgoing ACK is determined dynamically based on this state and S_{ack} .
- **Window scale values.** This hashmap stores a mapping from flow four-tuple to that flow’s TCP window scale value, which is set by the window scale monitor (below) and used by the RWND tuning program to scale RWND clamp sizes before writing them into packet headers.
- **Completed grants ringbuffer.** This ringbuffer allows the RWND tuning program to inform the userspace scheduler when a grant has completed. Inserting an entry into the ringbuffer triggers a subscribed callback in the scheduler, which initiates the selection of a new flow to which to offer a grant.

4.5.3.2 eBPF programs

Receiver flow scheduling uses three eBPF programs to implement RWND tuning and collect information from the Linux network stack.

- **RWND tuning — `tc egress`** This program processes all outgoing packets and overwrites the TCP header’s advertised window field with the flow’s custom RWND clamp size based on the current grant state. This program is installed as a filter on a `tc egress` queue.
- **Window scale monitor — `sock_ops`** This program monitors the TCP handshake to capture the receiver’s window scale value for each flow, which is then recorded in the window scale map. The window scale value is advertised as a TCP header option in the SYN or SYNACK packet. This logic is only executed during the TCP handshake, so it does not impose steady-state overhead.

- **CCA (bpf_cubic) — struct_ops** To pause and reactivate flows, the scheduler must send ACK packets with custom RWND clamp values. The clamp value installation is performed by the RWND tuning tc egress program described above, but the scheduler still needs a method to trigger an ACK when a new grant is offered. It is non-trivial for the userspace scheduler to directly trigger a pure transport-layer ACK on an arbitrary flow at a specific moment. Instead, we leverage eBPF to send an ACK, the details of which follow below.

Both inherent and practical limitations with eBPF make manually triggering an ACK difficult. First, most eBPF programs are inherently reactive, triggered by events in the Linux network stack like packet reception and congestion control logic. The application-level scheduler has no method to directly invoke eBPF programs containing custom functionality like sending an ACK. Also, if a flow is paused, then no packets will be arriving and consequently no eBPF callbacks will be triggered on that flow. Second, in practical terms, the event-based nature and memory safety requirements of eBPF programs place restrictions on which program types can access certain TCP state and invoke other functions. For example, while eBPF does provide a helper function for creating and transmitting an ACK, `bpf_tcp_send_ack()`, that function is only available to `struct_ops` programs. An eBPF `struct_ops` program is a set of functions that override one or more members of the `tcp_congestion_ops` struct, which specifies a set of callbacks that together form a TCP congestion control algorithm. Different CCA variants provide their own implementation of each callback.

In order to take leverage the `bpf_tcp_send_ack()` function, we built a `struct_ops` program that responds to a specific `getsockopt()` syscall, which required a minor modification to the eBPF backend. While most `tcp_congestion_ops` callbacks are event-based, there is one that we can trigger on demand: `get_info()`. This callback returns congestion control state information in response to the syscall `getsockopt(TCP_CC_INFO)`. In other words, the scheduler can make a `getsockopt(TCP_CC_INFO)` syscall and, as a side effect, trigger the `struct_ops` eBPF callback for `get_info()`. We take advantage of this and replace the existing `get_info()` logic, which is unused, with code to send an ACK using the `bpf_tcp_send_ack()` helper function. Unfortunately, eBPF does not expose `get_info()` to `struct_ops`. It is possible to override all of the other `tcp_congestion_ops` callbacks, but not `get_info()`. To remedy this, we made minor modifications to eBPF to enable `get_info()` in `struct_ops`. While this currently requires minor changes to the kernel, adding `get_info()` to `struct_ops` could be made a standard feature in eBPF. To be clear, the custom `get_info()` implementation to trigger an ACK is not a modification to the kernel, but is an eBPF program that is installed at runtime by the scheduler.

Bringing the pieces together, we implemented a `struct_ops` version of TCP Cubic, which we refer to as `bpf_cubic`, that delegates all of the `tcp_congestion_ops` callbacks to the existing kernel TCP Cubic functions with the exception of `get_info()`, which is customized to instead send an ACK. To enable ACK transmission functionality, the userspace scheduler sets a flow's CCA to `bpf_cubic` using the `setsockopt(TCP_CONGESTION)` syscall. Then, the scheduler can immediately trigger an ACK by calling `getsockopt(TCP_CC_INFO)`. We use this approach because it enables precise control over when to send an ACK, does not require manually crafting ACKs, preserves the existing socket abstraction for applications, and requires only minor modifications to the kernel.

4.5.4 Assumptions

The goal of this receiver flow scheduling case study is to show the benefits of dual-endpoint control, not deliver a fully general implementation, so we make several simplifying assumptions.

We assume that the receiver knows the size of the data that each sender intends to transmit in a burst. This is reflected in the design of IBG: the receiver's burst requests include an ask for a specific data size, and all flows transmit the same size. Knowing each flow's data size allows the receiver to grant precisely: the sender will never end up with excess grant. An excess grant at the end of a burst would

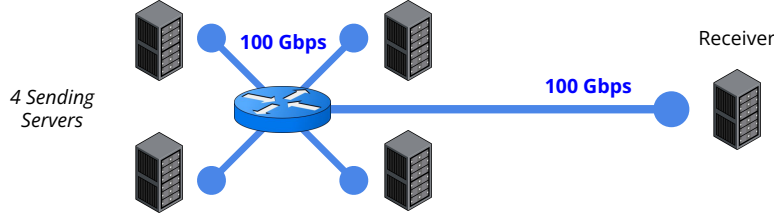


Figure 4.8: 100 Gbps CloudLab incast testbed with four sender hosts and one receiver host. Each sender host uses hundreds to thousands of flows.

allow a sender to immediately transmit at the start of the next burst. If many flows have excess grants at the end of a burst, then at the start of the next burst they will all transmit in addition to flows that have been purposely pregranted. This violates the goal of receiver flow scheduling that only N flows are active at once and potentially overflows switch buffers. In our implementation, the scheduler intercepts `send()` syscalls, deduces which are burst requests outbound to the senders, and parses the request size to know the total size of all grants to give to each flow. Tied to this implementation is the secondary assumption that all flows will send the same data size, but this is not fundamental: the receiver could request a different size from each flow. This also implies that all flows participate in every incast.

The assumption that the receiver knows each flow’s total data size is an example of a cross-layer optimization where dual-endpoint control incorporates useful information from the receiving application into its rate control decision. However, we now consider alternatives should this assumption not hold. One option is to allow excess grants and spurious transmissions at the start of a burst. As described above, this would likely cause congestion at the start of the next burst. Second, we can grant conservatively at the risk of leaving the sender stalled. In this case, the receiver can rely on the sender to transmit zero window probes [37] that will signal it has pending data but no receive window. This may lead to underutilization while the receiver waits for a zero window probe to signal that the sender needs another grant, which might be mitigated by proactively granting to another flow. Third, we can use historical data to estimate the data size, similar to how Chapter 3 uses historical data to estimate the number of flows in an incast. The accuracy of this approach depends on the stability of the data size over time, which depends on application behavior. Misestimation can lead to stalls or congestion at the start of bursts, similar to the other two options.

4.6 Evaluation

This section evaluates receiver flow scheduling to demonstrate that dual-endpoint control produces a rate decision that is better suited to both endpoints observations and objectives. We start with a description of the test environment and receiver flow scheduling parameters. Then we show that dual-endpoint control significantly improves burst completion time for incasts up to 5000 flows, achieving higher network utilization. Finally, we explore receiver-side impacts of high degrees of flow multiplexing and show improvements in CPU and memory efficiency.

4.6.1 Evaluation Setup

We evaluate a practical implementation of receiver flow scheduling based on dual-endpoint control using a Linux server testbed in CloudLab [32]. The machines used in this evaluation are node type R650 [31], which uses two 36-core Intel Xeon Platinum 8360Y CPUs and a Mellanox ConnectX-6 100 Gbps NIC on a 100 Gbps experiment network. Hosts may be located in different racks, which means that there may be at least three switch hops between a sender and the receiver. The MTU of the network is 1500 bytes.

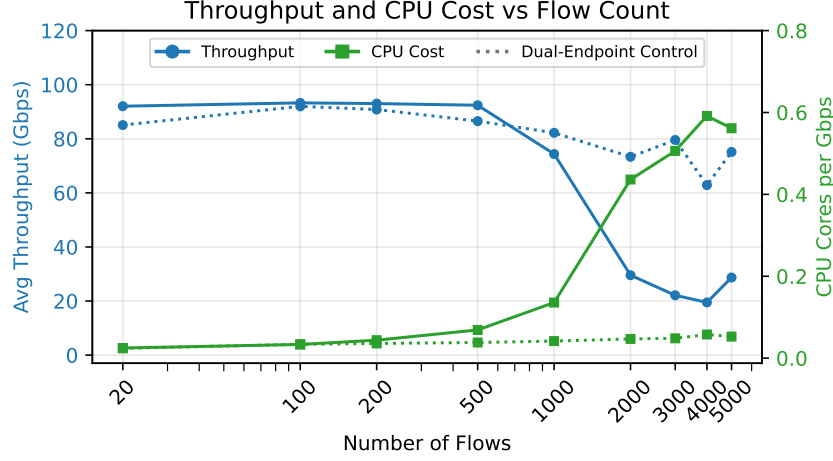


Figure 4.9: Dual-endpoint control benefits both the network (increased throughput) and the receiver host (lower CPU cost per unit throughput).

Host configuration. For improved data locality, we configure the NIC queue IRQs to map to CPU cores on the NIC-local NUMA node. Application threads only use this set of cores. We enable Accelerated Receive Flow Steering (aRFS), in which the NIC steers each flow to the appropriate queue corresponding to the core where the application thread calls `recv()` [77]. We set total TCP receive memory for all sockets to 2 GB to ensure that the network stack is not relying on flow control to reduce the sender’s rate and interfering with the RWND tuning technique that we use to implement dual-endpoint control. As described in Section 4.5.2.8, the per-socket receiver buffer is set to 8 MB. ⁴ Hosts use Ubuntu 24.04 with Linux kernel 6.9.12.

Congestion control algorithm. Chapters 2 and 3 analyze the performance of incast bursts and dual-endpoint control under the DCTCP [5] congestion control algorithm. DCTCP relies on ECN marks from switches to reduce the sender rate before queues overflow. In those chapters, ECN marks were available in the NS-3 simulator [92] and were supported in the production datacenter network. However, CloudLab does not enable ECN marking on their experiment switches and does not provide access to these switches to enable ECN marking ourselves. For that reason, in this section we evaluate TCP Cubic [51] instead. Cubic is a loss-based variate of TCP, meaning that the sender backs off when it detects a packet loss. Cubic is the default CCA in Linux and is common on the Internet. Datacenter traffic may use Cubic for long-distance flows, such as to end users or across datacenters.

Dual-endpoint control is agnostic to the CCA, so using Cubic instead of DCTCP does not fundamentally alter our arguments or results. However, this change does affect the balance of the sender and receiver rate decisions. Since Cubic is loss-based, it will fill switch queues to the point of packet drops, whereas DCTCP will keep queues shorter, converging around the ECN marking threshold. Because Cubic is more aggressive, in our situation of dual-endpoint control, it is the case that more often the final sender rate will end up being the rate decision of the receiver. In the case where there are no packet drops that cause Cubic to reduce the sender-side rate decision, the senders will try to transmit at peak throughput and will only be constrained by the receiver-side rate decision. This means that dual-endpoint control in this scenario behaves similarly to a receiver-side CCA. This shows the flexibility of

⁴With receiver flow scheduling with $N = 20$ flows active and a grant size of $G = 256$ KB, the maximum receive buffer used by all active flows is 5 MB.

dual-endpoint control based on the scenario: with our model of taking the least aggressive rate amenable to both endpoints, in some cases the sender decision may dominate, whereas in scenarios like this the receiver decision dominates. If one of the incast flows encounters congestion elsewhere in the network separate from the main incast, then it may still back off on its own.

Receiver flow scheduling parameters. Receiver flow scheduling has four parameters. (1) The *number of active flows* is set to $N = 20$, which balances having enough flows active to avoid underutilization as grants finish/start with not having too many flows active such that aggregation performance suffers. (2) The *grant size* is $G = 256$ KB, which is four full-size 64 KB segments, or ≈ 181 packets with an MTU of 1500 bytes. (3) The *grant end buffer* is $B = 64$ KB, which means that the grant will be considered complete and a new grant will be issued to a different flow when there is only one full-sized segment remaining in the grant. This reduces the “downtime” between grants by allowing a head start to the ACK carrying the next grant. (4) The *pregrant size* is $P = 256$ KB, a full-size grant, which means that the last 20 flows in a burst will be allowed to send a full grant at the start of the next burst.

Experiment setup. Throughout this evaluation, we look at the relationship between the scale of an incast (flow count) and the workload or host performance. Each experiment consists of 200–5000 flows, sending 500 bursts. The total data size of each burst is 1 GB, divided evenly between the flows. Experiments with more flows use a smaller amount of data per flow. This keeps the expected burst duration the same across all experiments, making it easier to compare the impact of the incast scale and the benefits of dual-endpoint control on the burst duration. The expected burst duration is 85.9 ms. The first 20 bursts are discarded as warmup.⁵

1 GB total burst size divided between 200–5000 flows is a per-flow data size of ≈ 200 KB–5 MB. Grant size (256 KB) is clamped to be no larger than the per-host data size so that a grant does not span bursts. Allowing grants larger than a flow’s data size would result in accidental pregrants for the following burst, but for all flows instead of just those intended for pregrants, leading to congestion at the start of the burst.

Reading figures. We visualize these results as CDFs, where each sample is one of 500 bursts or one measurement of host performance counters. Looking at a CDF such as Figure 4.10, each solid line corresponds to a particular flow count without receiver flow scheduling. A dotted line in the same color represents the same flow count using receiver flow scheduling via dual-endpoint control.

4.6.2 Overview: Benefits for network utilization and receiver efficiency

Figure 4.9 returns to the analysis presented in Section 4.2.3.1 (Figure 4.1) of throughput and CPU cost as the flow count scales to show an overview of the benefits to network utilization and receiver efficiency provided by dual-endpoint control.⁶ Without dual-endpoint control, throughput decreases at 1000 or more flows, while the CPU cost of each 1 Gbps increases at 500 or more flows despite the lower total throughput. With dual-endpoint control, the throughput tapers to a much lower degree: at 5000 flows, the average throughput is $2.6\times$ higher than without dual-endpoint control. The CPU cost comparison is even more stark: at 5000 flows, 1 Gbps requires $10.7\times$ as many CPU cores without dual-endpoint control, despite the host servicing lower higher throughput without scheduling. The remainder of this section explores other metrics in more detail to explain these benefits.

⁵In production workloads, connections are persistent and reused for subsequent bursts, so steady-state behavior dominates over warmup behavior.

⁶Note that the flow counts chosen for experimentation and the flow count at which performance degrades precipitately (i.e., throughput decreases while CPU cost increases) are slightly different here than in Figure 4.1 because these datasets were collected separately on different CloudLab clusters (using the same machine type, as described in Section 4.6.1).

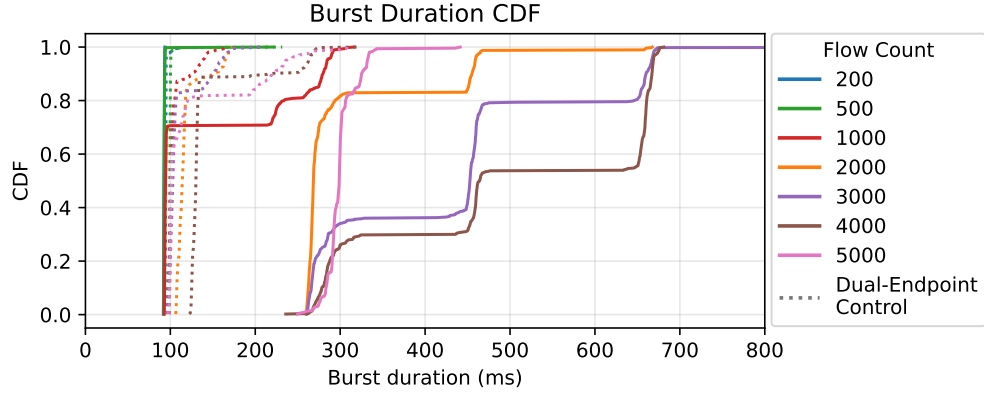


Figure 4.10: Burst duration CDF: Flow scheduling keeps the total in-flight data within the capacity of switch buffers, reducing retransmissions and improving tail latency.

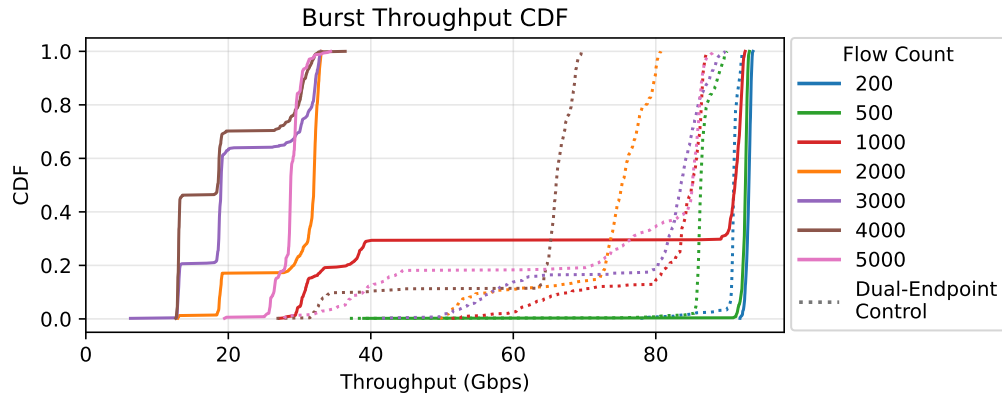


Figure 4.11: Average burst throughput CDF: Shorter burst durations with flow scheduling naturally implies higher throughput during bursts, making better use of available network resources.

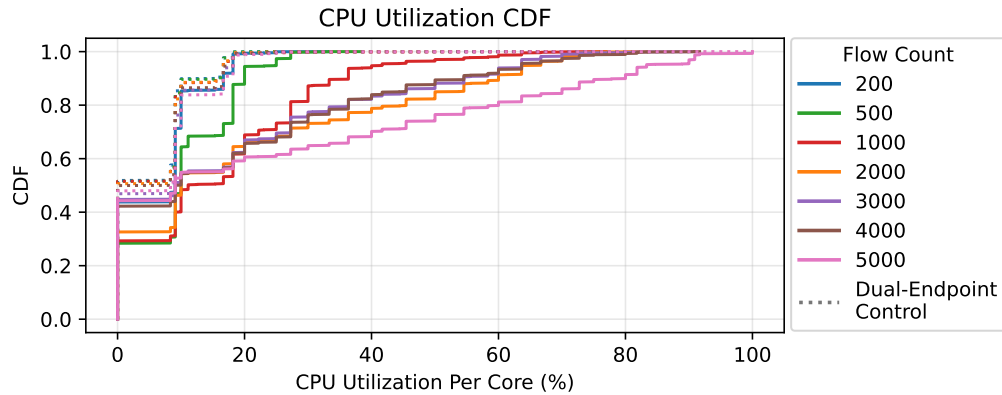


Figure 4.12: CPU utilization CDF: Reducing receiver multiplexing improves CPU utilization through more effective amortization of per-packet overheads and fewer caches accesses of control structures.

4.6.3 Burst duration and average throughput

First, we look at the impact of the incast scale and dual-endpoint control on the burst duration. Figure 4.10 shows the CDF of the burst duration with different flow counts. In this and the following measurements, we observe two performance regimes, 200 and 500 flows versus 1000+ flows. For the 200 and 500-flow experiments, the incast bursts perform well with single-endpoint control. At 1000 flows and above, in this figure we observe TCP timeouts due to switch overflows with insufficient packets in flight to trigger triple-dupACK fast retransmit. We see a step behavior in which a subset of flows experience successive timeouts that each increase the burst duration by 200 ms (the minimum TCP retransmission timeout). This corresponds to the behavior seen in Figure 4.6a, where many flows with a single packet in flight have enough aggregate data to cause the receiver’s ToR switch queue to overflow. Since application workloads are often barrier-synchronized and recursive with a hard latency objective on each step, this directly results in a slowdown in application response time for users.

The dotted lines in Figure 4.10 show the behavior of each flow count with receiver flow scheduling enabled. When introducing dual-endpoint control, for the 200 and 500-flow cases, we observe an up to 12.0% penalty in P99 burst duration. This is due to the additional overhead of pausing and unpausing flows during scheduling, the timing of which is inherently challenging to overlap. With 1000–5000 flows, the P99 burst duration decreases by up to 74.4%. These improvements are due to reducing (and in many cases eliminating) the TCP timeouts describe above. For 1000 flows and above, while the burst duration with scheduling is significantly improved, it is still above the target of 86 ms. This is due to scheduling overheads that emerge at high flow counts and can likely be resolved by increasing the number of active flows to increase the total in-flight data and increasing the grant end buffer to improve grant overlap. The inverse of a reduction in burst duration is an increase in average network throughput during bursts, as shown in Figure 4.11.

4.6.4 CPU utilization

By reducing the number of active flows, receiver flow scheduling allows each flow to have many packets in flight. This allows for improved use of cross-packet aggregations that amortize per-packet processing over a larger data size, resulting in lower CPU utilization to achieve line rate. We explore the impact on GRO in the next section. Figure 4.12 shows a CDF of per-core utilization for the cores running receiver threads. Each sample in the CDF is a CPU utilization measurement at intervals of 100 ms for 60 seconds. The CDF shows that, across all percentiles, scheduling reduces the CPU utilization at all flow counts. Even in the 200 and 500-flow cases where scheduling does not reduce burst duration, it still saves CPU. Across flow sizes, CPU utilization decreases by 59.6% on average

4.6.5 Socket Buffer Length

To understand the reasons behind the reduction in CPU utilization shown above, we turn to the efficacy of receiver aggregation techniques. As described in Section 4.2.2, receivers use a variety of approaches to amortize per-packet processing overheads over larger chunks of data. Per-packet processing includes updating internal state tracking reliability, assembling out-of-order data, and delivering data to the application. We look at one such technique in detail, GRO (Section 4.2.2.1), and show that flow scheduling presents more contiguous data from each flow to a receiver, enabling GRO (either in the NIC hardware or the software stack) to more effectively combine packets on the wire into larger segments.

Figure 4.4 shows the size of the socket buffers (SKBs) of the post-GRO segments in the Linux TCP stack. Due to the high packet rate, this measurement (in bytes) is collected in buckets sized as powers of two. Therefore, the figure shows a CDF of bucket occupancy (packet counts in that bucket’s range), not raw SKB sizes. Without dual-endpoint control, the solid lines show that as the flow count increases, the SKB size distribution shifts towards smaller sizes. At 2000 flows and above, the bulk of SKBs are

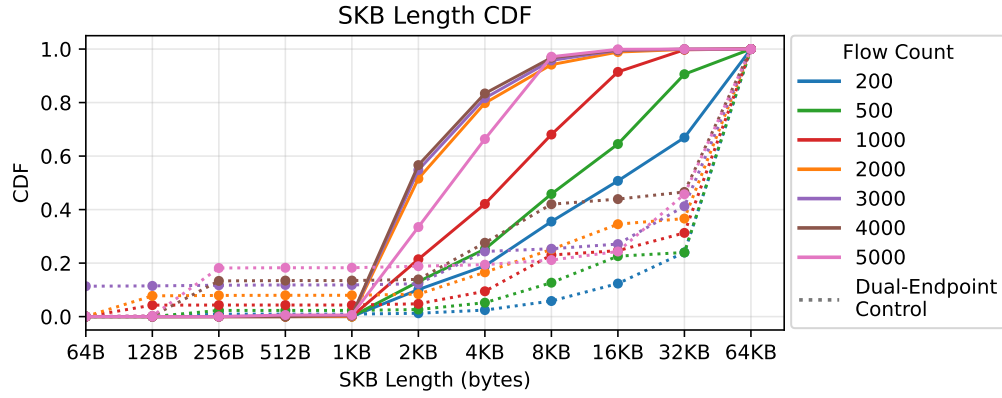


Figure 4.13: Socket buffer (SKB) length CDF: With flow scheduling, fewer active flows means better per-flow aggregation and therefore larger socket buffers.

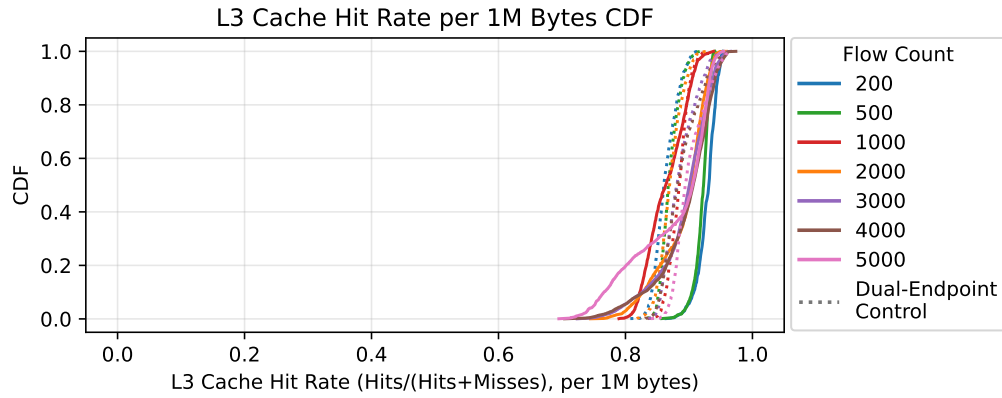


Figure 4.14: L3 cache hit ratio per 1 MB CDF: Despite reducing cache accesses, scheduling actually decreases L3 cache hit rate.

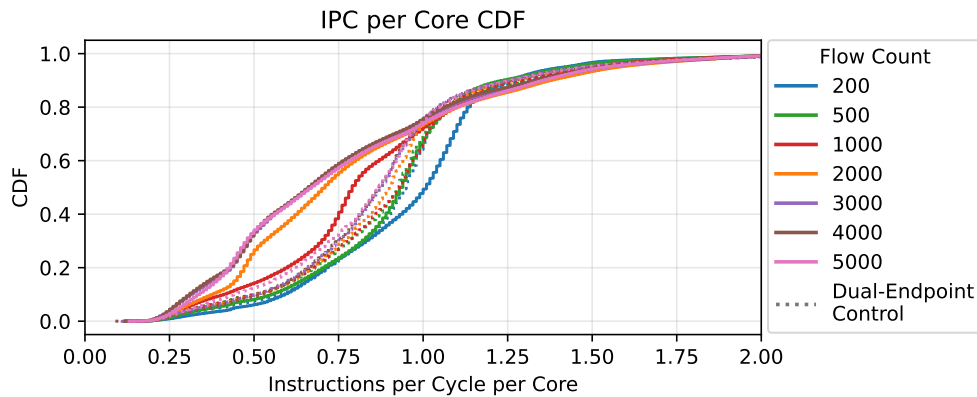


Figure 4.15: Instructions per cycle CDF: By reducing time spend waiting for cache accesses, flow scheduling allows CPUs to execute more instructions per cycle.

1 KB–8 KB, which is 1–5 packets given an MTU of 1500 B. With thousands of flows active, the receiver does not receive more than 1–5 packets for each flow within the GRO period. This is a result of two factors. First, each sender converges to a small number of packets in flight, so each flow does not offer more than a few packets at a time. Second, given a fixed receiver bandwidth and an increasing number of flows, regardless of the sender rate decision, each flow naturally receives a smaller fraction of the bandwidth and delivers fewer packets to the receiver per unit of time.

With receiver flow scheduling, Figure 4.13 shows that the median SKB length increases by an average of $6.9\times$ across flow sizes. This is more pronounced at high incast degree: with 2000 or more flows, the median SKB length increases by $8.0\times$. In these cases, most SKBs are 16 KB–64 KB (11–45 packets), demonstrating a higher amortization ratio for processing per-packet compared to packets 1–5 with sender control alone.

Interestingly, we also observed an increase in small-size SKBs (<1 KB) with scheduling. These are an artifact of how improved aggregation interacts with the specific data size of each burst. After aggregating most packets into large segments, there is occasionally a single small packet left over. Without scheduling, the per-flow packet rate is lower, meaning it is more likely that there is an open GRO wait in progress that can incorporate this small extra packet. In practice, these do not have a negative impact on performance during scheduling compared to the benefit of creating the large SKBs.

4.6.6 Cache behavior

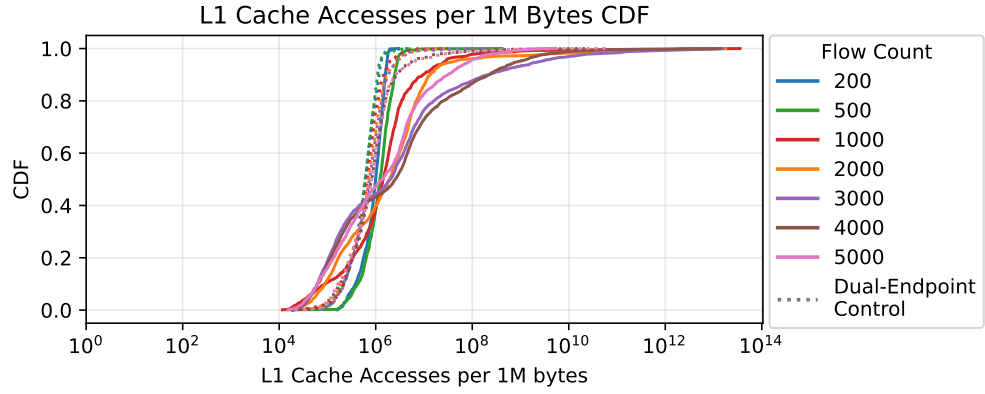
Reducing the degree of multiplexing also improves cache locality. For each packet, the receiver must reference the data structures that contain metadata for that flow. In TCP, the receiver tracks sequence numbers, timestamps, and data pointers, among other things. By reducing the number of active flows, receiver flow scheduling reduces the number of separate control structures that must be loaded and stored in the cache per unit of time. Figure 4.16 shows CDFs of the L1, L2, and L3 cache access counts. Note that the x-axes are log scale. We see that in all cases, the median and P99 cache access count per megabyte decreases when using flow scheduling, even for the cases of 200 and 500 flows. Across the three caches, the average access count across flow sizes decreases by 63.0%. This directly follows from the increase in SKB size above, as well as improved temporal locality within flows.

Receiver flow scheduling reduces the degree of control multiplexing that the receiver must perform, reducing the size of the working set. This has two benefits. First, with fewer active flows, it is more likely that a packet’s control structures are already in cache. When a flow is given a grant and its new packets start to arrive, it will inherently incur a memory access to fetch its control structures, as is the case with packet arrivals in sender-only control. The benefit from reducing the degree of multiplexing comes from avoiding cache misses for subsequent accesses to the same flow’s state: instead of always needing to fetch the state from memory, now we only fetch at the start of each grant. The second benefit is that by reducing the working set size, we free up cache space for other data, likely improving the hit rate of other programs.

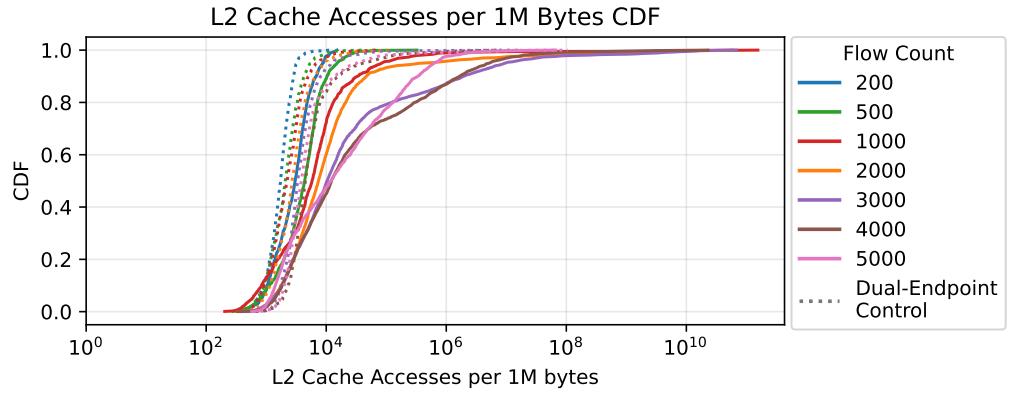
Despite the significant reduction in cache accesses, if we look at the cache hit rate within the accesses which do occur, we actually observe that with scheduling the hit rate decreases. Figure 4.14 shows the L3 cache hit ratio per 1 MB of received data and we observe that median hit rate decreases by 3.0% across flow sizes. This is likely because scheduling removes metadata accesses, so a higher ratio of accesses come from loading the packet data buffers, which incur compulsory misses.

4.6.7 Instructions per cycle

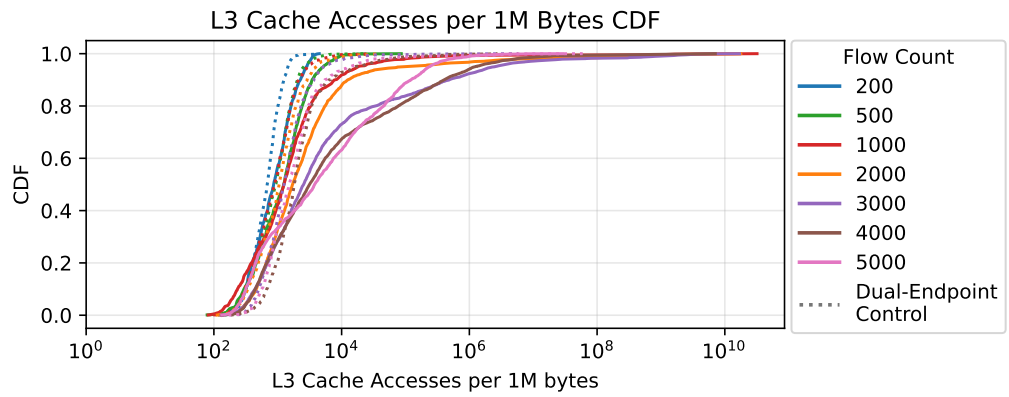
Lastly, we look at the impact of multiplexing on CPU efficiency in terms of instructions per cycle (IPC). IPC is a measure of the parallelism that the CPU is able to achieve, and is influenced by factors such as cache hit rate. Figure 4.15 shows a CDF of IPC, measured at 100 ms intervals over 60 seconds. We see



(a) L1 cache.



(b) L2 cache.



(c) L3 cache.

Figure 4.16: CDFs of L1, L2, and L3 cache access counts per 1 MB of received data. Flow scheduling reduces multiplexing, improving cache locality and reducing access counts. Note the log-scale x-axis.

a similar behavior to the L3 cache hit rate. With single-endpoint control, there is a crossover between 500 and 1000 flows: incasts with 1000 or more flows experience lower IPC whereas 200 and 500-flow incasts experience high IPC. This is likely a direct result of the cache behavior above: by incurring more cache accesses, the 1000+-flow incasts spend more time stalled waiting on data instead of executing instructions. With receiver flow scheduling, the average IPC increases by 13.1% for 1000+ flows, but decreases by 1.8% for 200 and 500 flows (for an average increase across all flow counts of 8.8%). These changes in IPC alone are insufficient to explain the reductions in CPU utilization in Figure 4.12. The bulk of the CPU benefits come from improved GRO efficacy as described in Section 4.6.4.

4.7 Conclusion

The continual increase in network bandwidth in combination with the growth of high-fan-in datacenter applications means that high degrees of multiplexing are a pressing challenge for both datacenter networks and end hosts. This chapter explored how dual-endpoint control incorporates the objectives of the receiver, in this case network and host resource efficiency under multiplexing, into a previously sender-oriented rate decision. Our receiver flow scheduling solution demonstrates the flexibility of a dual-endpoint approach to provide a different network service model, converting a processor-sharing resource allocation scheme into a grant-based scheme that trades off short-term unfairness for improved network, CPU, and memory efficiency for incasts of thousands of flows. Our implementation is based on standard Linux TCP augmented with an eBPF datapath and a transparent userspace scheduler. We also provide a new incast burst traffic generator to facilitate future studies of this ubiquitous traffic pattern.

Looking towards future work, the grant-based round-robin scheduling algorithm presented here is just a starting point. The design makes several assumptions, including that bursts repeat, that incast scale is stable, and that the transport layer can determine the size of burst requests. Section 4.5 describes approaches to relax these assumptions, but together these point to a spectrum of questions about how to provide precise flow scheduling under imperfect workload information.

Another direction is to evaluate the performance of receiver flow scheduling under background congestion. In our experiments, the incast is the only congestion on the network. Even if an incast is the overall bottleneck for a workload, a flow may experience background congestion elsewhere. Since dual-endpoint control incorporates a receiver rate decision with existing sender-side congestion control, the combined algorithm can adapt to congestion other than the incast that the receiver is scheduling for. However, if the flow's sender rate decision is lower than its receiver rate decision, then it will not meet the rate allocation that the receiver is providing. If many flows experience background congestion and cannot meet their receiver allocations, then the incast link (which was previously the bottleneck) may be underutilized. The receiver scheduling design could be extended to detect flows that cannot meet their rate allocations and shift grants to other flows to maintain high utilization.

In addition to resource-focused policies, application-aware policies are also possible. Through cross-layer optimizations that incorporate applications themselves, a priority based scheduling design that is tailored to particular workloads provides additional opportunities to improve end-to-end performance. For example, with visibility into applications, flow scheduling can enforce classes of service (or fairness) across flows without the need for in-network techniques like priority queues or fair queuing. Other endpoint considerations, such as rate limiting or fairness between in-datacenter and cross-datacenter flows, as described briefly in Section 3.4.3, are also possible. In the next chapter, we look at how to expand the utility of dual-endpoint control to a more heterogeneous environment, the Internet at large, and the policies of Internet users.

Chapter 5

Dual-Endpoint Control for Internet Fairness

5.1 Introduction

Our case studies with datacenter incast bursts showed the benefits of a dual-endpoint approach to rate control for a structured environment where the receiver faces a specific pathological traffic pattern using well-understood applications. In this chapter, we leave the datacenter behind and consider the implications of dual-endpoint control for the Internet at large. The Internet is a more heterogeneous environment than a datacenter, where we observe interactions between a wider array of applications, devices, and congestion control algorithms. Our goal in this chapter is to show the generality of the dual-endpoint design by exploring cases with different endpoint policies, instead of a particular traffic pattern.

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 [19, 40], there are two places that this *tussle* [29] 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 [26], although not all CCAs achieve their goals [118]. 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 [33], or traffic prioritization (e.g., VoIP traffic over web browsing [9]). Perhaps more starkly than in the datacenter environment, the final rate achieved by a flow is therefore the *union* of the sender and network operator’s rate control policies.

Similar to the earlier chapters about datacenter incast bursts, we argue that the observations, requirements, and rate policies of the receiver are left out of the discussion on today’s Internet as well. In this chapter, we treat the receiver’s policy more ambiguously to show the generality of the dual-endpoint approach. We identify three challenges that arise on the Internet 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 [33, 88].
- **Example 2 — Differences in rate control algorithm:** Most traffic on a network uses a particular loss-based CCA, like TCP Reno [61]. A client accesses a service using a different loss-based CCA, such as TCP Cubic [51], 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 [17]. However, internal flows must coexist with incoming Internet traffic that uses a buffer-filling algorithm that prioritizes high throughput 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 chapter, we show how dual-endpoint congestion control gives both endpoints a stake in Internet 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 dual-endpoint 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, dual-endpoint control does not interfere with the packet stream to delay or drop packets.

To be clear, dual-endpoint control is also not a new CCA and, in fact, the implementation of such a mechanism at the receiver poses different challenges than traditional CCA designs. 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.

To illustrate receiver policies, we design Ratemon, which employs similar receiver window tuning techniques as in Chapter 4. Ratemon is 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, dual-endpoint 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 dual-endpoint 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 dual-endpoint 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 dual-endpoint 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 [81], 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, dual-endpoint control for the Internet can implement other policies related to access control, fairness, or resource management.

Section 5.2 examines the above three challenges in detail as well as existing steps towards dual-endpoint rate control. Section 5.3 describes the design of Ratemon, including its mechanism for cooperative control. Sections 5.4 and 5.5 present ServicePolicy and FlowPolicy, respectively. Section 5.6 evaluates these techniques to demonstrate the efficacy of dual-endpoint control in giving the receiver a stake in rate allocation on the Internet. Section 5.7 discusses related work, and Section 5.9 concludes.

Ratemon is available open source at github.com/cmu-snap/ratemon commit **0d23307**.¹ This chapter describes work that was published in a tech report [21].

5.2 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, receivers on the Internet 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.

5.2.1 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 [45], round trip time (RTT) fairness (where the flow rates are scaled inversely proportionally to the RTT), quality-of-experience fairness [89], absolute fairness, slowdown [124], 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 dissertation 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 chapter, 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 dual-endpoint rate control to adhere to the receiver’s policy. For example, a network dominated by incumbent TCP Reno flows that achieve

¹Note that this is the same repository used for the receiver flow scheduling code in Chapter 4, but an earlier commit.

per-flow fairness interacts with a newcomer TCP Cubic flow using a different ramp-up algorithm.

5.2.1.1 Datacenter techniques

Much of the earlier chapters of this dissertation deal with the datacenter environment. While we take inspiration from existing datacenter techniques, for this chapter we target the Internet, 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 workloads that lend themselves to application-specific optimization, as we explored for incast bursts.

5.2.2 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 dual-endpoint rate control offers benefits over the status quo for that particular class of problem.

5.2.2.1 Challenge 1: Different allocation granularity

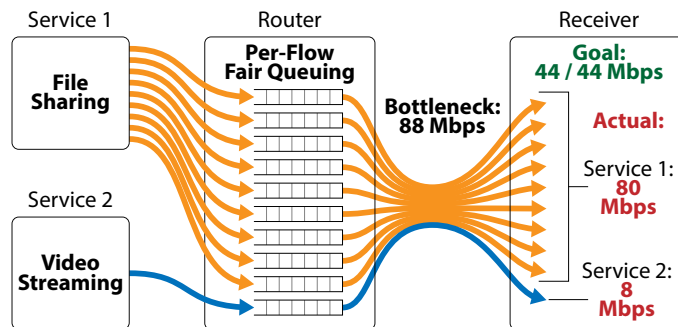
Entities consider rate allocation at different granularities. To illustrate this challenge, we explore how an application that uses multiple concurrent flows can claim more bandwidth than an environment based on per-flow rate control would normally allocate to another application that uses only a single 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 chapter focuses on bulk download applications that are bottlenecked in the network, although we evaluate interactions with application-bottlenecked flows in Section 5.6.2.4. 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 [91], HTTP/2 [111], and QUIC [59] 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 [10, 11, 12, 93, 112], described in Section 5.7) have considered rate allocation at the granularity of parallel flows between two endpoints.

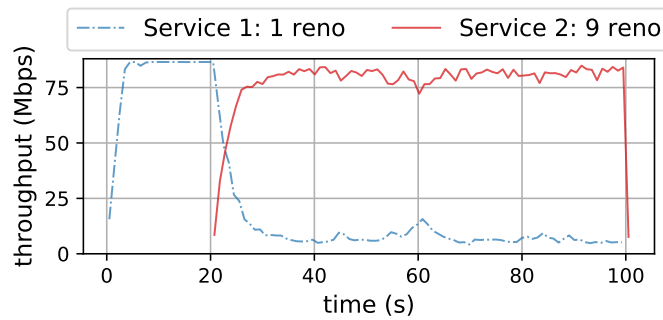
Consider Example 1 (Figure 5.1) from the introduction. Suppose that a receiver’s policy is to share bandwidth equally between services. Figure 5.1a shows a client accessing two services, an incumbent video streaming service, such as YouTube [123], that uses a single connection, and a newcomer file sharing service, such as MEGA [82], 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 ($\approx 3 \times \text{BDP}$ ²). 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 5.1b shows a throughput trace for this example, where the video streaming service starts first, followed 20 seconds later by the file sharing service.

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) [33, 88] to provide strong enforcement of flow-level fairness (as illustrated in Figure 5.1a), but per-flow FQ has no benefit in this

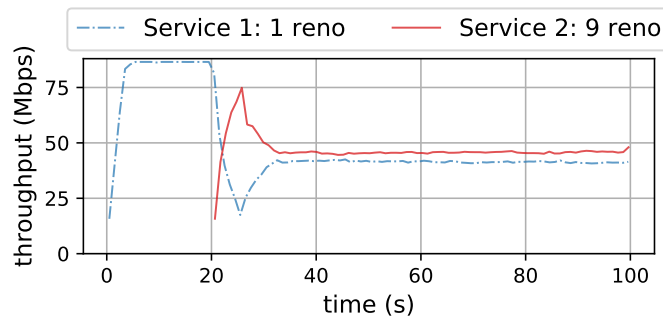
²The bandwidth-delay product (BDP), which is equal to the bottleneck bandwidth times the RTT, is the volume of data that a network path can transport before inducing queuing.



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

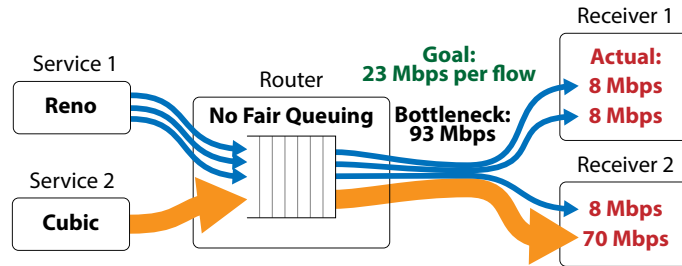


(b) The file sharing service claims over 75% of the bandwidth, but the client desires that both services get the same allocation.

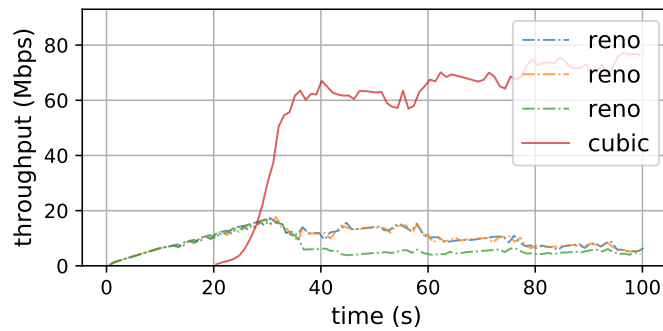


(c) ServicePolicy limits all the flows in each service collectively to the behavior of one Reno flow.

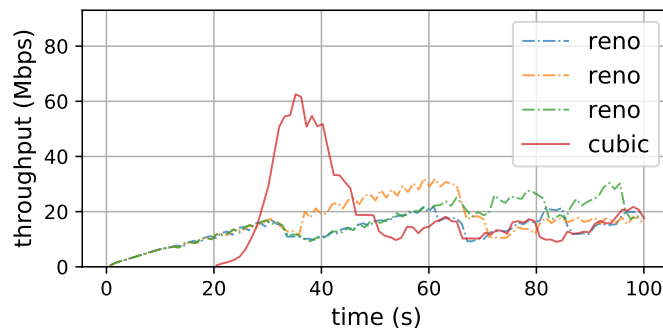
Figure 5.1: Example 1: In an environment of per-flow rate allocation, a service using many flows claims a higher rate.



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

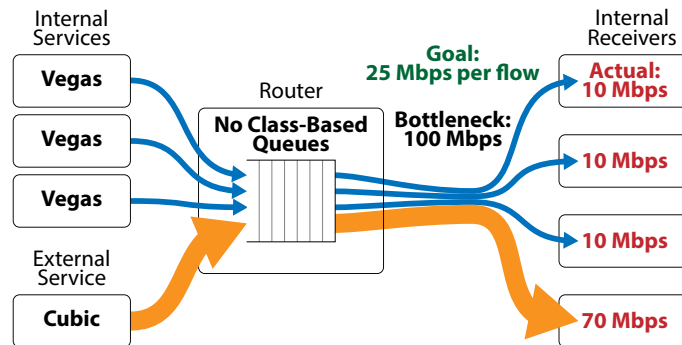


(b) The one Cubic flow claims over 65% of the bandwidth, but the client desires that each flow get a 25% bandwidth allocation.

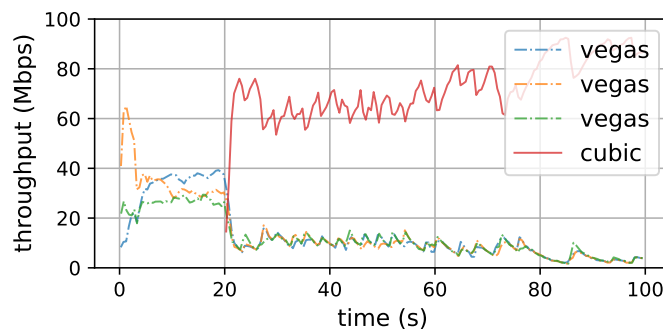


(c) FlowPolicy detects that the Cubic flow is unfair and dynamically guides it to an appropriate rate.

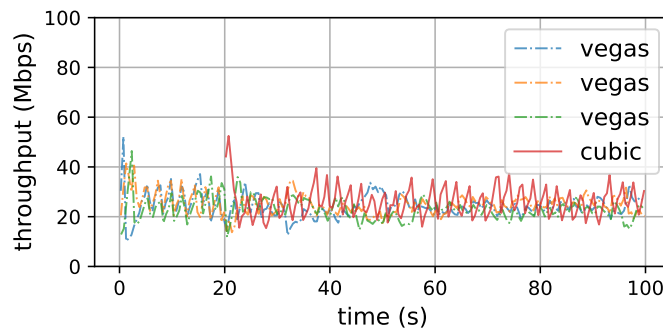
Figure 5.2: Example 2: Two CCAs with the same metric (loss) but different algorithms do not converge to equal rates.



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



(b) The Cubic flow claims $\approx 70\%$ 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 5.3: Example 3: Two CCAs with different congestion metrics (delay vs. loss) do not converge to equal rates.

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 [11, 12, 93, 112]) 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. Dual-endpoint rate control offers an alternative: give the receiver influence over rate allocation at the transport layer, externally from the application. Figure 5.1c shows how ServicePolicy, described in Section 5.4, holds each service to its independent fair rate.

5.2.2.2 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, which governs 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 [26, 62].

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 5.2) 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 [51]. 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 [26].

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 5.2a 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 ($\approx 1 \times \text{BDP}$). Figure 5.2b 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 5.2c shows how FlowPolicy, described in Section 5.5, 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.

5.2.2.3 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 [17], 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 [17]. However, by keeping queues short, these techniques risk underutilizing 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 5.3) 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 5.3a 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 ($\approx 0.5 \times \text{BDP}$). 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 5.3b. The company could configure its routers to divide the internal and external flows into separate traffic classes served by separate queues, but Section 5.5 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 5.3c shows FlowPolicy, described in Section 5.5, 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 5.6 also considers interactions between Cubic and rate-based BBR, which is used by many Internet services.

5.3 Design Overview

To address the three challenges in Section 5.2.2, this section describes how dual-endpoint rate control is implemented for Internet flows in the Ratemon platform. For dual-endpoint control on the Internet, we again rely on receiver window tuning, which has been described extensively in this dissertation. First, we explain the design constraints faced by dual-endpoint control and how these differ from CCA design. Second, we describe limitations of receiver window tuning. Finally, we outline the Ratemon architecture, including its pluggable rate control policy.

5.3.1 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 dual-endpoint 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.

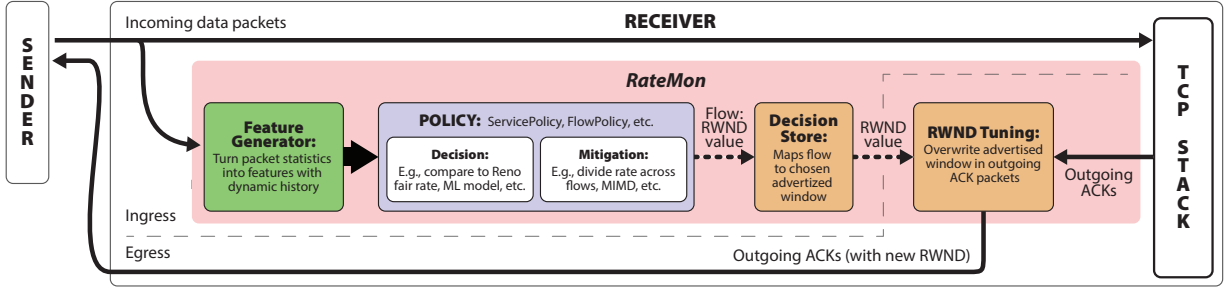


Figure 5.4: The Ratemon platform. Solid lines represent data and dotted lines represent control.

5.3.2 Limitations of receiver window tuning

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, 110], 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 dissertation. The sender must obey TCP flow control for Ratemon to be effective. The receiver does not control the packet stream, so dual-endpoint 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.

5.3.3 The Ratemon platform

Building on our understanding of RWND tuning from earlier chapters, this section outlines how to use this technique to design the Ratemon system as a whole: its architecture, continuous rate monitoring, and implementation details.

5.3.3.1 Architecture

Ratemon consists of two components: a general, flexible *mechanism* for receiver-assisted rate control based on RWND tuning, and a pluggable 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 [36] programs that run on a receiver and are separate from the TCP stack. Refer to Section 5.8.2 for eBPF details and rationale. Figure 5.4 gives an architectural overview. Modules are described below.

- **Feature Generator:** Ratemon generates a copy of incoming data packets using libpcap [48]. 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 5.5.2.2). Examples include the loss event rate [54], 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 5.4) and FlowPolicy (Section 5.5). 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 Section 5.8.2 for implementation details.

5.3.3.2 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 5.4 & 5.5 describe the unique mitigation approaches taken by ServicePolicy and FlowPolicy, respectively.

5.4 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 dual-endpoint rate control through a well-understood context (services using concurrent flows) and technique (Mathis equation for Reno throughput [81]). This section defines ServicePolicy’s objectives, describes its implementation in Ratemon, and revisits Example 1 (Figure 5.1) to demonstrate its performance.

5.4.1 ServicePolicy definition

Section 5.2.2.1 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 throughput it would achieve if it were using a single flow implementing Reno’s congestion avoidance algorithm [81] 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.

5.4.2 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 [81], which specifies the

rate that a well-behaved Reno flow should achieve given a certain RTT, loss rate (p), and maximum segment size (MSS): $\text{throughput} = (MSS \times C) / (RTT \times \sqrt{p})$, 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) [54] 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 [12, 93, 112]. 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.

5.4.3 ServicePolicy demonstration

To show ServicePolicy in action, we revisit Example 1 from Section 5.2.2.1. Later, Section 5.6 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 5.1b shows that without ServicePolicy, the nine file sharing flows claims the majority of the bandwidth. Figure 5.1c illustrates the utility of dual-endpoint 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.

5.5 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

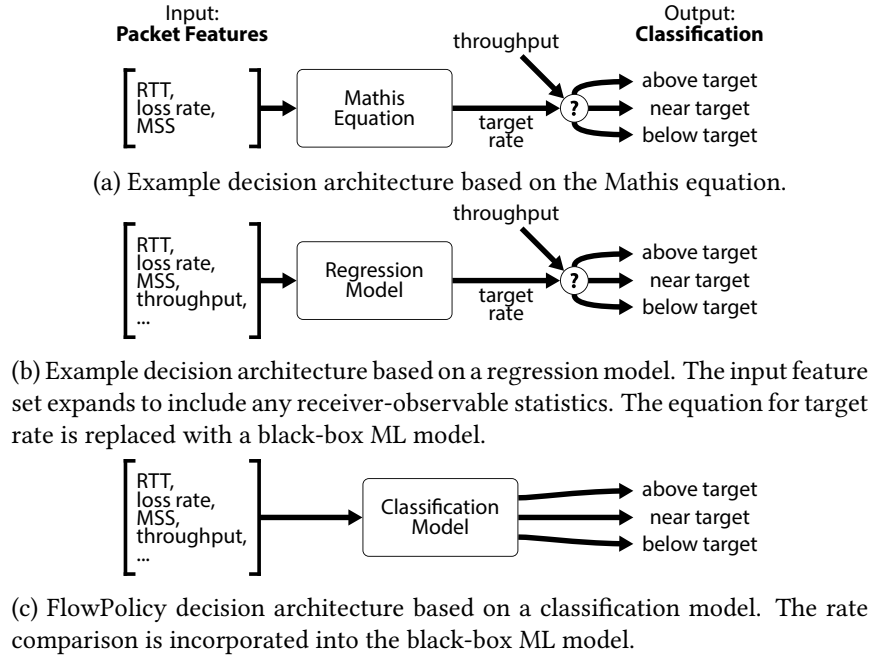


Figure 5.5: Comparing decision architectures.

the Mathis equation, which models TCP Reno behavior. To address this issue, FlowPolicy supports more expressive rate allocation policies by relying on 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.

5.5.1 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 5.5a. 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 5.5, 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 5.5b). Since the desired output of FlowPolicy’s decision module is a classification of above,

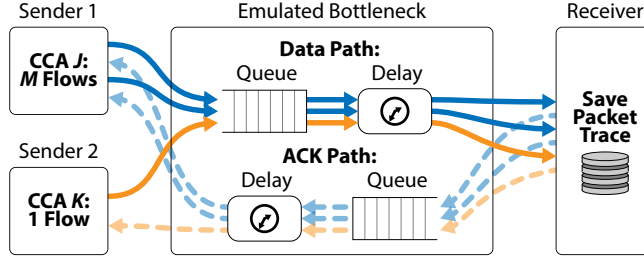


Figure 5.6: The emulator’s topology. Solid lines are data packets and dotted lines are ACK packets. Blue lines are incumbents and orange lines are newcomers. Each delay module emulates half of the RTT.

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 5.5c. In a process described in Section 5.5.3, 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:

$$\text{class} = \begin{cases} \text{above target} & \text{if } T \geq 1.1 \cdot B/N \\ \text{near target} & \text{if } 0.9 \cdot B/N \leq T < 1.1 \cdot B/N \\ \text{below target} & \text{if } T < 0.9 \cdot B/N \end{cases} \quad (5.1)$$

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.

5.5.2 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 5.3.3, the logic described here comprises FlowPolicy’s *decision* module.

5.5.2.1 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 5.6, creates a dumbbell topology from three hosts. One host uses the Berkeley Extensible Software Switch (BESS) [52] 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

Table 5.1: 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

emulate the senders and receiver and communicate via the bottleneck host. Table 5.1 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 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.

5.5.2.2 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 Section 5.8.1, Table 5.4.

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 [57]. 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 Section 5.8.1, Table 5.4 for a list of features that use dynamic history. Section 5.8.1, Table 5.5 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 5.3.1, 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* [83], 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 [69], then the clusters are arranged in a hierarchy and a cut of this hierarchy is chosen that yields the desired number of clusters, F [102]. 3) A total order of features is created based on permutation importance [18]. 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. Section 5.8.1.1 explores how the selected features change based on the CCAs in the training data.

5.5.2.3 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 dual-endpoint control.

FlowPolicy uses a subtype of DT called a *histogram-based gradient boosting decision tree* [67]. 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 [44]. 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 $H = 255$) [67].

Recent work has shown that learning-based CCAs struggle to generalize to scenarios beyond their training data [105]. While FlowPolicy is not a CCA, it likely faces similar limitations. Section 5.6.2 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 5.1 covers a sufficiently broad cross-section of home and mobile network conditions to demonstrate FlowPolicy’s capabilities. Refer to Section 5.8.3 for additional training details and parameters.

Sage [122] 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.

5.5.3 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. Referring to Section 5.3.3, 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:

$$T' = \begin{cases} 0.57 \cdot T & \text{if class == above target} \\ T & \text{if class == near target} \\ 1.3 \cdot T & \text{if class == below target} \end{cases} \quad (5.2)$$

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.

5.5.4 FlowPolicy demonstration

To show FlowPolicy in action, we revisit Challenges 2 and 3 from Section 5.6.1. Later, Section 5.6 evaluates FlowPolicy on a broader range of scenarios.

5.5.4.1 Revisiting Challenge 2 – Differences in algorithm

We revisit Example 2 in Figure 5.2. Three incumbent Reno flows start first, followed by a newcomer Cubic flow, over a 93 Mbps bottleneck. Figure 5.2c 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.

5.5.4.2 Revisiting Challenge 3 – Differences in optimization metric

We revisit Example 3 in Figure 5.3. Three incumbent delay-based Vegas flows start first, followed by a loss-based newcomer Cubic flow, over a 100 Mbps bottleneck. Figure 5.3c 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.

5.6 Evaluation

This section evaluates dual-endpoint rate control more broadly than the three specific scenarios described so far. First, we show how ServicePolicy and FlowPolicy generalize beyond earlier specific 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 5.1, 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) [63], 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 dual-endpoint control, which reduces sender rates, does not cause a disproportionate decrease in overall utilization.

5.6.1 Broad evaluation of the three challenges

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

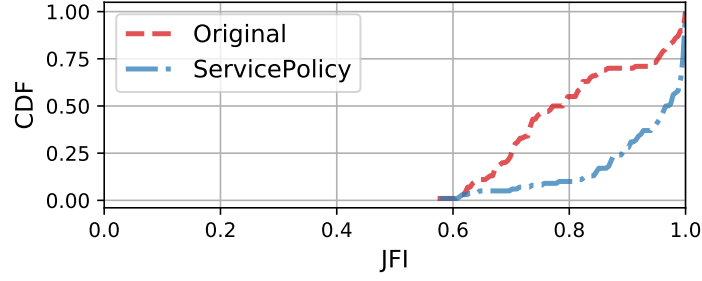
5.6.1.1 Challenge 1

Figure 5.7 evaluates ServicePolicy on 100 random configurations of bandwidth, RTT, queue size, and flow count. Figure 5.7a 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 10th percentile decreases by 11.4%. Figure 5.7b 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 [110].

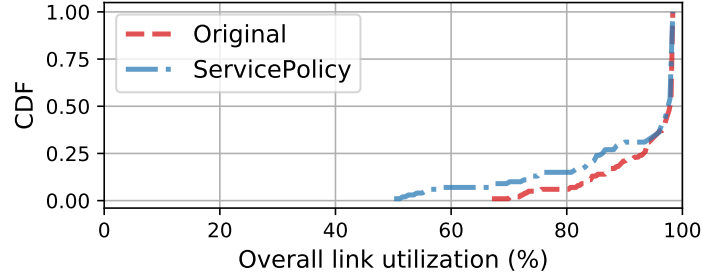
5.6.1.2 Challenge 2

Figure 5.8 evaluates FlowPolicy, trained with Reno as the incumbent and Cubic as the newcomer, on 100 random configurations. Figure 5.8a shows that FlowPolicy improves the median per-flow JFI by 32.7%. Figure 5.8b 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

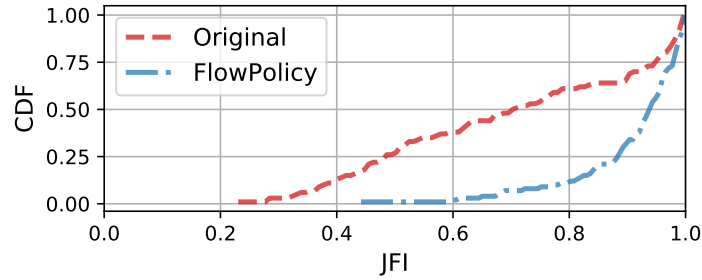


(a) CDF of service-level JFI.

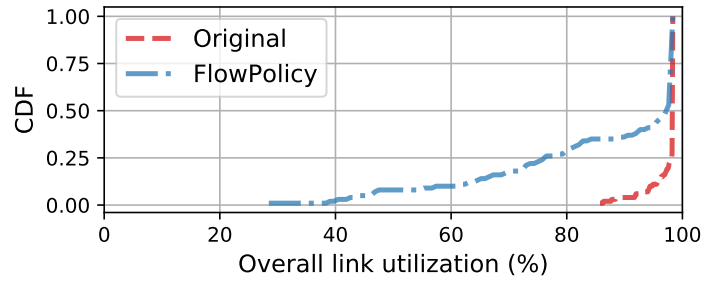


(b) CDF of average utilization.

Figure 5.7: Challenge 1: Across 100 scenarios, ServicePolicy improves adherence to the receiver's policy for rate allocation at the service granularity.

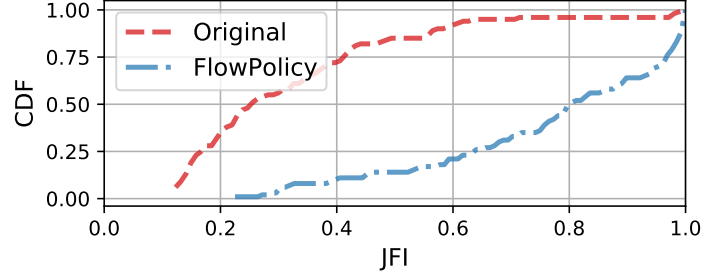


(a) CDF of per-flow JFI.

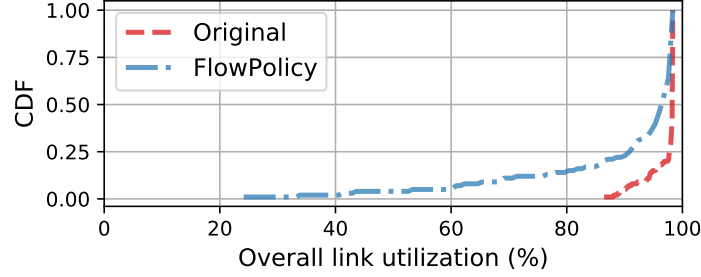


(b) CDF of average utilization.

Figure 5.8: 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 5.9: Challenge 3: Across 100 scenarios, FlowPolicy (Vegas vs. Cubic) improves adherence to the receiver’s policy for optimization metric.

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.

5.6.1.3 Challenge 3

Figure 5.9 evaluates FlowPolicy, trained with Vegas as the incumbent and Cubic as the newcomer, on 100 random configurations. Figure 5.9a shows that FlowPolicy improves the median per-flow JFI by 220.8%. Figure 5.9b 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.

5.6.2 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:

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.

5.6.2.1 Complex metric — BBR

BBR [23] 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 [118]. 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 [118]. 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 [118].

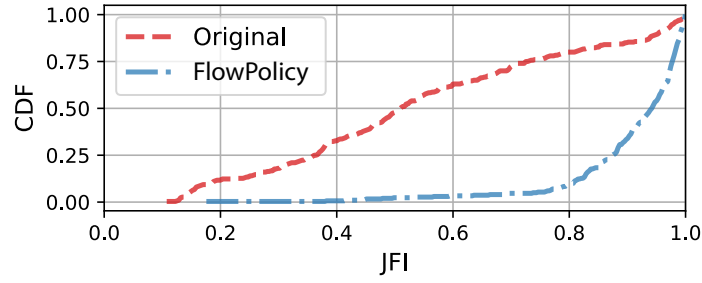
Figure 5.10 shows results for various numbers of Cubic incumbents interacting with a BBR newcomer in 300 random network conditions. Figure 5.10a shows a CDF of the flow-level JFI. BBR is known to take more bandwidth than loss-based CCAs like Cubic, as shown by the uniform distribution of JFI in the “Original” line. FlowPolicy significantly improves the median JFI from 0.51 to 0.94, an increase of 85.1%.

Figure 5.10b 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 only 2.3% below perfectly fair. Figure 5.10c 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%.

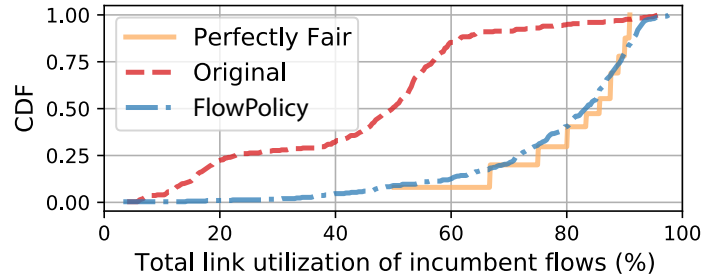
5.6.2.2 All CCAs in Linux

Next, we consider cases where the newcomer CCA is completely foreign to the model. This demonstrates the ability of dual-endpoint 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 5.11 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 10th percentile and median are 6.0% and 0.95%, respectively.

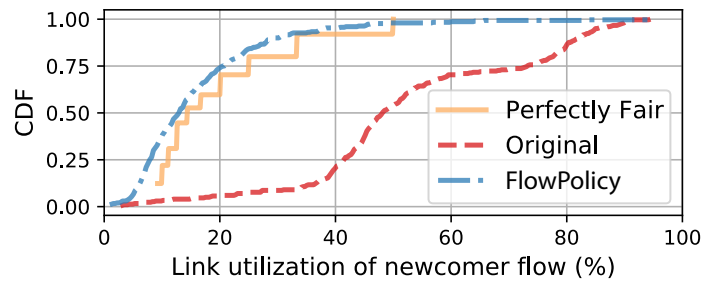
Unseen Highspeed. Next, to show FlowPolicy’s generalizability we look in detail at a newcomer that is not present in the training data. We consider a newcomer flow using TCP Highspeed [39], a modification of AIMD that adapts to high bandwidth-delay product networks by scaling the response function as the congestion window grows. Here, M Cubic flows compete with 1 HighSpeed flow. Figure 5.12a shows that, similar to Vegas, HighSpeed claims less bandwidth than Cubic. FlowPolicy is effective in



(a) CDF of per-flow JFI.



(b) CDF of total incumbent utilization.



(c) CDF of total newcomer utilization.

Figure 5.10: FlowPolicy constrains a rate-based newcomer (BBR) to prevent harm to loss-based incumbents (Cubic).

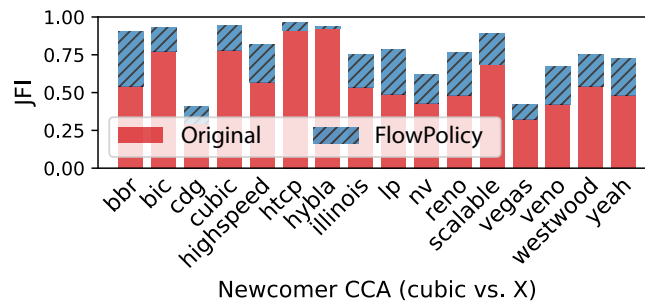
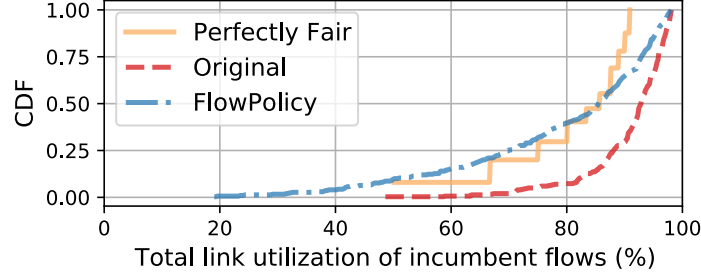
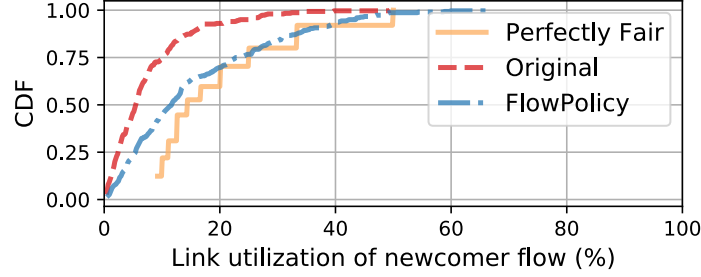


Figure 5.11: When competing with Cubic, all CCAs in Linux see a JFI improvement when using FlowPolicy.



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



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

Figure 5.12: A FlowPolicy model can improve fairness even when a CCA (e.g., HighSpeed) was not part of the training data.

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 5.12b, HighSpeed’s median utilization increases to $2.1 \times$ its value when not constrained, and largely tracks the target fair share.

5.6.2.3 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 5.13 shows the CDF of JFI across 400 configurations. Similar to Figure 5.10, 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.

5.6.2.4 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 5.1 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.

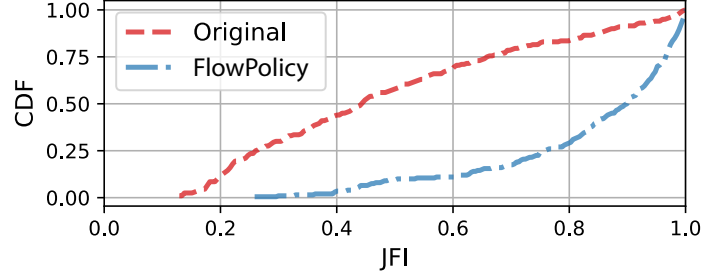


Figure 5.13: FlowPolicy, despite being trained in an emulator, significantly improves the JFI for Cubic vs. BBR even when flows traverse the Internet.

Table 5.2: 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

- 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 5.6.2.1 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%.
- 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 5.6.2.1). Median utilization drops by 5.4%.

5.6.2.5 Partial deployments

Dual-endpoint 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 5.2 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

Table 5.3: 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

deploy FlowPolicy only on hosts that may receive traffic that does not comply with the receiver’s policy, which, in this case, is inbound traffic from the Internet. Hosts that only interact with compliant traffic do not require FlowPolicy.

5.6.3 Microbenchmarks

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

5.6.3.1 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 5.3 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.

5.6.3.2 Runtime performance

Our implementation does not optimize performance (RateMon is written in Python), and instead serves as an example of the types of policies that are possible with dual-endpoint 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 dual-endpoint 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.

5.7 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 [112] leverages existing flows’ state information during new connection initialization. TCP-int [11] shares a congestion window between flows and uses packet delivery on one flow to detect loss on another, as does TCP sessions [93]. The Congestion Manager (CM) [10, 12] proposes a separate module that uses notifications about data transmission and receipt to provide “TCP-friendly” [54] 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 dual-endpoint 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), dual-endpoint 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 [59] protocol is gaining popularity and some multimedia services use UDP and RTP [101]. Extending dual-endpoint 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 chapter 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 [68], or enforcing user-specified weights on traffic classes as in CRAB [110]. 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, 20], and pathological traffic patterns such as incast, similar to ICTCP [121].

5.8 Implementation Details

This section provides additional implementation details regarding FlowPolicy’s dynamic history features and training process.

5.8.1 FlowPolicy features

The tables in this section describe the FlowPolicy model’s input features in detail. Table 5.4 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 5.5 shows the parameter ranges for dynamic history (EWMA decay rates and window sizes).

5.8.1.1 Selected features

Examining the set of features that dynamic history (Section 5.8.1) selects for each model provides insights into the characteristics that are important to the receiver’s rate control policy. Table 5.6 shows the top 10 (out of 20 total) features selected by the three models evaluated in this chapter. 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 5.4 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.

5.8.2 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 [36]. We selected eBPF for this implementation

Table 5.4: 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 5.5 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 [54]	-	-	✓
Mathis model throughput, using loss rate	Throughput that the Mathis model predicts a Reno flow should achieve given the current RTT and loss rate [81]	-	✓	✓
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 5.5: 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, $32 \times \text{min RTT}$
Sliding window size when using LER	4, 8, $16 \times \text{current RTT}$

Table 5.6: 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 5.7: The number of network configurations in a FlowPolicy model’s training data, as chosen from Table 5.1.

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

because it allows low-level access into the Linux network stack without the complexity of writing a custom TCP variant, a kernel module, or modifying the network 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 [15]. 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) [58].

5.8.3 FlowPolicy training

5.8.3.1 Training data

Our core training data includes many configurations from Table 5.1, chosen uniformly at random without replacement. Table 5.7 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.

5.8.3.2 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 dual-endpoint rate control.

5.9 Conclusion

The debate over rate control on the Internet is alive and well, including what policies to enforce and which entities should enforce them. 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 two practical policies that show the potential for dual-endpoint congestion control on the Internet at large. The efficacy of these approaches opens the door for future work for a broader set of scenarios based on the preferences of each individual user. The three examples in this chapter are general concerns, but each Internet user has their own set of implicit (or explicit) objectives when interacting with services. It is natural for a dual-endpoint approach to include these as well. We conclude in the next chapter with a look at final takeaways from our case studies of dual-endpoint congestion control and potential future directions.

Chapter 6

Conclusion

This dissertation is based on the key observation that senders and receivers have different objectives for a rate control decision. We propose dual-endpoint congestion control, which incorporates rate decisions from both endpoints in a connection instead of shipping congestion state and making a central decision at a single endpoint. We implement dual-endpoint control on top of TCP flow control and demonstrate its benefits through three case studies into progressively more expressive endpoint policies.

To conclude this dissertation, this chapter reflects on our findings. The chapter begins by revisiting our key contributions. We then return to our design limitations and assumptions, notably the choice to use TCP flow control, and consider alternative designs that also point the direction towards future work. This chapter then considers two recent research trends that show the importance of the receiver to congestion control and characterize the Internet fairness environment, painting new opportunities for dual-endpoint control. Finally, we end with a survey of additional topics for future work.

6.1 Takeaways

We begin by briefly summarizing the key contributions of this dissertation.

1. The objectives of both endpoints matter. Whereas senders seek to balance resource allocation across their clients, each client in turn must balance across its services. Senders control probing and put bytes on the wire, but receivers may have useful observability into congestion. We follow two motivating examples. In datacenter incast bursts, senders seeking to meet client deadlines are oblivious to the receiver-centric traffic pattern and penalties to packet processing efficiency that the receiver experiences under high degrees of multiplexing. On the Internet, content providers map application demand to flows in a manner that may not respect client preferences using diverse CCAs that may not coexist fairly. In these cases and others, the sender and receiver have differing objectives.

2. Combining state is insufficient. We find that techniques that centralize state instead of allowing each entity its own rate decision inherently disregard one or both endpoints' objectives. Congestion control is traditionally sender-based, with probing of the network and limited congestion feedback from switches, but with no input from the receiver about the traffic pattern or host state. Centralized techniques combine state from multiple endpoints, but ultimately make the rate control decision according to global optimization criteria. Receiver-oriented techniques take demand requests from the senders to perform efficient flow scheduling, but are agnostic to cross-client concerns.

3. Dual-endpoint control combines rate decisions. We propose *dual-endpoint congestion control*, where the sender and receiver combine rate decisions to produce a unified allocation that is amenable to both endpoints' objectives. This approach is not a new CCA, and leverages the sender's existing model of the network to control fine-grained packet transmission. In addition to existing signals like ECN, the receiver also communicates a direct rate decision based on its observation of congestion, the traffic

pattern, and its host capabilities. To avoid resource overload, the sender takes the minimum of both endpoints' rates.

4. Transparent implementation is possible atop existing TCP. We implement dual-endpoint control by leveraging existing TCP flow control, by which the receiver provides a limit on the sender's in-flight bytes. Policy-specific userspace mechanisms can choose a receiver rate and convert it to a window size, then lightweight extensions to the kernel datapath can program that window size into outgoing ACKs. At the sender, the final rate is the minimum of the sender's congestion window, based its modeling of the network path, and the receivers advertised window, customized by dual-endpoint control.

5. Case study on incast bursts shows significant improvements in completion time and resource efficiency. Incasts with thousands of flows surpass the limits of window-based congestion control and negate receiver-side aggregation techniques that are crucial for host CPU performance. Using dual-endpoint control, we build receiver flow scheduling to reduce the number of active flows, trading off temporary unfairness for improved performance. Our techniques enable an order-of-magnitude higher degrees of incast while preserving host CPU efficiency.

6. Dual-endpoint control supports expressive policies akin to in-network controls. In the context of congestion control fairness on the Internet, we show how dual-endpoint control goes beyond resource-based decisions to express rich user policies such as enforcing service-level fair queuing or detecting and mitigating aggressive flows. By observing the incoming packet stream, a receiver can build a model of flow behavior, aggregate flows across services, and reason about fairness properties on a distant bottleneck link. The resulting receiver rate decision, when incorporated with the sender's, gives users a stake in rate allocation on the Internet.

6.2 Limitations and Alternative Designs

The case studies in this dissertation all implement dual-endpoint congestion control transparently, within the transport layer, using existing TCP flow control. Although these case studies show benefits from dual-endpoint control for network utilization, workload completion time, and endpoint policies such as fairness, the choice to implement these techniques on top of an existing mechanism within a longstanding protocol naturally leads to several limitations. These limitations in turn force the question of how our design and its capabilities would differ if dual-endpoint control were implemented as a clean-slate design.

This section explores the limitations of flow control, assumptions we make about application behavior, and interactions with in-network control, and how a clean-slate design might resolve these. We will not describe a clean-slate design in detail, but for each of the above limitations, we will outline considerations regarding how they should be taken into account. We assume that a new design is based on a custom transport protocol that provides functionality similar to TCP, but which can be extended as necessary.

6.2.1 Limitations of flow control

TCP flow control was originally designed to allow the receiver to apply back-pressure to the sender in the event that it had limited receive memory. This informed the design of the advertised window field: the value in the packet header is a window size that directly corresponds to the available memory for in-flight data and is in units of bytes. In the intervening decades since TCP was designed, host memory capacity has outpaced the bandwidth-delay product, and now receive buffer is typically not a limiting resource. However, by implementing dual-endpoint control on top of the advertised window, we are necessarily constrained by its original design.

6.2.1.1 Transmitting a rate decision

In this dissertation, we describe dual-endpoint control as incorporating a rate decision from the receiver with one from the sender (where the sender-side decision may still be informed by other communicated state, such as ECN or INT). This is amenable to our TCP-compatible implementation because a rate is convertible to a window size for use in the TCP advertised window field by multiplying by the RTT. However, other types of decisions are also useful. We see this in the flow scheduling case study, for example, where instead of just a rate, the receiver is also communicating the time duration over which that rate is valid. Together, this rate plus duration becomes a byte grant, which is amenable to the flow control implementation.

However, distilling a grant out of a rate and duration is nontrivial, as we see with the complex bookkeeping needed to grant precisely, update the advertised window as a grant is consumed, and both pregrant and handle “extra” grants due to intricacies with small grant sizes (these mechanisms are all described in Section 4.5.2). For these reasons, in our experience building a correct grant implementation on top TCP was more complicated than expected and took much work to get right. Still, our implementation is fragile and likely to have outstanding edge cases. Looking beyond our case studies, there are other endpoint policies that are challenging to implement in flow control.

As described in Section 4.3.3, a key property that the receiver may want to enforce is prioritization between flows based on considerations of the application layer. Aside from extending the grant design to perform a weighted time-slicing where high-priority flows are chosen for grants more often (or given larger grants), it is not intuitive to implement strict priority atop flow control. Ultimately, the advertised window model is effective as a demonstration and for situations where TCP compatibility is desirable, such as the Internet, but more expressive policies could be achieved, with a simpler implementation, by providing richer primitives to express dual-endpoint policies.

Based on these considerations, a clean-slate design should adhere to the following properties regarding how the receiver provides a rate decision to the sender:

- **Send receiver rate decision, not a window size.** Even if the sender is using a window-based CCA, the sender is in a better position than the receiver to measure the RTT to convert the rate into a window size. For situations where the receiver is limited by memory capacity, flow control is still useful.
- **Send a time duration for which the rate decision is valid.** A timeout is useful to control behavior after idle. As we saw with bursty traffic and DCTCP in Section 2, the rate decision at the end of a period of activity may not apply at the start of the next burst (even if the burst is not an incast). In the flow scheduling case study, fine grant bookkeeping and pregrants were required to prevent a flow from automatically transmitting at the start of the next burst. In cases where the network, workload, or receiver state may change during an idle period, a timeout is a more natural way to force the sender to wait for an up-to-date receiver rate decision before transmitting.
- **Send a flow priority.** The sender should then treat the priority as scoped to the flows destined for that particular receiver. In other words, the sender should use the transmitted priority value to organize its flows to that receiver, but when balancing across receivers, the sender should still use its own considerations. This allows the sender to provide the receiver with a more favorable service model (such as serializing flows or providing more throughput to certain flows) while still respecting the sender’s policies across its clients.
- **(Optional) Still send a flow control window size.** A receiver may still be limited by its memory capacity in pathological cases. If receive memory is not a concern, then this is unnecessary.

6.2.1.2 Combining rate decisions

In addition to the question of how to provide a rate decision, using flow control also impacts how the sender and receiver rate decisions are combined. With flow control, the final sender window is the minimum of the sender's congestion window and the receiver's advertised window. A minimum is conservative: the final rate decision will not exceed the decision of either endpoint. This is appropriate for the original goal of flow control, which is to guarantee that the sender will not exceed the receiver's available memory.

However, for general dual-endpoint control, this is not the only valid model. Sometimes, it is useful for the sender to transmit faster if it is not the bottleneck. In certain scenarios, such as in optical networks where circuits between endpoints are established according to a schedule, the sender must immediately ramp up to full line rate when a circuit becomes available [24]. Also, suppose that the receiver temporarily has CPU available and wants to turbocharge receiving data to compute on. Based on client priority, the sender may be able to temporarily tolerate allocating more bandwidth to a particular receiver. For a more in-depth example, suppose that the sender is implementing a token bucket filter where its rate decision controls the token replacement rate. It may be useful to support a burst mode where a flow can quickly burn accumulated tokens as quickly as the sender can offer, but should still respect the receiver's rate decision during the burst mode. Once the burst tokens have been exhausted, the sender can then return to its own rate decision. Also, as discussed above, a rate decision is not the only useful information for the receiver to communicate to the sender. With access to the receiver's priority for each flow, the sender can schedule flows itself, far beyond what is expressible through the minimum of two rates.

6.2.2 Involving the application layer

By focusing broadly on the considerations and policies of both endpoints, dual-endpoint control raises questions that naturally extend beyond the transport layer into the purview of applications, workloads, and users. We see this in two of our case studies. In Chapter 4, the implementation of the userspace scheduler assumes that the receiver sends requests that then lead to bursts, that it can identify these requests, and that it can decode them to determine the exact data size of the forthcoming burst. Other scheduling policies based on priorities depend on knowing the relative importance of flows, which is inherently an application consideration. The design of applications and how they use incoming data also impacts the receiver's objectives in dual-endpoint control. For example, an application that processes data incrementally may be best served by a model where all flows make some progress, like is achieved by the grant-based round robin scheduling policy. Other applications that must read a flow's entire message before proceeding may instead benefit from serializing flows to enable better network-compute pipelining. In Chapter 5, the FlowPolicy and ServicePolicy designs, while implemented at the transport layer, encode policies that are based on user requirements for fairness at different granularities. One takeaway from these examples is that no transport layer solution is one-size-fits-all from the application perspective.

In our case studies, the transport layer makes assumptions about application behavior, but a more advanced design would provide explicit mechanisms to achieve cross-layer optimization by passing limited application preferences to the transport layer. This could take place in several ways. The application could provide priorities (which could be based on its deadlines) or per-flow weights to the transport layer, either through a socket option (i.e., `setsockopt()` [80]) if the design is based on Linux or through a custom API if using a userspace network stack. To expose certain policies to users (such as those in Chapter 5), the operating system (or application) could expose common models like service-level or flow-level fairness through network preferences.

6.2.2.1 In-network policies

Dual-endpoint control focuses on the objectives of the sender and receiver, but, as discussed in Chapter 1, these are not the only entities to impact rate control; network switches and middleboxes also shape and load balance flows based on the policies of the network operator. Examples include an Internet service provider rate-limiting a customer to their contracted bandwidth, an intrusion detection system dropping malicious packets, or a switch performing fair queuing. Even under dual-endpoint control, flows are still subject to these in-network controls. In a heterogeneous environment, this is likely unavoidable. Dual-endpoint control allows the endpoints to approximate similar behaviors, such as emulating fair queuing or clamping an aggressive CCA, as we show in this dissertation.

In a controlled situation like a datacenter, it is possible to extend the design of dual-endpoint control to explicitly include the objectives of in-network devices as well. By doing so, the sender could avoid congesting the network with packets that would eventually be dropped due to in-network policies. In a method similar to how switches use INT to attach queue statistics to packets, as in HPCC [72], switches could instead attach rate decisions. As a packet travels through the network, it could accumulate a vector of rate decisions from the path of network devices that it traversed. A switch’s rate decision could be based on congestion, policy, or enforcement considerations. The receiver could then reflect this rate vector in an ACK, and the sender, which decides the final transmission rate, could then incorporate these in-network decisions as well as the receiver’s. While such a design would require switch support, it would both create an end-to-end cooperative rate control and reduce the burden on the network in regards to sending traffic that may eventually be discarded. Also, incorporating a full rate decision vector allows the sender to build a more complete model of network behavior, similar to the goal of INT.

6.3 Other Relevant Trends

Two research trends have developed in recent years that have interesting implications for our dual-endpoint approach: the growing prevalence of receiver feedback in congestion control decisions and emerging studies on congestion control fairness on the Internet.

6.3.1 Involving receivers in congestion control

Researchers are increasingly involving the receiver in new congestion control techniques, in terms of host congestion, flow scheduling, and for new workloads like machine learning training. This proliferation of receiver-aware projects attests to the timely nature of our work on dual-endpoint control. In fact, dual-endpoint control represents a generalization of these techniques, as many of the ideas described in these projects could be implemented using dual-endpoint control.

For the past several years, researchers have focused on the implications of intra-host congestion on end-to-end rate allocation [2, 3, 103, 115, 116]. In particular, RDMA intra-Host Congestion Control (RHCC) [116] is similar to dual-endpoint control because it calculates a receiver rate, based on measurements of the receiver host latency, and a sender rate, based on congestion control, and combines them into final rate that is the minimum of the two. Host latency, I/O bandwidth, buffer occupancy, and PCIe credits are examples of host signals that could inform a receiver rate under dual-endpoint control.

Recent schemes for datacenter congestion control also leverage receiver scheduling. The Sender-Informed, Receiver-Driven (SIRD) [98] transport protocol is a receiver-centric approach that issues credits to senders. SIRD uses a dual control loop that combines congestion signals from shared fabric links (to react to other flows) with exclusive scheduling of the receiver’s last-hop link (which is solely the purview of a single receiver). This is an example of collecting state to produce a single rate decision, but at the receiver instead of the sender. This allows SIRD to still react to congestion that is not at the last-hop, which is the same problem that dual-endpoint control solves by still including the sender’s rate

decision.

In the emerging field of transport protocols for distributed machine learning systems, the feedback of the receiver is crucial as well. The Multipath Reliable Connection (MRC) [106] transport protocol for RDMA includes several mechanisms for the receiver to restrict the sender rate. It provides a window based on available receive memory, similar to TCP flow control, and also includes a limit on the number of in-flight write-immediate messages to control receiver multiplexing. For congestion control, the receiver provides its internal processing latency to help the sender accurately calculate just the network latency. MRC also supports receiver back-pressure due to congestion in the memory subsystem, which is incorporated into the sender’s window control. These techniques fall into the category of sending state to the sender to improve its rate decision, whereas dual-endpoint control advocates for the receiver to also use this state to make its own rate decision and then pass that to the sender as well. The dual-endpoint approach would simplify MRC’s receiver feedback from four signals to two: a rate and an in-flight write-immediate limit.

Together with our case studies on dual-endpoint control, this breadth of receiver-focused work solidifies the argument that the receiver deserves a first-class role in congestion control. Our work goes one step further to clarify that this does not mean simply conveying more state or moving control from the sender to the receiver, but instead requires creating a unified rate allocation that encompasses both endpoints’ considerations.

6.3.2 Internet fairness studies

Chapter 5 extends dual-endpoint control into the domain of congestion control fairness on the Internet. This work intersects with many open questions regarding CCAs on the Internet: which CCAs are deployed by Internet services, what are their fairness properties, and how do those properties interact across different CCAs? The genesis of this line of work was earlier investigations that revealed significant unfairness on the part of BBR [23] to classic loss-based CCAs like Cubic [118], leading to new metrics to quantify the harm one CCA causes to another [119]. Later work showed that traditional congestion control techniques break down at the scale of flows seen on the Internet [96]. This culminated in a set of studies that classify the congestion control algorithms deployed for Internet services and how they interact [66, 97, 120]. To summarize their findings, the set of CCAs deployed on the Internet is diverse and growing, with new techniques such as the BBR family of protocols taking a large share of the landscape. This diverse set of relatively new protocols often demonstrate unfairness between flows. In an effort to bring structure to this bandwidth battleground, a recent proposal [104] explored how congestion safety audits can help service providers ensure their CCAs share bandwidth equitably with competitors.

Dual-endpoint control introduces a new model under which to think about this Internet congestion control landscape. This dissertation shows how dual-endpoint control can detect and mitigate unfairness between CCAs with different granularities of fairness, metrics, and convergence algorithms. In this diverse CCA battleground, dual-endpoint control provides Internet clients, which are typically the receiver, with a mechanism to control the service model that their flows follow. For both accidental fairness and intentional design differences, if dual-endpoint control were deployed widely, it could create a new Internet landscape governed not only by the service’s choice of CCA or the network operator’s rate control but also by the clients’ individual preferences for how their flows should interact.

6.4 Additional Future Directions

Section 6.2 captures several potential future directions through its exploration of the limitations of flow control, the benefits of involving the application layer and in-network policies, and considerations for a clean-slate design. To review, these directions include incorporating richer rate decisions, combining

endpoint rate decisions using mechanisms other than a minimum, providing application priorities to the transport layer, and assembling a vector of in-network rate decisions, among others. This section describes a selection of additional directions.

This dissertation has followed a progression from controlled to less controlled environments. Starting in the datacenter environment with a particular workload, incast bursts, Chapters 2 and 3 focused on specific workload patterns and assumptions that enabled a simple dual-endpoint technique in production. Chapter 4 expanded this approach by making it dynamic and applicable to both network and host resource control. Chapter 5 left the datacenter behind to show how dual-endpoint control can express user policies regarding service and flow fairness in a much more heterogeneous environment, the Internet, with a greater variety of congestion control algorithms. Continuing this line of thought, this raises the question of what is the next environment for dual-endpoint control? We describe two paths forwards: expanding user controls and representing in-network controls via the endpoints.

6.4.1 A novel Internet environment

Dual-endpoint control creates a novel Internet environment. Section 6.3.2 described how the Internet environment is inundated with novel CCAs that often do not play fairly with incumbents. Previously, Internet clients had no recourse to control untested, aggressive, or misconfigured CCAs; all control fell to services, which balance across their clients but are ignorant of per-client preferences, and network operators, who observe congestion but are oblivious to both service and client objectives. Dual-endpoint control levels the playing field by giving clients a stake as well. In light of the work characterizing the state of congestion control on the Internet [66, 97, 118, 120], this invites followups to show how dual-endpoint control can detect and mediate these coexistence challenges. Building on [104], should a CCA fail a congestion safety audit, then dual-endpoint control offers a mechanism to force the offending service into compliant behavior.

6.4.2 In-network policies

Dual-endpoint control allows end hosts to express high-level policies that are typically provided by in-network controls. This is possible in two ways: (1) by directly incorporating switch rate decisions as described in Section 6.2.2.1, or (2) by using rich endpoint policies to approximate in-network techniques without direct switch feedback. The former, as noted above, expands the dual-endpoint rate decision to be end-to-end. This enables improved bandwidth efficiency by not transmitting packets that would be dropped by a switch policy, and also allows better analysis of the behavior of the final rate decision because all requisite sub-decisions are available for review. Whereas this design extension explicitly includes in-network rate decisions, this is of course challenging to implement because it requires switch support to communicate a rate alongside additional markings like ECN and INT.

Instead, Chapter 5 showed how in-network controls such as fair queuing can be approximated through dual-endpoint control with careful receiver monitoring of incoming packet behavior. ServicePolicy approximates fair queuing at the level of services, while FlowPolicy does so at the level of flows. Observing the packet stream, the receiver was able to infer the behavior of a remote bottleneck link and use that information to make a rate decision. Other policies are possible as well, such as token bucket filters, application specific prioritization, enforcing per-user rate allocations, and certain network functions. In environments like the Internet where the uniform deployment of network functions is challenging due to heterogeneous control domains, dual-endpoint control provides a way to push functionality out of the network core to the endpoints.

6.5 Closing Remarks

Congestion control has been an important topic in networking research for forty years, and this work is far from the end of the story. What is clear is that both networks and workloads have evolved to a scale where traditional single-endpoint techniques are insufficient to produce a rate decision that is suitable for all entities involved. At the same time, new approaches to congestion control, including and beyond those in this dissertation, represent a new era of opportunities to incorporate more expressive information in rate decisions, from endpoint objectives to workload characteristics, rich in-network metrics, and more. Our work shows that dual-endpoint control is a useful step towards that vision.

Bibliography

- [1] Vamsi Addanki, Oliver Michel, and Stefan Schmid. PowerTCP: Pushing the performance limits of datacenter networks. In *USENIX NSDI 2022*, pages 51–70, Renton, WA, April 2022. USENIX Association. ISBN 978-1-939133-27-4. URL <https://www.usenix.org/conference/nsdi22/presentation/addanki>. 1.1, 1.2.2.2, 2.3.3, 2.5.2, 3.1
- [2] Saksham Agarwal, Rachit Agarwal, Behnam Montazeri, Masoud Moshref, Khaled Elmeleegy, Luigi Rizzo, Marc Asher de Kruijf, Gautam Kumar, Sylvia Ratnasamy, David Culler, and Amin Vahdat. Understanding host interconnect congestion. In *ACM HotNets 2022*, page 198–204, New York, NY, 2022. ACM. ISBN 9781450398992. doi: 10.1145/3563766.3564110. URL <https://doi.org/10.1145/3563766.3564110>. 1.1.3, 4.1, 4.4.4, 5.7, 6.3.1
- [3] Saksham Agarwal, Arvind Krishnamurthy, and Rachit Agarwal. Host congestion control. In *ACM SIGCOMM 2023*, page 275–287, New York, NY, 2023. ACM. ISBN 9798400702365. doi: 10.1145/3603269.3604878. URL <https://doi.org/10.1145/3603269.3604878>. 1.1.3, 1.3.4.1, 4.1, 1, 4.2.3.3, 4.4.4, 5.7, 6.3.1
- [4] Aditya Akella, Srinivasan Seshan, and Hari Balakrishnan. The impact of false sharing on shared congestion management. In *11th IEEE International Conference on Network Protocols, 2003. Proceedings.*, pages 84–94, 2003. doi: 10.1109/ICNP.2003.1249759. 5.4.2
- [5] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data Center TCP (DCTCP). *ACM SIGCOMM CCR*, 40(4):63–74, August 2010. ISSN 0146-4833. doi: 10.1145/1851275.1851192. URL <https://doi.org/10.1145/1851275.1851192>. 1.1, 1.1.1, 1.2.1, 2.1, 2.2, 2.3.3, 2.3.4, 2.4, 2.4.1.3, 3.4.3, 4.3, 4.3.2.1, 4.4.1, 4.6.1, 5.6.2.2
- [6] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less is more: Trading a little bandwidth for ultra-low latency in the data center. In *USENIX NSDI 2012*, pages 253–266, San Jose, CA, April 2012. USENIX Association. ISBN 978-931971-92-8. URL <https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/alizadeh>. 5.3.2, 5.6.1.1
- [7] Serhat Arslan, Yuliang Li, Gautam Kumar, and Nandita Dukkkipati. Bolt: Sub-RTT congestion control for ultra-low latency. In *USENIX NSDI 2023*, pages 219–236, Boston, MA, April 2023. USENIX Association. ISBN 978-1-939133-33-5. URL <https://www.usenix.org/conference/nsdi23/presentation/arslan>. 2.3.3, 2.5.2
- [8] Marcelo Bagnulo, Alberto Garcia-Martinez, Gabriel Montenegro, and Praveen Balasubramanian. rLEDBAT: Receiver-driven low extra delay background transport for TCP. Internet-Draft draft-irtf-icrg-rledbat-09, Internet Engineering Task Force, November 2024. URL <https://datatracker.ietf.org/doc/draft-irtf-icrg-rledbat/09/>. Work in Progress. 1.3.3
- [9] Fred Baker, David L. Black, Kathleen Nichols, and Steven L. Blake. Definition of the differentiated

- services field (DS field) in the IPv4 and IPv6 headers. RFC 2474, December 1998. URL <https://www.rfc-editor.org/info/rfc2474>. 5.1
- [10] Hari Balakrishnan and Srinivasan Seshan. The Congestion Manager. Internet-Draft draft-balakrishnan-cm-03, Internet Engineering Task Force, March 2000. URL <https://datatracker.ietf.org/doc/draft-balakrishnan-cm/03/>. Work in Progress. 5.2.2.1, 5.7
 - [11] Hari Balakrishnan, Venkata N. Padmanabhan, Srinivasan Seshan, Mark Stemm, and Randy H. Katz. TCP behavior of a busy Internet server: Analysis and improvements. In *IEEE INFOCOM*, volume 1, pages 252–262 vol.1, 1998. doi: 10.1109/INFCOM.1998.659661. 5.2.2.1, 5.7
 - [12] Hari Balakrishnan, Hariharan S. Rahul, and Srinivasan Seshan. An integrated congestion management architecture for Internet hosts. *ACM SIGCOMM CCR*, 29(4):175–187, August 1999. ISSN 0146-4833. doi: 10.1145/316194.316220. URL <https://doi.org/10.1145/316194.316220>. 5.2.2.1, 5.4.2, 5.7
 - [13] Theophilus Benson, Aditya Akella, and David A. Maltz. Network traffic characteristics of data centers in the wild. In *ACM IMC 2010*, page 267–280, New York, NY, USA, 2010. ACM. ISBN 9781450304832. doi: 10.1145/1879141.1879175. URL <https://doi.org/10.1145/1879141.1879175>. 2.3.3
 - [14] Ethan Blanton, Vern Paxson, and Mark Allman. TCP congestion control. RFC 5681, September 2009. URL <https://www.rfc-editor.org/info/rfc5681>. 1.1, 2.4, 2.4.1.3, 2.5.2, 3.3.2
 - [15] David Borman, Robert T. Braden, Van Jacobson, and Richard Scheffenegger. TCP extensions for high performance. RFC 7323, September 2014. URL <https://www.rfc-editor.org/info/rfc7323>. 5.8.2
 - [16] David A. Borman, Robert T. Braden, and Van Jacobson. TCP extensions for high performance. RFC 1323, May 1992. URL <https://www.rfc-editor.org/info/rfc1323>. 3.3.3
 - [17] Lawrence S. Brakmo, Sean W. O’Malley, and Larry L. Peterson. TCP Vegas: New techniques for congestion detection and avoidance. In *ACM SIGCOMM 1994*, page 24–35, New York, NY, 1994. ACM. ISBN 0897916824. doi: 10.1145/190314.190317. URL <https://doi.org/10.1145/190314.190317>. 1.2.1, 3.4.3, 4.4.1, 5.1, 5.2.2.3
 - [18] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001. doi: 10.1023/A:1010933404324. URL <https://doi.org/10.1023/A:1010933404324>. 5.5.2.2
 - [19] Bob Briscoe. Flow rate fairness: Dismantling a religion. *ACM SIGCOMM CCR*, 37(2):63–74, March 2007. ISSN 0146-4833. doi: 10.1145/1232919.1232926. URL <https://doi.org/10.1145/1232919.1232926>. 6. 5.1
 - [20] Qizhe Cai, Shubham Chaudhary, Midhul Vuppapapati, Jaehyun Hwang, and Rachit Agarwal. Understanding host network stack overheads. In *ACM SIGCOMM 2021*, page 65–77, New York, NY, 2021. ACM. ISBN 9781450383837. doi: 10.1145/3452296.3472888. URL <https://doi.org/10.1145/3452296.3472888>. 1.1.3, 1.2.2.2, 4.1, 4.4.4, 5.7
 - [21] Christopher Canel, Zecheng He, Elliot Lockerman, Adithya Abraham Philip, Justine Sherry, and Srinivasan Seshan. Cooperative rate control for the Internet. Technical Report CMU-CS-24-129, Computer Science Department, School of Computer Science, Carnegie Mellon University, May 2014. URL <http://reports-archive.adm.cs.cmu.edu/anon/2024/CMU-CS-24-129.pdf>. 5.1
 - [22] Christopher Canel, Balasubramanian Madhavan, Srikanth Sundaresan, Neil Spring, Prashanth Kannan, Ying Zhang, Kevin Lin, and Srinivasan Seshan. Understanding incast bursts in modern datacenters. In *Proceedings of the 2024 ACM on Internet Measurement Conference*, IMC ’24, page

- 674–680, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400705922. doi: 10.1145/3646547.3689028. URL <https://doi.org/10.1145/3646547.3689028>. 2.1, 3.4.0.1
- [23] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. BBR: Congestion-based congestion control. *ACM Queue*, 14, September–October:20 – 53, 2016. URL <http://queue.acm.org/detail.cfm?id=3022184>. 1.1.4, 1.2.1, 4.3.2.1, 5.6.2.1, 6.3.2
 - [24] Shawn Shuoshuo Chen, Weiyang Wang, Christopher Canel, Srinivasan Seshan, Alex C. Snoeren, and Peter Steenkiste. Time-division TCP for reconfigurable data center networks. In *ACM SIGCOMM 2022*, page 19–35, New York, NY, 2022. ACM. ISBN 9781450394208. doi: 10.1145/3544216.3544254. URL <https://doi.org/10.1145/3544216.3544254>. 6.2.1.2
 - [25] Yanpei Chen, Rean Griffith, Junda Liu, Randy H. Katz, and Anthony D. Joseph. Understanding TCP incast throughput collapse in datacenter networks. In *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking*, WREN ’09, page 73–82, New York, NY, 2009. ACM. ISBN 9781605584430. doi: 10.1145/1592681.1592693. URL <https://doi.org/10.1145/1592681.1592693>. 2.1, 2.3.3
 - [26] Dah-Ming Chiu and Raj Jain. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Computer Networks and ISDN Systems*, 17(1):1–14, June 1989. ISSN 0169-7552. doi: 10.1016/0169-7552(89)90019-6. URL [https://doi.org/10.1016/0169-7552\(89\)90019-6](https://doi.org/10.1016/0169-7552(89)90019-6). 4.3.2.2, 5.1, 5.2.2.2
 - [27] Inho Cho, Keon Jang, and Dongsu Han. Credit-scheduled delay-bounded congestion control for datacenters. In *ACM SIGCOMM 2017*, page 239–252, New York, NY, 2017. ACM. ISBN 9781450346535. doi: 10.1145/3098822.3098840. URL <https://doi.org/10.1145/3098822.3098840>. 1.2.2.3, 2.3.3, 2.5.1, 2.5.2, 4.1, 4.4.2.2
 - [28] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. Efficient coflow scheduling with Varys. In *ACM SIGCOMM 2014*, page 443–454, New York, NY, 2014. ACM. ISBN 9781450328364. doi: 10.1145/2619239.2626315. URL <https://doi.org/10.1145/2619239.2626315>. 1.1.3, 2.5.1, 4.1, 4.4.2.1
 - [29] David D. Clark, John Wroclawski, Karen R. Sollins, and Robert Braden. Tussle in cyberspace: Defining tomorrow’s Internet. In *ACM SIGCOMM 2002*, page 347–356, New York, NY, 2002. ACM. ISBN 158113570X. doi: 10.1145/633025.633059. URL <https://doi.org/10.1145/633025.633059>. 5.1
 - [30] D.D. Clark. Window and acknowledgement strategy in TCP. RFC 813, July 1982. URL <https://www.rfc-editor.org/info/rfc813/>. 4.5.2.8
 - [31] CloudLab. The CloudLab manual - 13.3 CloudLab clemson. [https://docs.cloudlab.us/cloudlab-manual.html#\(part,_cloudlab-clemson\)](https://docs.cloudlab.us/cloudlab-manual.html#(part,_cloudlab-clemson)), 2026. 4.6.1
 - [32] CloudLab. CloudLab. <https://cloudlab.us>, 2026. 4.6.1
 - [33] Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and simulation of a fair queueing algorithm. In *ACM SIGCOMM 1989*, page 1–12, New York, NY, 1989. ACM. ISBN 0897913329. doi: 10.1145/75246.75248. URL <https://doi.org/10.1145/75246.75248>. 1.1.2, 1.2.2.1, 3.1, 5.1, 5.2.2.1
 - [34] Abhishek Dhamija, Balasubramanian Madhavan, Hechao Li, Jie Meng, Shrikrishna Khare, Madhavi Rao, Lawrence Brakmo, Neil Spring, Prashanth Kannan, Srikanth Sundaresan, and Soudeh Ghorbani. A large-scale deployment of DCTCP. In *USENIX NSDI 2024*, pages 239–252, Santa Clara, CA, April 2024. USENIX Association. ISBN 978-1-939133-39-7. URL <https://www.usenix.org/conference/nsdi24/presentation/dhamija>. 2.2
 - [35] The Linux Kernel Documentation. Segmentation offloads. <https://www.kernel.org/doc/html/latest/networking/segmentation-offloads.html>, 2023. 1.2.2.2, 2.5.2, 4.1, 4.3.2.2

- [36] eBPF. eBPF - introduction, tutorials & community resources. <https://ebpf.io>, September 2022. 2.3.2, 3.1, 3.3.1, 4.5.2, 4.5.3, 5.3.3.1, 5.8.2
- [37] Wesley Eddy. Transmission control protocol (TCP). RFC 9293, August 2022. URL <https://www.rfc-editor.org/info/rfc9293>. 1.3.2, 2.1, 4.5.4
- [38] ESnet. iperf3 3.21 documentation. <https://software.es.net/iperf/>, 2026. 4.5.1
- [39] Sally Floyd. HighSpeed TCP for large congestion windows. RFC 3649, December 2003. URL <https://www.rfc-editor.org/info/rfc3649>. 5.6.2.2
- [40] Sally Floyd. Metrics for the evaluation of congestion control mechanisms. RFC 5166, March 2008. URL <https://www.rfc-editor.org/info/rfc5166>. 5.1
- [41] Sally Floyd and Kevin Fall. Promoting the use of end-to-end congestion control in the Internet. *IEEE/ACM Transactions on Networking*, 7(4):458–472, August 1999. ISSN 1063-6692. doi: 10.1109/90.793002. URL <https://doi.org/10.1109/90.793002>. 1.2.2.1
- [42] Sally Floyd, Jamshid Mahdavi, Matthew Mathis, and Dr. Allyn Romanow. TCP selective acknowledgment options. RFC 2018, October 1996. URL <https://www.rfc-editor.org/info/rfc2018>. 3.3.2
- [43] Sally Floyd, K. K. Ramakrishnan, and David L. Black. The addition of explicit congestion notification (ECN) to IP. RFC 3168, September 2001. URL <https://www.rfc-editor.org/info/rfc3168>. 1.1, 2.2
- [44] Jerome Friedman. Greedy function approximation: A gradient boosting machine. *The Annals of Statistics*, 29, November 2000. doi: 10.1214/aos/1013203451. 5.5.2.3
- [45] Eli Gafni and Dimitri Bertsekas. Dynamic control of session input rates in communication networks. *IEEE Transactions on Automatic Control*, 29(11):1009–1016, 1984. doi: 10.1109/TAC.1984.1103431. 5.2.1
- [46] Peter X. Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. PHost: Distributed near-optimal datacenter transport over commodity network fabric. In *ACM CoNEXT 2015*, New York, NY, 2015. ACM. ISBN 9781450334129. doi: 10.1145/2716281.2836086. URL <https://doi.org/10.1145/2716281.2836086>. 1.2.2.3, 2.3.3, 2.5.1, 2.5.2, 4.1, 4.4.2.2
- [47] Ehab Ghabashneh, Yimeng Zhao, Cristian Lumezanu, Neil Spring, Srikanth Sundaresan, and Sanjay Rao. A microscopic view of bursts, buffer contention, and loss in data centers. In *ACM IMC 2022*, page 567–580, New York, NY, 2022. ACM. ISBN 9781450392594. doi: 10.1145/3517745.3561430. URL <https://doi.org/10.1145/3517745.3561430>. 2.1, 2.2, 2.3.1, 2.3.2, 2.3.3, 2.3.4
- [48] The Tcpdump Group. tcpdump & libpcap. <https://www.tcpdump.org>, June 2023. 5.3.3.1
- [49] Rajarshi Gupta. WEBTP: A user-centric receiver-driven web transport protocol. Technical Report UCB/ERL M98/77, EECS Department, University of California, Berkeley, Dec 1998. URL <http://www2.eecs.berkeley.edu/Pubs/TechRpts/1998/3562.html>. 1.2.2.3
- [50] Andrei Gurtov, Tom Henderson, and Sally Floyd. The NewReno modification to TCP’s fast recovery algorithm. RFC 3782, April 2004. URL <https://www.rfc-editor.org/info/rfc3782>. 1.2.1, 4.3.2.1
- [51] Sangtae Ha, Injong Rhee, and Lisong Xu. CUBIC: A new TCP-friendly high-speed TCP variant. *ACM SIGOPS Operating Systems Review*, 42(5):64–74, July 2008. ISSN 0163-5980. doi: 10.1145/1400097.1400105. URL <https://doi.org/10.1145/1400097.1400105>. 1.1, 1.2.1, 3.1, 3.4.3, 3.4.3.1, 4.2.3.1, 4.3.2.1, 4.4.1, 4.6.1, 5.1, 5.2.2.2
- [52] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy.

- SoftNIC: A software nic to augment hardware. Technical Report UCB/EECS-2015-155, EECS Department, University of California, Berkeley, May 2015. URL <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-155.html>. 5.5.2.1
- [53] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *ACM SIGCOMM 2017*, page 29–42, New York, NY, 2017. ACM. ISBN 9781450346535. doi: 10.1145/3098822.3098825. URL <https://doi.org/10.1145/3098822.3098825>. 1.1, 1.2.2.3, 2.3.3, 2.5.1, 2.5.2, 4.1, 4.4.2.2
 - [54] Mark J. Handley, Jitendra Padhye, Sally Floyd, and Joerg Widmer. TCP friendly rate control (TFRC): Protocol specification. RFC 5348, September 2008. URL <https://www.rfc-editor.org/info/rfc5348>. 1.3.4.2, 5.3.3.1, 5.4.2, 5.7, 5.4
 - [55] Keqiang He, Eric Rozner, Kanak Agarwal, Yu (Jason) Gu, Wes Felter, John Carter, and Aditya Akella. AC/DC TCP: Virtual congestion control enforcement for datacenter networks. In *ACM SIGCOMM 2016*, page 244–257, New York, NY, 2016. ACM. ISBN 9781450341936. doi: 10.1145/2934872.2934903. URL <https://doi.org/10.1145/2934872.2934903>. 1.3.3
 - [56] Jörn-Thorben Hinz, Vamsi Addanki, Csaba Gyöngyi, Theo Jepsen, and Stefan Schmid. TCP’s third eye: Leveraging eBPF for telemetry-powered congestion control. In *Proceedings of the 1st Workshop on EBPF and Kernel Extensions, eBPF ’23*, page 1–7, New York, NY, 2023. ACM. ISBN 9798400702938. doi: 10.1145/3609021.3609295. URL <https://doi.org/10.1145/3609021.3609295>. 1.2.2.2
 - [57] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, December 1997. doi: 10.1162/neco.1997.9.8.1735. 5.5.2.2
 - [58] IO Visor. BPF compiler collection (BCC). <https://github.com/iovisor/bcc>, September 2022. 5.8.2
 - [59] Jana Iyengar and Martin Thomson. QUIC: A UDP-based multiplexed and secure transport. RFC 9000, May 2021. URL <https://www.rfc-editor.org/info/rfc9000>. 5.2.2.1, 5.7
 - [60] Van Jacobson. Congestion avoidance and control. In *ACM SIGCOMM 1988*, page 314–329, New York, NY, 1988. ACM. ISBN 0897912799. doi: 10.1145/52324.52356. URL <https://doi.org/10.1145/52324.52356>. 1.2.1, 4.3.2.2
 - [61] Van Jacobson. Modified TCP congestion avoidance algorithm. Email to end2end-interest Mailing List. Obtain via <ftp://ftp.ee.lbl.gov/email/vanj.90apr30.txt>, April 1990. 1.2.1, 5.1
 - [62] Raj Jain and K. K. Ramakrishnan. Congestion avoidance in computer networks with a connection-less network layer: Concepts, goals and methodology. In *[1988] Proceedings. Computer Networking Symposium*, pages 134–143, 1988. doi: 10.1109/CNS.1988.4990. 5.2.2.2
 - [63] Rajendra K. Jain, Dah-Ming W. Chiu, and William R. Hawe. A quantitative measure of fairness and discrimination. *Eastern Research Laboratory, Digital Equipment Corporation, Hudson, MA*, 21, 1984. 1.1.4, 5.6
 - [64] Raj Joshi, Ting Qu, Mun Choon Chan, Ben Leong, and Boon Thau Loo. Burstradar: Practical real-time microburst monitoring for datacenter networks. In *ACM APSys 2018*, New York, NY, USA, 2018. ACM. ISBN 9781450360067. doi: 10.1145/3265723.3265731. URL <https://doi.org/10.1145/3265723.3265731>. 2.3.3
 - [65] Lampros Kalampoukas, Anujan Varma, and K. K. Ramakrishnan. Explicit window adaptation: A method to enhance TCP performance. In *IEEE INFOCOM*, volume 1, pages 242–251 vol.1, 1998. doi: 10.1109/INFCOM.1998.659660. 1.2.2.3

- [66] Darshil D. Kaneria, Ranysha Ware, and Srinivasan Seshan. VCCAnalyzer: Identifying congestion control algorithms used by video streaming services. *Proc. ACM Netw.*, 3(CoNEXT4), November 2025. doi: 10.1145/3768979. URL <https://doi.org/10.1145/3768979>. 6.3.2, 6.4.1
- [67] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. LightGBM: A highly efficient gradient boosting decision tree. In *NIPS 2017*, page 3149–3157, Red Hook, NY, 2017. Curran Associates Inc. ISBN 9781510860964. 5.5.2.3
- [68] Frank Kelly. Charging and rate control for elastic traffic. *European Transactions on Telecommunications*, 8, February 1997. doi: 10.1002/ett.4460080106. 5.7
- [69] Stephen Kokoska and Daniel Zwillinger. *CRC Standard Probability and Statistics Tables and Formulae (1st ed.)*. CRC Press, 2000. 5.5.2.2
- [70] Elie Krevat, Vijay Vasudevan, Amar Phanishayee, David G. Andersen, Gregory R. Ganger, Garth A. Gibson, and Srinivasan Seshan. On application-level approaches to avoiding TCP throughput collapse in cluster-based storage systems. In *Proceedings of the 2nd International Workshop on Petascale Data Storage: Held in Conjunction with Supercomputing '07*, PDSW '07, page 1–4, New York, NY, 2007. ACM. ISBN 9781595938992. doi: 10.1145/1374596.1374598. URL <https://doi.org/10.1145/1374596.1374598>. 2.1
- [71] Gautam Kumar, Nandita Dukkupati, Keon Jang, Hassan M. G. Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. Swift: Delay is simple and effective for congestion control in the datacenter. In *ACM SIGCOMM 2020*, page 514–528, New York, NY, 2020. ACM. ISBN 9781450379557. doi: 10.1145/3387514.3406591. URL <https://doi.org/10.1145/3387514.3406591>. 1.1.1, 2.3.2, 2.3.3, 2.5.2, 3.1, 3.4.3, 4.3, 4.3.1, 4.3.2.1, 4.4.1
- [72] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. HPCC: High precision congestion control. In *ACM SIGCOMM 2019*, page 44–58, New York, NY, 2019. ACM. ISBN 9781450359566. doi: 10.1145/3341302.3342085. URL <https://doi.org/10.1145/3341302.3342085>. 1.1, 1.2.1, 1.2.2.2, 2.3.3, 2.5.2, 3.1, 3.4.3, 4.3, 6.2.2.1
- [73] Shiyu Liu, Ahmad Ghalayini, Mohammad Alizadeh, Balaji Prabhakar, Mendel Rosenblum, and Anirudh Sivaraman. Breaking the Transience-Equilibrium nexus: A new approach to datacenter packet transport. In *USENIX NSDI 2021*, pages 47–63. USENIX Association, April 2021. ISBN 978-1-939133-21-2. URL <https://www.usenix.org/conference/nsdi21/presentation/liu>. 2.1, 2.3.3, 4.4.3
- [74] Linux man-pages 6.18. accept(2) — Linux manual page. <https://man7.org/linux/man-pages/man2/accept.2.html>, 2026. 4.5.2.1, 2
- [75] Linux man-pages 6.18. connect(2) — Linux manual page. <https://man7.org/linux/man-pages/man2/connect.2.html>, 2026. 2
- [76] Linux man-pages 6.18. epoll(7) — Linux manual page. <https://man7.org/linux/man-pages/man7/epoll.7.html>, 2026. 4.5.1
- [77] Linux man-pages 6.18. recv(2) — Linux manual page. <https://man7.org/linux/man-pages/man2/recv.2.html>, 2026. 4.6.1
- [78] Linux man-pages 6.18. select(2) — Linux manual page. <https://man7.org/linux/man-pages/man2/select.2.html>, 2026. 4.5.1
- [79] Linux man-pages 6.18. send(2) — Linux manual page. <https://man7.org/linux/man-pages/man2/send.2.html>, 2026. 4.6.1

endmsg.2.html, 2026. 4.5.2.2

- [80] Linux man-pages 6.18. getsockopt(2) — Linux manual page. <https://man7.org/linux/man-pages/man2/setsockopt.2.html>, 2026. 6.2.2
- [81] Matthew Mathis, Jeffrey Semke, Jamshid Mahdavi, and Teunis Ott. The macroscopic behavior of the TCP congestion avoidance algorithm. *ACM SIGCOMM CCR*, 27(3):67–82, July 1997. ISSN 0146-4833. doi: 10.1145/263932.264023. URL <https://doi.org/10.1145/263932.264023>. 5.1, 5.4, 5.4.1, 5.4.2, 5.4
- [82] MEGA. Home - mega. <https://mega.io>, February 2024. 5.2.2.1
- [83] Jianyu Miao and Lingfeng Niu. A survey on feature selection. *Procedia Computer Science*, 91: 919–926, December 2016. doi: 10.1016/j.procs.2016.07.111. 5.5.2.2
- [84] Radhika Mittal, Vinh The Lam, Nandita Dukkkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. TIMELY: RTT-based congestion control for the datacenter. In *ACM SIGCOMM 2015*, page 537–550, New York, NY, 2015. ACM. ISBN 9781450335423. doi: 10.1145/2785956.2787510. URL <https://doi.org/10.1145/2785956.2787510>. 1.2.1, 4.4.1
- [85] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *ACM SIGCOMM 2018*, page 221–235, New York, NY, 2018. ACM. ISBN 9781450355674. doi: 10.1145/3230543.3230564. URL <https://doi.org/10.1145/3230543.3230564>. 1.1, 1.1.3, 1.2.2.3, 2.3.3, 2.5.1, 2.5.2, 4.1, 4.3.3, 4.4.2.2
- [86] Matthew K. Mukerjee. Eliminating adverse control plane interactions in independent network systems. Technical Report CMU-CS-18-106, Computer Science Department, School of Computer Science, Carnegie Mellon University, May 2018. URL <http://reports-archive.adm.cs.cmu.edu/anonymous/2018/CMU-CS-18-106.pdf>. 1.1
- [87] David Nagle, Denis Serenyi, and Abbie Matthews. The Panasas ActiveScale storage cluster - delivering scalable high bandwidth storage. In *SC '04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, page 53, 2004. doi: 10.1109/SC.2004.57. 2.1, 4.3
- [88] John Nagle. On packet switches with infinite storage. RFC 970, December 1985. URL <https://www.rfc-editor.org/info/rfc970>. 1.2.2.1, 5.1, 5.2.2.1
- [89] Vikram Nathan, Vibhaalakshmi Sivaraman, Ravichandra Addanki, Mehrdad Khani, Prateesh Goyal, and Mohammad Alizadeh. End-to-end transport for video QoE fairness. In *ACM SIGCOMM 2019*, page 408–423, New York, NY, 2019. ACM. ISBN 9781450359566. doi: 10.1145/3341302.3342077. URL <https://doi.org/10.1145/3341302.3342077>. 5.2.1
- [90] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. Understanding PCIe performance for end host networking. In *ACM SIGCOMM 2018*, page 327–341, New York, NY, USA, 2018. ACM. ISBN 9781450355674. doi: 10.1145/3230543.3230560. URL <https://doi.org/10.1145/3230543.3230560>. 4.4.4
- [91] Henrik Nielsen, Jeffrey Mogul, Larry M. Masinter, Roy T. Fielding, Jim Gettys, Paul J. Leach, and Tim Berners-Lee. Hypertext transfer protocol – HTTP/1.1. RFC 2616, June 1999. URL <https://www.rfc-editor.org/info/rfc2616>. 5.2.2.1
- [92] ns 3. ns-3: A discrete-event network simulator for Internet systems. <https://www.nsnam.org>, 2020. 2.4, 4.6.1
- [93] Venkata Narayana Padmanabhan. Addressing the challenges of web data transport, 1998. 5.2.2.1,

- [94] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. Fastpass: A centralized “zero-queue” datacenter network. *ACM SIGCOMM CCR*, 44(4):307–318, August 2014. ISSN 0146-4833. doi: 10.1145/2740070.2626309. URL <https://doi.org/10.1145/2740070.2626309>. 1.1.3, 2.5.1, 4.1, 4.4.2.1
- [95] Amar Phanishayee, Elie Krevat, Vijay Vasudevan, David G. Andersen, Gregory R. Ganger, Garth A. Gibson, and Srinivasan Seshan. Measurement and analysis of TCP throughput collapse in cluster-based storage systems. In *USENIX FAST 2008*, USA, 2008. USENIX Association. 2.1
- [96] Adithya Abraham Philip, Ranysha Ware, Rukshani Athapathu, Justine Sherry, and Vyas Sekar. Revisiting TCP congestion control throughput models & fairness properties at scale. In *Proceedings of the 21st ACM Internet Measurement Conference, IMC ’21*, page 96–103, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450391290. doi: 10.1145/3487552.3487834. URL <https://doi.org/10.1145/3487552.3487834>. 6.3.2
- [97] Adithya Abraham Philip, Rukshani Athapathu, Ranysha Ware, Fabian Francis Mkocheke, Alexis Schlomer, Mengrou Shou, Zili Meng, Srinivasan Seshan, and Justine Sherry. Prudentia: Findings of an Internet fairness watchdog. In *Proceedings of the ACM SIGCOMM 2024 Conference, ACM SIGCOMM ’24*, page 506–520, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400706141. doi: 10.1145/3651890.3672229. URL <https://doi.org/10.1145/3651890.3672229>. 1.1.4, 6.3.2, 6.4.1
- [98] Konstantinos Prasopoulos, Ryan Kosta, Edouard Bugnion, and Marios Kogias. SIRD: A sender-informed, receiver-driven datacenter transport protocol. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 451–471, Philadelphia, PA, April 2025. USENIX Association. ISBN 978-1-939133-46-5. URL <https://www.usenix.org/conference/nsdi25/presentation/prasopoulos>. 6.3.1
- [99] The Linux Documentation Project. Introduction to linux traffic control. <https://tldp.org/HOWTO/Traffic-Control-HOWTO/intro.html>, 2023. 2.3.2, 3.3.1
- [100] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the social network’s (datacenter) network. In *ACM SIGCOMM 2015*, page 123–137, New York, NY, USA, 2015. ACM. ISBN 9781450335423. doi: 10.1145/2785956.2787472. URL <https://doi.org/10.1145/2785956.2787472>. 2.3.2, 2.3.3
- [101] Henning Schulzrinne, Stephen L. Casner, Ron Frederick, and Van Jacobson. RTP: A transport protocol for real-time applications. RFC 3550, July 2003. URL <https://www.rfc-editor.org/info/rfc3550>. 5.7
- [102] scikit-learn. Permutation importance with multicollinear or correlated features. https://scikit-learn.org/stable/auto_examples/inspection/plot_permutation_importance_multicollinear.html, September 2022. 5.5.2.2
- [103] Erfan Sharafzadeh, Sepehr Abdous, and Soudeh Ghorbani. Understanding the impact of host networking elements on traffic bursts. In *USENIX NSDI 2023*, pages 237–254, Boston, MA, April 2023. USENIX Association. ISBN 978-1-939133-33-5. URL <https://www.usenix.org/conference/nsdi23/presentation/sharafzadeh>. 1.1.3, 4.1, 4.2.3.2, 6.3.1
- [104] Justine Sherry, Ranysha Ware, Isabel C. Suizo, Srinivasan Seshan, and Adithya Abraham Philip. Congestion safety audits for Internet services. In *Proceedings of the TPRC 53: The 53rd Research Conference on Communication, Information and Internet Policy 2025*, August 2025. URL <http://dx.d>

- [105] Anirudh Sivaraman, Keith Winstein, Pratiksha Thaker, and Hari Balakrishnan. An experimental study of the learnability of congestion control. In *ACM SIGCOMM 2014*, page 479–490, New York, NY, 2014. ACM. ISBN 9781450328364. doi: 10.1145/2619239.2626324. URL <https://doi.org/10.1145/2619239.2626324>. 5.5.2.3
- [106] Rip Sohan, Eric Spada, Eric Davis, Mark Handley, Idan Burstein, Tony Hurson, Jithin Jose, Vivek Kashyap, Rong Pan, Sayantan Sur, Sreevatsa Anantharamu, Aviv Barnea, Adrian Caulfield, Elazar Cohen, Elliot Edmunds, Yamin Friedman, Mahdieh Ghazi, Murali Guramali, Torsten Hoeffler, Vipin Jain, Abdul Kabbani, Noam Katz, Yanfang Le, Charlie Mbariky, Guglielmo Morandin, Masoud Moshref, Shane O’Neil, Michael Papamichael, Jonas Pfefferle, Siva Santosh Pyla, Costin Raiciu, David Riddoch, Karen Schramm, Yuval Shpigelman, Shahaf Shuler, Shy Shyman, Raghava Sivaramu, Amin Tootoonchian, and Yang Wang. The multipath reliable connection (MRC) transport, 2026. URL <https://arxiv.org/abs/2606.18170>. 6.3.1
- [107] Neil T. Spring, Maureen Chesire, Mark Berryman, Vivek Sahasranaman, Thomas Anderson, and Brian Bershad. Receiver based management of low bandwidth access links. In *IEEE INFOCOM*, volume 1, pages 245–254 vol.1, 2000. doi: 10.1109/INFCOM.2000.832194. 1.1, 1.2.2.3, 1.3.3, 4.4.3
- [108] Ion Stoica, Scott Shenker, and Hui Zhang. Core-stateless fair queueing: Achieving approximately fair bandwidth allocations in high speed networks. *ACM SIGCOMM CCR*, 28(4):118–130, October 1998. ISSN 0146-4833. doi: 10.1145/285243.285273. URL <https://doi.org/10.1145/285243.285273>. 1.1.2
- [109] Srikanth Sundaresan, Neil Spring, and Yimeng Zhao. A fine-grained network traffic analysis with millisampler. <https://engineering.fb.com/2023/04/17/networking-traffic/millisampler-network-traffic-analysis/>, 2023. 2.3.2
- [110] Ammar Tahir and Radhika Mittal. Enabling users to control their Internet. In *USENIX NSDI 2023*, pages 555–573, Boston, MA, April 2023. USENIX Association. ISBN 978-1-939133-33-5. URL <https://www.usenix.org/conference/nsdi23/presentation/tahir>. 1.2.2.3, 1.3.2, 5.3.2, 5.6.1.1, 5.7
- [111] Martin Thomson and Cory Benfield. HTTP/2. RFC 9113, June 2022. URL <https://www.rfc-editor.org/info/rfc9113>. 5.2.2.1
- [112] Joseph D. Touch, Michael Welzl, and Safiqul Islam. TCP control block interdependence. RFC 9040, July 2021. URL <https://www.rfc-editor.org/info/rfc9040>. 5.2.2.1, 5.4.2, 5.7
- [113] Balajee Vamanan, Jahangir Hasan, and T. N. Vijaykumar. Deadline-aware datacenter TCP (D2TCP). In *ACM SIGCOMM 2012*, page 115–126, New York, NY, 2012. ACM. ISBN 9781450314190. doi: 10.1145/2342356.2342388. URL <https://doi.org/10.1145/2342356.2342388>. 2.1, 2.3.3
- [114] Vijay Vasudevan, Amar Phanishayee, Hiral Shah, Elie Krevat, David G. Andersen, Gregory R. Ganger, Garth A. Gibson, and Brian Mueller. Safe and effective fine-grained TCP retransmissions for datacenter communication. *ACM SIGCOMM CCR*, 39(4):303–314, August 2009. ISSN 0146-4833. doi: 10.1145/1594977.1592604. URL <https://doi.org/10.1145/1594977.1592604>. 4.4.3
- [115] Midhul Vuppapapati, Saksham Agarwal, Henry Schuh, Baris Kasikci, Arvind Krishnamurthy, and Rachit Agarwal. Understanding the host network. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM ’24, page 581–594, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400706141. doi: 10.1145/3651890.3672271. URL <https://doi.org/10.1145/3651890.3672271>. 6.3.1
- [116] Zirui Wan, Jiao Zhang, Yuxiang Wang, Kefei Liu, Haoyu Pan, and Tao Huang. Rethinking intra-

- host congestion control in RDMA networks. In *Proceedings of the 8th Asia-Pacific Workshop on Networking*, APNet '24, page 31–37, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400717581. doi: 10.1145/3663408.3663413. URL <https://doi.org/10.1145/3663408.3663413>. 1.1.3, 6.3.1
- [117] Weitao Wang, Masoud Moshref, Yuliang Li, Gautam Kumar, T. S. Eugene Ng, Neal Cardwell, and Nandita Dukkkipati. Poseidon: Efficient, robust, and practical datacenter CC via deployable INT. In *USENIX NSDI 2023*, pages 255–274, Boston, MA, April 2023. USENIX Association. ISBN 978-1-939133-33-5. URL <https://www.usenix.org/conference/nsdi23/presentation/wang-weitao>. 1.1, 1.2.2.2, 2.3.3, 2.5.2, 3.1
 - [118] Ranysha Ware, Matthew K. Mukerjee, Srinivasan Seshan, and Justine Sherry. Modeling BBR’s interactions with loss-based congestion control. In *ACM IMC*, IMC ’19, page 137–143, New York, NY, 2019. ACM. ISBN 9781450369480. doi: 10.1145/3355369.3355604. URL <https://doi.org/10.1145/3355369.3355604>. 1.1.4, 3.4.3, 5.1, 5.6.2.1, 6.3.2, 6.4.1
 - [119] Ranysha Ware, Matthew K. Mukerjee, Srinivasan Seshan, and Justine Sherry. Beyond Jain’s fairness index: Setting the bar for the deployment of congestion control algorithms. In *ACM HotNets 2019*, page 17–24, New York, NY, 2019. ACM. ISBN 9781450370202. doi: 10.1145/3365609.3365855. URL <https://doi.org/10.1145/3365609.3365855>. 6.3.2
 - [120] Ranysha Ware, Adithya Abraham Philip, Nicholas Hungria, Yash Kothari, Justine Sherry, and Srinivasan Seshan. CCAnalyzer: An efficient and nearly-passive congestion control classifier. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM ’24, page 181–196, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400706141. doi: 10.1145/3651890.3672255. URL <https://doi.org/10.1145/3651890.3672255>. 6.3.2, 6.4.1
 - [121] Haitao Wu, Zhenqian Feng, Chuanxiong Guo, and Yongguang Zhang. ICTCP: Incast congestion control for TCP in data center networks. In *ACM CoNEXT 2010*, New York, NY, 2010. ACM. ISBN 9781450304481. doi: 10.1145/1921168.1921186. URL <https://doi.org/10.1145/1921168.1921186>. 1.3.3, 2.1, 2.3.3, 3.1, 4.3, 4.4.2.2, 5.7
 - [122] Chen-Yu Yen, Soheil Abbasloo, and H. Jonathan Chao. Computers can learn from the heuristic designs and master Internet congestion control. In *ACM SIGCOMM 2023*, page 255–274, New York, NY, USA, 2023. ACM. ISBN 9798400702365. doi: 10.1145/3603269.3604838. URL <https://doi.org/10.1145/3603269.3604838>. 5.5.2.3
 - [123] YouTube. Youtube. <https://www.youtube.com>, February 2024. 5.2.2.1
 - [124] Adrian Zapletal and Fernando Kuipers. Slowdown as a metric for congestion control fairness. In *ACM HotNets 2023*, page 205–212, New York, NY, USA, 2023. ACM. ISBN 9798400704154. doi: 10.1145/3626111.3628185. URL <https://doi.org/10.1145/3626111.3628185>. 5.2.1
 - [125] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. High-resolution measurement of data center microbursts. In *ACM IMC 2017*, page 78–85, New York, NY, USA, 2017. ACM. ISBN 9781450351188. doi: 10.1145/3131365.3131375. URL <https://doi.org/10.1145/3131365.3131375>. 2.3.2
 - [126] Alexander Zimmermann, Arnd Hannemann, and Tim Kosse. Flowgrind - a new performance measurement tool. In *2010 IEEE Global Telecommunications Conference GLOBECOM 2010*, pages 1–6, 2010. doi: 10.1109/GLOCOM.2010.5684167. 4.5.1